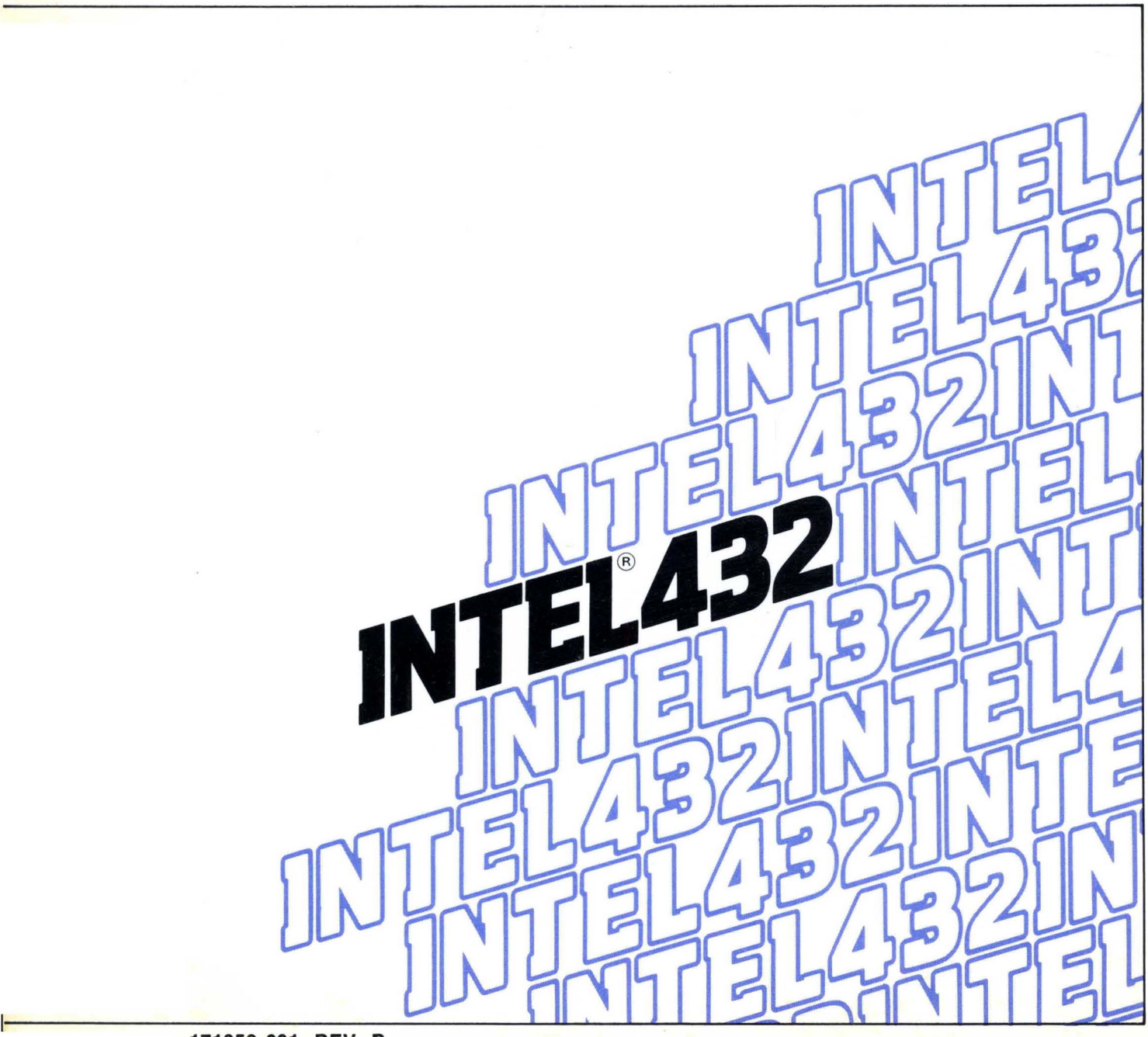
intel

iAPX 432 Object Primer



171858-001, REV. B

iAPX 432 OBJECT PRIMER

Manual Order Number: 171858-001 Rev. B

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 TO REVISION B

Revision B modifies the original issue (August 1980) in these substantive ways:

Chapter 1	INTRODUCTION	There is new material introducing 432 architectural innovations.
Chapter 2	WHAT'S AN OBJECT?	Primitives are not described as "simple objects".
Chapter 4	INTERPROCESS COMMUNICATION	Non-FIFO queuing disciplines are mentioned. There is no WAIT TO RECEIVE OR N TIME QUANTA instruction.
Chapter 7	THE I/O SUBSYSTEM	The I/O example is changed because the Interface Processor can't create 432 objects, and because there can be multiple IP processes.

There are also these changes in terminology:

SEND MESSAGE is now SEND.

WAIT TO RECEIVE MESSAGE is now RECEIVE.

TRANSFORMER OBJECT is now DESCRIPTOR CONTROL OBJECT.

LABEL OBJECT is now TYPE DEFINITION OBJECT.

"Seal" is now "private type".

"Trademark" is now "public type".

TABLE OF CONTENTS

TABLE OF CONTENTS CONTINUED

TABLE OF CONTENTS CONTINUED

CHAPTER 7 THE I/O SUBSYSTEM

Chapter 1

INTRODUCTION

This book focuses on objects, and on how the 432's system objects support such high-level functions as concurrent processing, modularity, and type checking.

A broader discussion of the iAPX 432 can be found in:

Introduction to the iAPX 432 Architecture
Manual Order Number 171821-001

This book provides a conceptual overview of objects and their use in the 432 architecture, and does not explain implementation details. The definitive references for the 432 processors are:

iAPX 432 General Data Processor
Architecture Reference Manual
Manual Order Number 171860-001

and

iAPX 432 Interface Processor
Architecture Reference Manual
Manual Order Number 171863-001

WHY INVENT A NEW ARCHITECTURE?

Decreasing hardware costs have changed the economics of complex computer systems -- software costs are now a major part of overall costs for many systems. As a result, any computer application that requires extensive software development needs a computer architecture designed to reduce software costs. In fact, this is the fundamental design objective of the 432:

> DEFINE A NEW COMPUTER ARCHITECTURE THAT SIGNIFICANTLY REDUCES THE COST OF SOFTWARE.

A SOFTWARE-ORIENTED ARCHITECTURE

The 432 is based on a complete rethinking of what hardware facilities are needed to reduce the costs of software development, testing, and maintenance. It is a radical departure from conventional architectures:

1. The 432 provides a wide range of system performance without software changes. 432 software runs on 1-, 2-, or many-processor systems without software changes. Higher performance can often be achieved by simply adding more processors.

2. The 432 standardizes and speeds up important operating system functions by placing them in hardware. These hardware functions include storage allocation, and scheduling and communications for multiple software tasks. The 432's support of these functions makes possible more dynamic information and program structures, with reliability still provided by the 432's protection and checking mechanisms.

3. There are important software errors that are <u>impossible</u> on the 432:

   - A module can never maliciously or accidentally access data outside the set of data that it has a "need-to-know."

   - A module cannot write a data structure that it should only read, nor read a data structure that it should only write.

   - A module cannot perform an operation that is not allowed for the type of data being operated on (such as executing data as instructions).

   The major facilities of a conventional architecture are:

   - the processor registers

   - the various addressing modes

   - the built-in data types (e.g., bytes and words), and the built-in operations on those types (e.g., add or compare)

The 432 architecture, on the other hand, raises the level of the hardware/software interface. It uses the most advanced semiconductor technology (over 100,000 transistors on one chip) to implement the most advanced ideas in computer architecture. The 432's major facilities are ones that conventional systems try to provide (but provide inadequately) in operating systems and compilers:

- provide access-checking and protection, to enforce modularity;
- provide the execution environment for program modules;
- provide scheduling and communications for multiple software tasks;
- provide control and dispatching for multiple hardware processors.

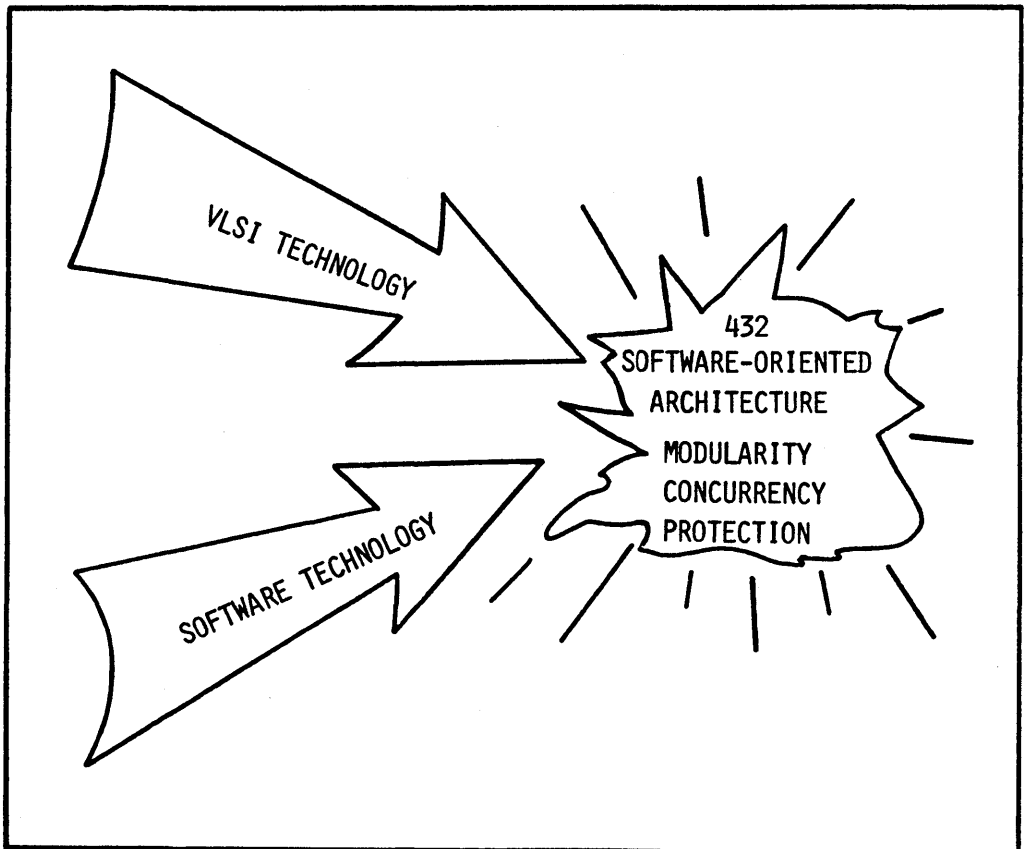


Figure 1.1 -- Two state-of-the-art technologies merge to raise the level of the hardware/software interface.

WHAT YOU WILL LEARN

First you will learn what an object is. The object concept is then used to explain the major facilities of the 432 architecture.

This book has six other chapters:

CHAPTER	DESCRIPTION
2. WHAT'S AN OBJECT?	How objects evolved, what they are, and the symbols we use to represent them.
3. PROGRAM STRUCTURE	An overview of the hardware-recognized objects used to structure a program (i.e., what objects are used to execute a program).
4. INTERPROCESS COMMUNICATION	It is often advantageous to structure a program so that several parts of it can execute in parallel. This chapter describes the facilities of the 432 architecture that allow these separate parts to communicate and synchronize with each other.
5. TRANSPARENT MULTIPROCESSING	Any software written for the 432 can run on systems with 1, 2, or many processors. No software changes are required. This chapter explains how we did it.
6. DESIGNING SOFTWARE SYSTEMS	This chapter describes object-oriented design and the support that the architecture provides for it. This method can reduce software costs.

CHAPTER	DESCRIPTION
7. THE I/O SUBSYSTEM	This chapter describes the physical partitioning of a 432 system into major functional blocks (i.e., processors, memory, and I/O subsystems), and explains how I/O is handled by decentralized I/O subsystems.

This booklet is an "Object Primer" because all of these facilities are based on objects; this is why objects are explained before any of the other facilities.

SELF-TESTS

The Object Primer is designed for self-teaching, and includes a one-page quiz at the end of each subsequent chapter, with answers when you turn the page.

Chapter 2

WHAT'S AN OBJECT?

This chapter answers that question by covering these subjects:

- PRIMITIVE DATA TYPES
- 432 PRIMITIVE DATA TYPE SUPPORT
- EVOLUTION OF OBJECTS
- DEFINITION OF OBJECTS
- SYMBOLS FOR OBJECTS

PRIMITIVE DATA TYPES

To know what a certain pattern of bits in a computer memory means, you must know what type of information it is; i.e., is it an integer, a character, or a real? For example, consider the following 8-bit value:

0100 0001

If this is an ASCII character it means: 'A'
If this is an integer it means: 65

In short, the data's type determines how the data is interpreted.

Integers, characters, and reals are examples of very simple data called primitives. Primitives have three important characteristics:

PRIMITIVES

- Are data structures that contain information in an organized manner.
- Have a set of basic operations defined for them that directly manipulate the data structure, e.g., ADD INTEGER, or MULTIPLY REAL.
- Each primitive can be referenced (addressed) as one thing; you don't need to reference each of the parts.

In the 432, these simple data structures are called primitives to differentiate them from the more complex structures called objects. Just as there are different types of primitives (integers, reals, etc.), there are different types of objects. I'll discuss the precise differences between an object type and a primitive type in a moment, but for now just visualize an object type as a more complex primitive data type. Let's take a side track for a moment to look at the support the 432 provides for primitive data types, then I'll come back to the definition of object types.

432 PRIMITIVE DATA TYPE SUPPORT

The 432 supports these eight primitive data types:

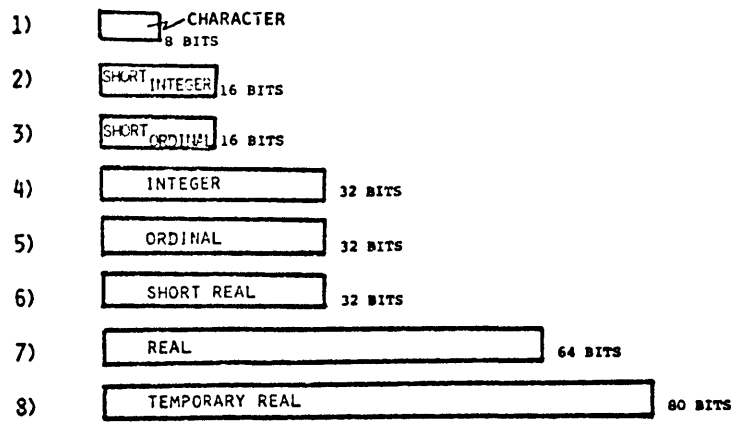
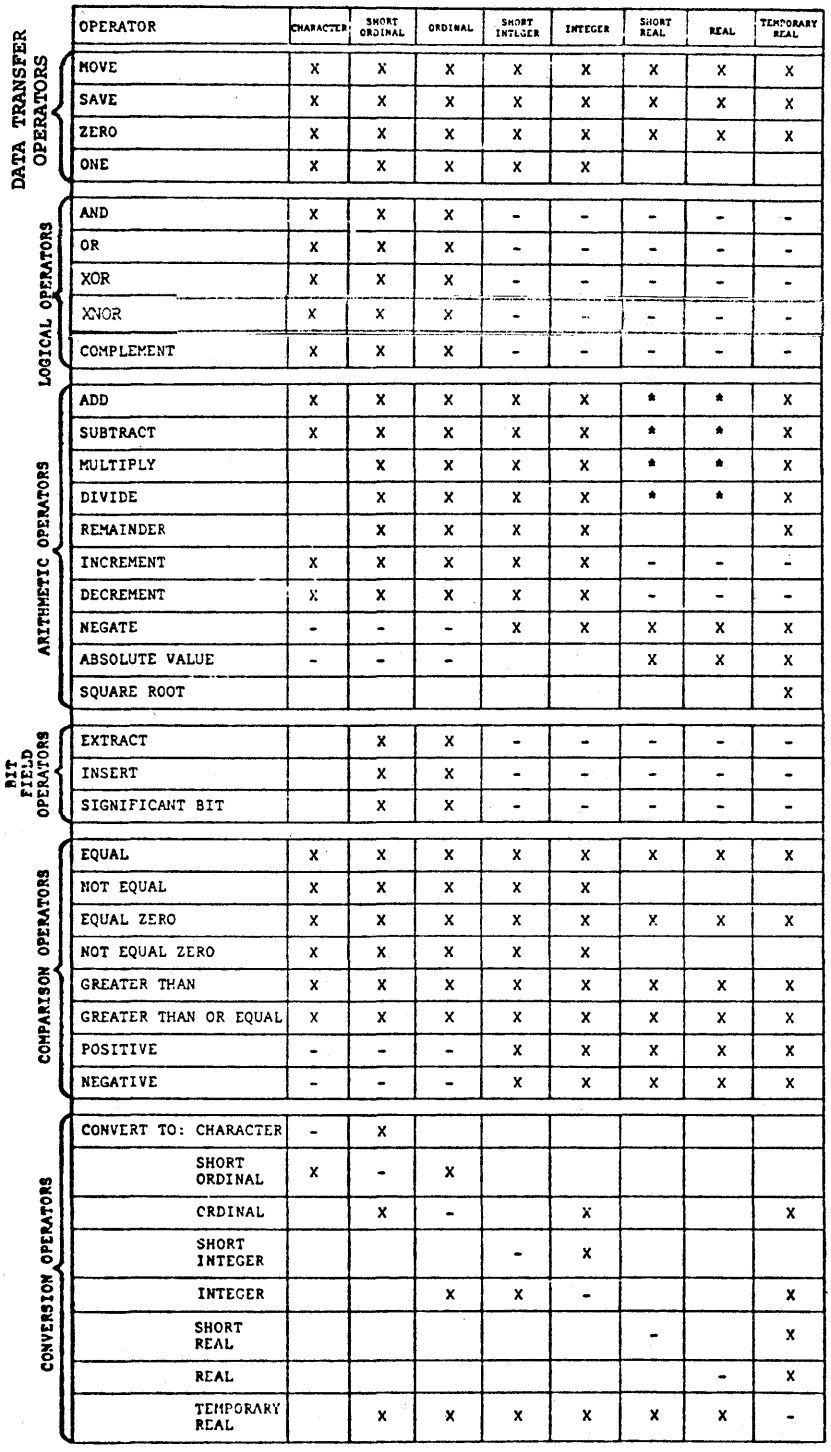


Figure 2.1 -- Primitive data types

This support includes more than 150 unique* operators for these eight types. Figure 2.2 on the next page summarizes the operators available for these types.

*System management and branching operators are not included in this count. Instructions which are the same for two different data types (e.g., MOVE INTEGER and MOVE ORDINAL) are only counted once. Also, all instructions have multiple addressing modes but are only counted once.



	OPERATOR	CHARACTER	SHORT ORDINAL	ORDINAL	SHORT INTEGER	INTEGER	SHORT REAL	REAL	TEMPORARY REAL
DATA TRANSFER 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	X	X	X	-	-	-	-	-
	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	-	-	-	-	-
COMPARISON 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.

Figure 2.2 -- Primitive operator summary

EVOLUTION OF OBJECTS

> OBJECTS RAISE THE LEVEL OF
> THE HARDWARE/SOFTWARE INTERFACE.

This is easy to understand if you look at the history of computer development.

Early machines had very simple hardware operations such as "move byte" and "add integer" which manipulated hardware-recognized data types such as bytes and integers. The hardware on these early machines was not capable of manipulating floating-point numbers. If you needed floating-point you had to implement it in the software.

As technology progressed and computer hardware gained functionality, more complicated operations such as floating-point add, multiply, divide, etc., were moved into the hardware. This increase in hardware functionality increased the speed with which more complicated operations (such as floating-point) could be handled, and eliminated the need to program these operations in software. Of course, moving "software" into hardware had other benefits, e.g., the enforcement of standard programming methods.

The 432 carries this progression one step further by placing system management operations (such as process scheduling, memory management, and interprocess communication) into the hardware where they can be handled quickly and more securely.

Just as moving floating-point operations into the hardware meant that the hardware had to manipulate the data structures used to represent floating-point numbers, moving system management operations into the 432 hardware means that the 432 has to manipulate the data structures associated with process scheduling, memory managment, and interprocess communication. These data structures are called objects.

The following chart summarizes the evolution of object-oriented machines.

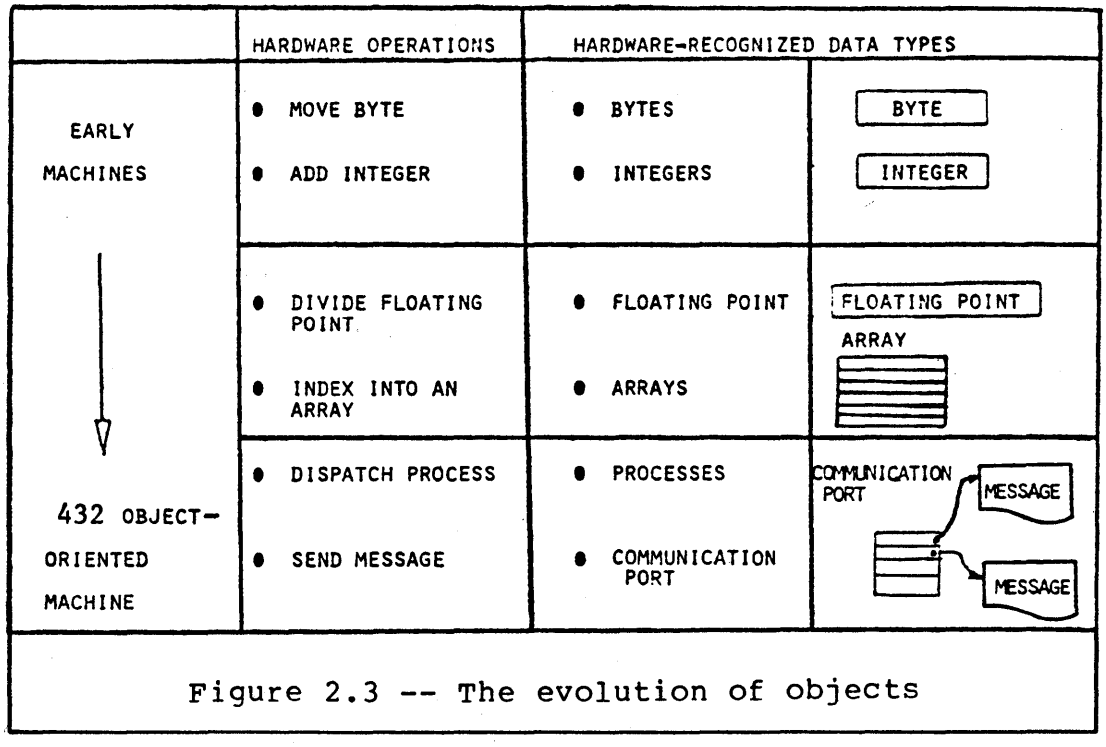


Figure 2.3 -- The evolution of objects

Examples of some of the objects used by different 432 system management operations can be found in the following chart:

OPERATION	OBJECTS
• Process scheduling	• Process object • Processor object • Dispatching port object
• Dynamic memory allocation	• Storage resource object
• Interprocess communication	• Communication port object

Figure 2.4 -- Some objects

Don't worry if the object names (such as dispatching port) sound mysterious; their functions are explained later. The important point to remember now is that objects are simply data structures in memory which may be manipulated in controlled ways by hardware and/or software.

This brings up an important point. Not all objects are manipulated only by hardware operations; some are manipulated by a combination of hardware and software operations and some are manipulated only by software operations. Wherever possible, frequently-used or time-critical operations have been placed in the hardware and less-frequently-used operations have been left to the software

DEFINITION OF OBJECTS

The early part of this chapter explained that primitives have three important characteristics. What are those three characteristics? (Fill in the blanks.)

(1) _____

(2) _____

(3) _____

(Check your answers by reviewing page 2-2)

An object has these same three characteristics plus one more: An object has a label that tells its type.

This label is used to check an object's type before it is used in an operation. I'll say more about type checking in Chapter 6. The important thing to remember is that the difference between an object and a primitive is that an object has a label which tells its type. A primitive does not have a label. In summary:

AN OBJECT

- Is a data structure that contains information in an organized manner.
- Has a set of basic operations defined for it that directly manipulate the data structure. (The 432 hardware ensures that these are the only operations that can directly manipulate the data structure.)
- Can be referenced (addressed) as one thing; you don't need to reference each of the parts.
- Has a label that tells the object's type.

Figure 2.5 -- Object characteristics

SYMBOLS FOR OBJECTS

This book uses various symbols to represent objects.

For example, the rather shapeless symbol:

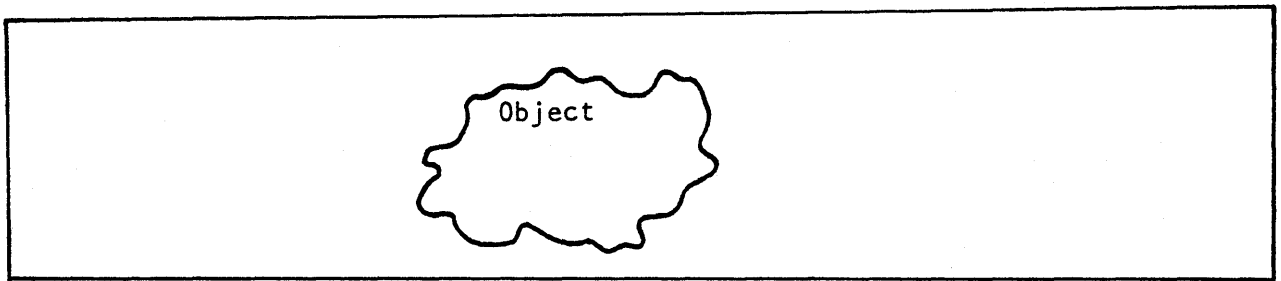


Figure 2.6 -- Any object

represents an object when we don't know or don't care what its type is. More definite symbols such as:

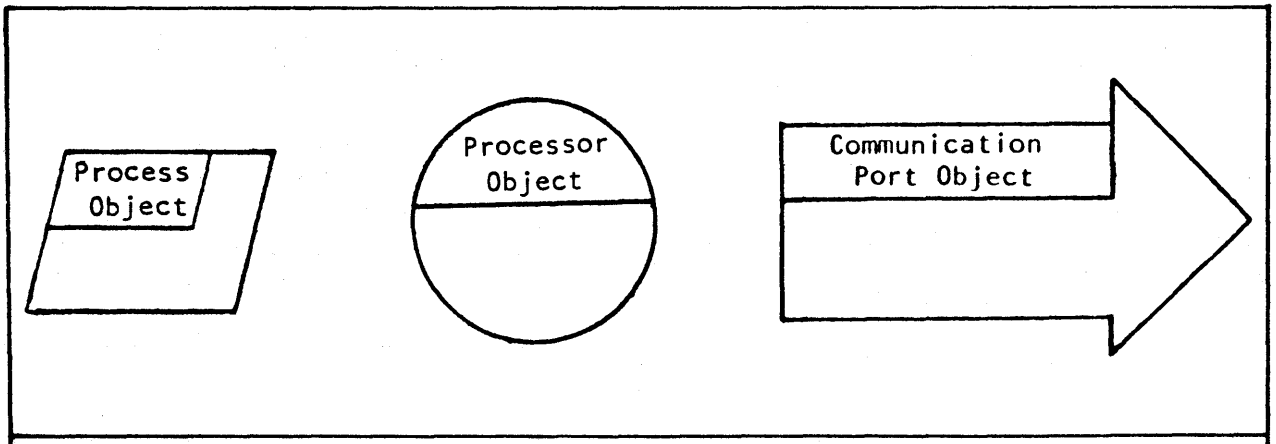


Figure 2.7 -- Specific object types

represent specific types of objects. Remember that all of the above symbols represent objects (data structures) - not chunks of silicon.

The one symbol which represents silicon is the processor symbol:

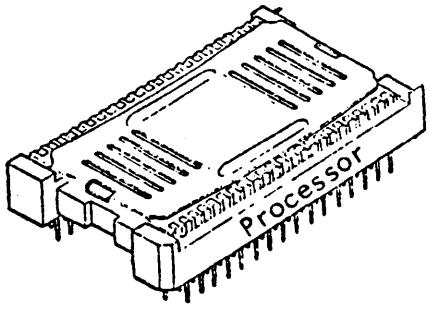


Figure 2.8 -- Physical processor

which represents a physical processor - not a data structure (not an object).

The other symbol which is used frequently is the arrow:

Figure 2.9 -- Object reference

which represents a protected object reference.

The 432 is different from most machines because it has an object-oriented addressing and protection mechanism. This method of addressing allows a program to address an object only if it has a reference for it. You'll learn more about this when I talk about a procedure's access environment in the next chapter. For now, the important thing to remember is that the arrow represents an object reference which gives a program the ability to access the object pointed to by the reference.

For example, the figure:

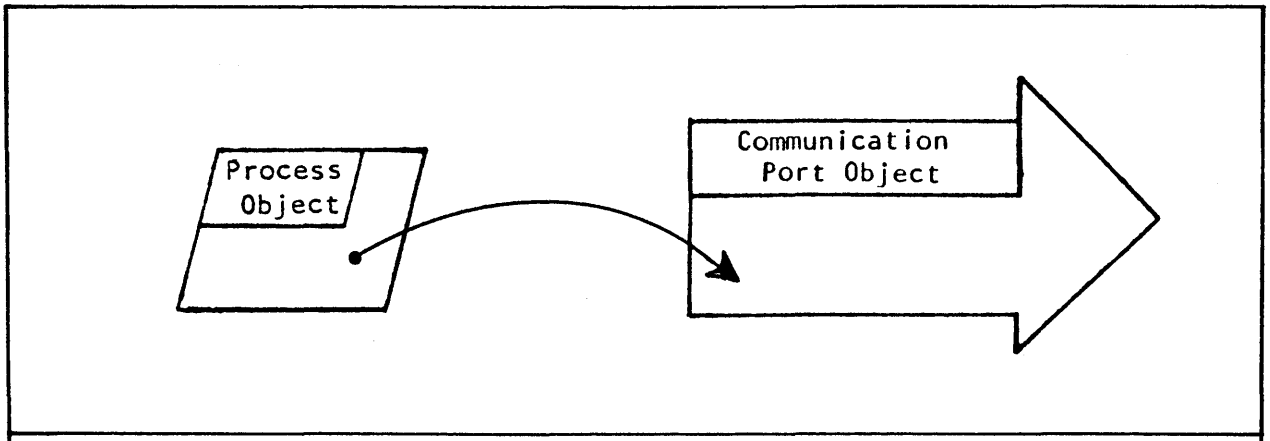


Figure 2.10

indicates that the process object can access the communication port object because it has a reference for it.

OBJECTS VS. SEGMENTS

An object is a data structure of arbitrary complexity. What I mean by "arbitrary complexity" is that an object does not have to occupy a contiguous set of memory locations. An object can look like this:

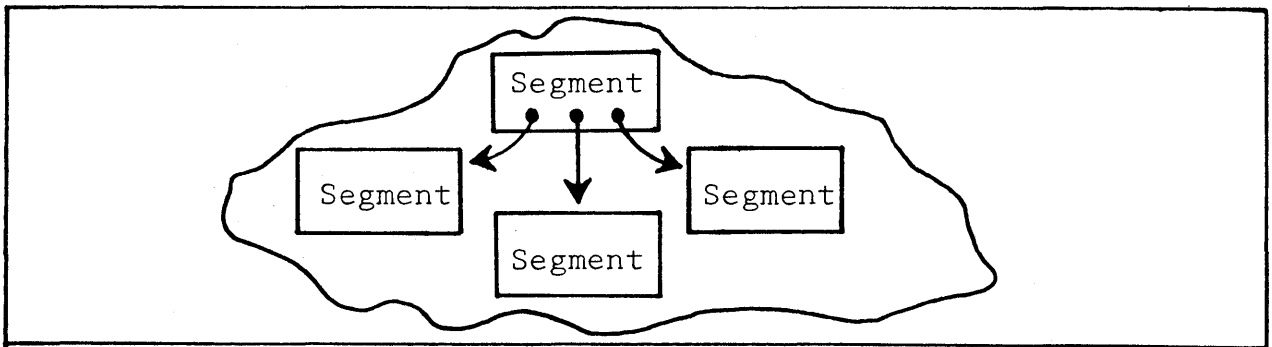


Figure 2.11 -- An object can contain several segments.

Where each of the rectangles in Figure 2.11 represents a different physical region in memory.

A set of contiguous memory locations (a rectangle in Figure 2.11) is called a segment. Segments should not be confused with objects. An object can be a segment, several segments, or part of a segment.

The term segment is used to talk about the physical structure of data in memory, i.e., where the structure is located.

The term object is used to talk about the logical structure of data in memory, i.e., how the memory is used.

Every bit, byte, and word of information in memory is contained in an object of some kind. That same bit, byte, or word of information is also contained in a segment.

Segments are not discussed in this book; I only talk about objects in this book. I'm warning you about the difference between segments and objects because it is easy to confuse the two terms.

"WHAT'S AN OBJECT?" QUIZ

1. Integers, characters, and reals are all examples of _____.

   Communication ports and dispatching ports are both examples of _____.

2. Objects came about as part of the "natural" evolution of more powerful computers. Explain.

3. Circle the correct answer. Operations on objects are performed by the:

   - Hardware
   - Software

   Explain.

4. What are the four characteristics of an object? Which of these four is not a characteristic of a primitive?

KEY TO "WHAT'S AN OBJECT?" QUIZ

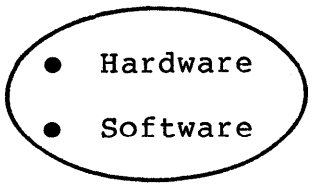
1. Integers, characters, and reals are all examples of primitives.

   Communication ports and dispatching ports are both examples of objects.

2. Objects came about as part of the "natural" evolution of more powerful computers. Explain.

   As our ability to build computer hardware has improved, we've moved functionality from the software to the hardware. As we've done this, the hardware has come to recognize more complex data types. An example of this is moving the functionality for floating point operations into the hardware -- the hardware had to "learn" to recognize the structure of a floating point number. The 432 has moved operating system and high-level language operations (e.g., process scheduling) into the hardware -- thus the 432 had to "learn" to recognize the data structures needed for these operations. We call these more complex structures "objects".

3. Circle the correct answer. Operations on objects are performed by the:

   Explain.

   The 432 hardware recognizes many objects and has many instructions for manipulating them. In general, these hardware operations are the ones executed frequently (so they need to be fast) and the ones that are sensitive (so they need to be protected).

   This is not to say, however, that software cannot manipulate objects -- it can. In fact the 432 includes a mechanism for defining additional "software instructions" for manipulating objects. This mechanism is covered in Chapter 6.

4. What are the four characteristics of an object? Which of these four is not a characteristic of a primitive?

   The four characteristics of an object are:
   - it is a data structure that contains information in an organized manner;
   - it has a set of operations that manipulate it;
   - it can be referenced (addressed) as one thing;
   - it has a label that tells its type.

   Primitives have the first 3 characteristics, but do not have a type label.

Chapter 3

PROGRAM STRUCTURE

This chapter is an overview of the structure of a 432 program. Later chapters fill in some of the details and explain how some of the objects are used. The major purpose of this chapter is to put it all into perspective so you can see how the pieces fit together.

CHAPTER ORGANIZATION

You start your journey through the structure of a program with the:

- Physical processors . . . . the chunks of silicon that fetch and execute instructions

Along the way you briefly explore each of the objects which make up a program:

- Processor objects . . . . . contain information about the state of a particular physical processor
- Process objects . . . . . contain information about a particular "unit of work for a processor" (process)
- Context objects . . . . . are activation records for instances of procedures
- Instruction objects . . . . contain the instructions that are fetched and executed by a processor
- Data objects . . . . . contain the data manipulated by a program

At this point, you may wonder "Why haven't I seen a discussion of program structure in the descriptions of other architectures?" The reason is simple. In most machines this structure is part of the operating system or the language run-time environment. But the 432 is not a "flat" machine; these structures are part of the 432 architecture and are recognized by the hardware.

The structure of a program starts with:

PHYSICAL PROCESSORS

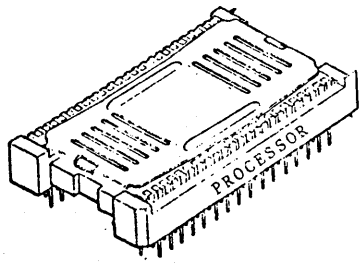
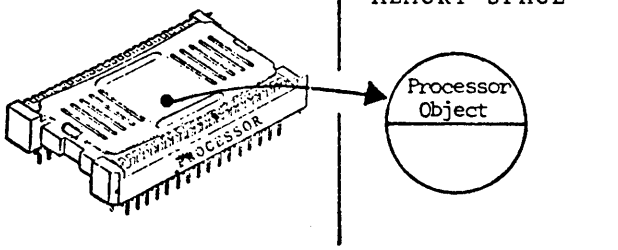


Figure 3.1 -- A physical processor

This symbol (Figure 3.1) represents a physical processor. The physical processor is not an object (it is not a data structure), although, as you'll see in the next section, there is an object called a processor object. The physical processor, however, is made of silicon, sits inside a package on a printed circuit board, and fetches and executes instructions.

PROCESSOR OBJECTS

Each physical processor has, in memory, a processor object (see Figure 3.2). This is used to record information about a particular processor such as whether it is running or halted, diagnostic and machine check information, and references for other objects which the processor needs.



A Processor Object:

- Is a hardware-recognized data structure in memory
- Each processor has one
- Contains information like:
  - Processor status (e.g., running, halted, etc.)
  - Diagnostic and machine check information
  - References for other objects

Figure 3.2 -- The processor object is part of the structure of a program.

For those of you familiar with earlier mainframe implementations, some of them used low-order memory to record diagnostic and status information about the processor. But they had a problem when they started to build a multiprocessor system -- each processor thought that it should use the same set of locations in low-order memory. One way that the problem was solved was by creating two separate banks of low-order memory and adding special instructions to the processors for switching between the two banks of low-order memory.

The 432 has taken this concept of storing diagnostic and status information about a processor and generalized it for use with multiple processors. Each processor has its own processor object (see Figure 3.3) which can be located anywhere in memory and can be dynamically relocated if necessary. Each processor addresses its processor object using an on-chip reference for the processor object.

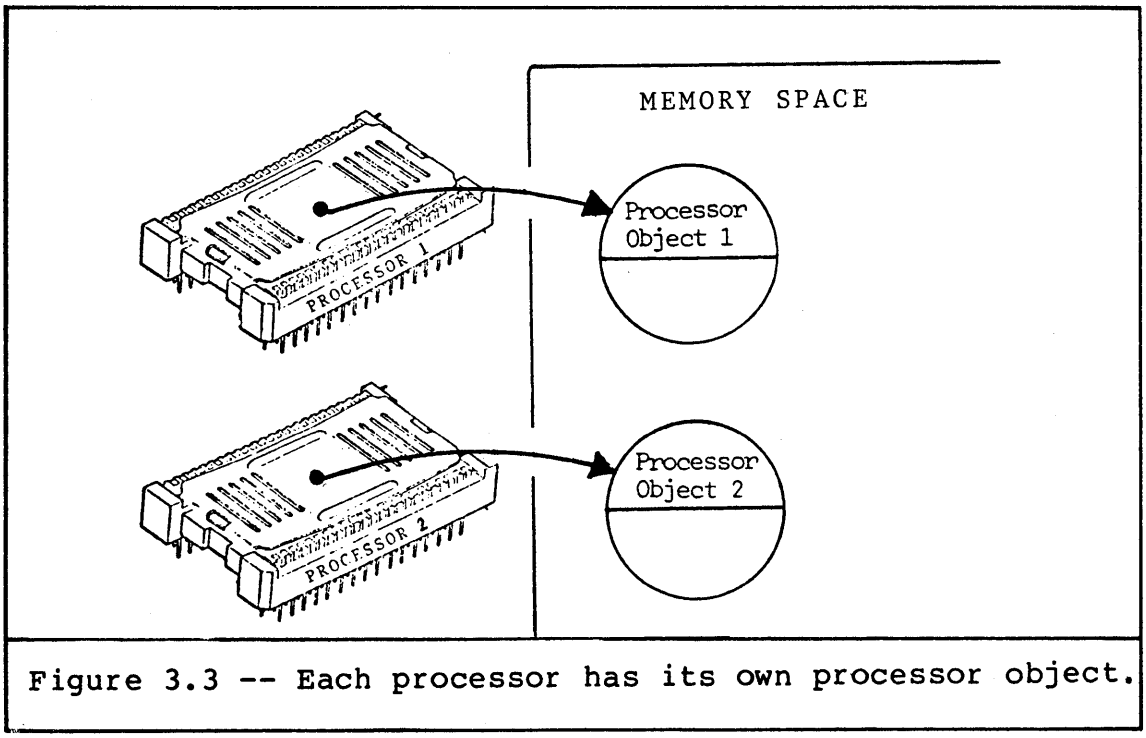


Figure 3.3 -- Each processor has its own processor object.

(Note: For those of you wondering about the chicken-and-the-egg problem, the processor loads its processor object reference in a rather straightforward manner at initialization time.)

In summary, processor objects are really simple. The important thing to remember is that a processor object contains information about a particular physical processor.

PROCESSES

The next stop on this journey through the structure of a program is the process, and, you guessed it, the data structures that contain information about processes are called process objects. But before I talk about the kind of information found in a process object, I'll define "process":

PROCESS - A unit of work for a processor, i.e., the smallest unit of programming activity which can be scheduled to run on a processor

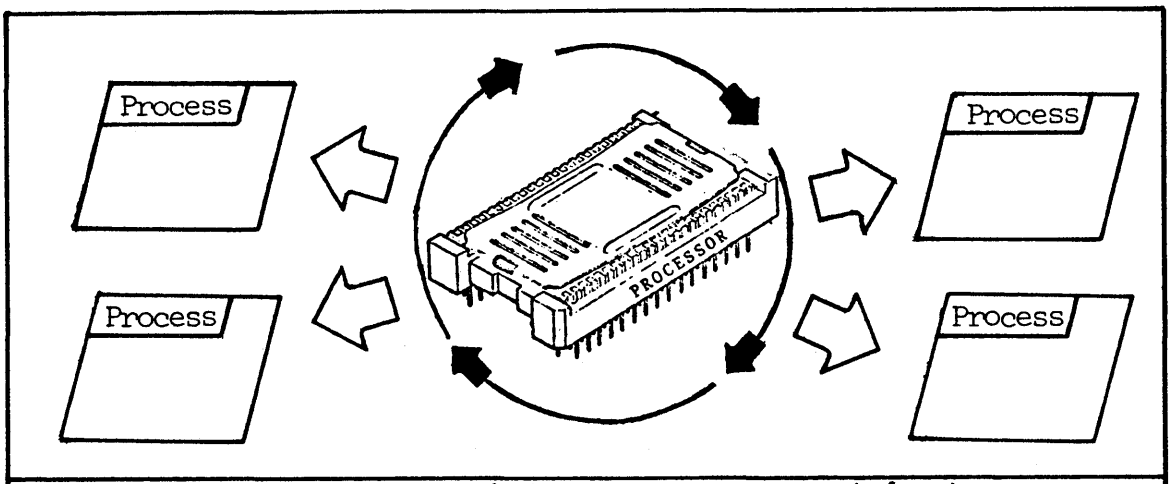


Figure 3.4 -- Multiple processes can take turns on a processor.

In a system with two processes that are ready to run, they can either:

- "take turns" executing on a single processor (i.e., timeshare), or
- execute at the same time if there are two processors available.

As an example, consider a single processor timesharing system with two users (Bob and Tom) sitting at the terminals. Bob is editing some text and Tom is compiling a COBOL program. In the example, there are two processes taking turns using the processor. Bob's editing process and Tom's COBOL compiler process are scheduled to share the processor on a timesharing basis (see Figure 3.5).

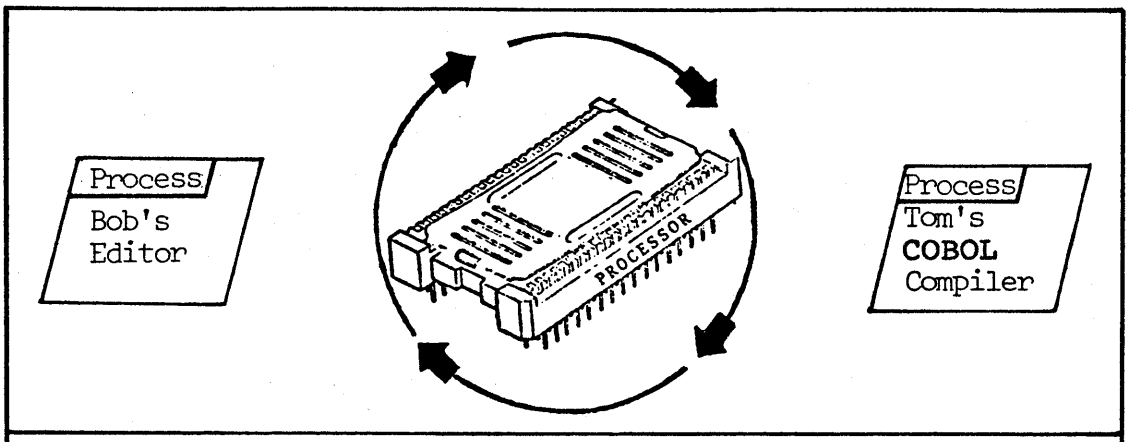


Figure 3.5 -- Timesharing processes

Bob and Tom have separate processes that share the same processor, but the two processes are completely unrelated; they do not work together to accomplish a common end. The next example examines a single program with three processes that work together.

In industrial control applications, there are many events which can take place simultaneously. Structuring the program to reflect this natural parallelism makes it easier to understand and debug.

The industrial control program example uses three separate processes. The first monitors all the sensors and performs preliminary pre-processing of the data. This is called the sensor monitor process.

The second process receives the pre-processed data from the sensor monitor process and uses it to make control decisions. This is called the control decision process.

The third process receives commands from the control decision process and translates them into specific instructions for each of the servo devices that position the controls. This is called the servo controller process.

This program structure is summarized in Figure 3.6.

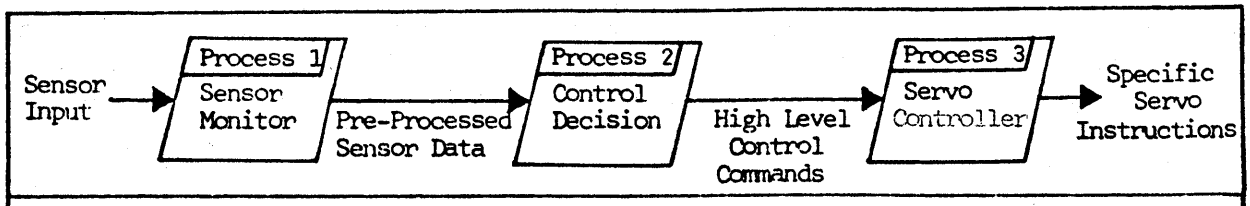


Figure 3.6 -- The structure of an example industrial control program

Note that these three processes can execute in parallel. While the sensor monitor is preparing a message for the control decision process, the control decision process can be computing the action required by data received earlier. At the same time, the servo controller can be carrying out the commands it received earlier. The program is structured so that three different parts can be executing on different processors at the same time.

However, the hardware used to execute this industrial control program could be a single processor that shares its time among the three processes. Or, if more computational power is required to make the control program execute faster, a second or third processor can be added to execute more than one process at the same time (see Figure 3.7).

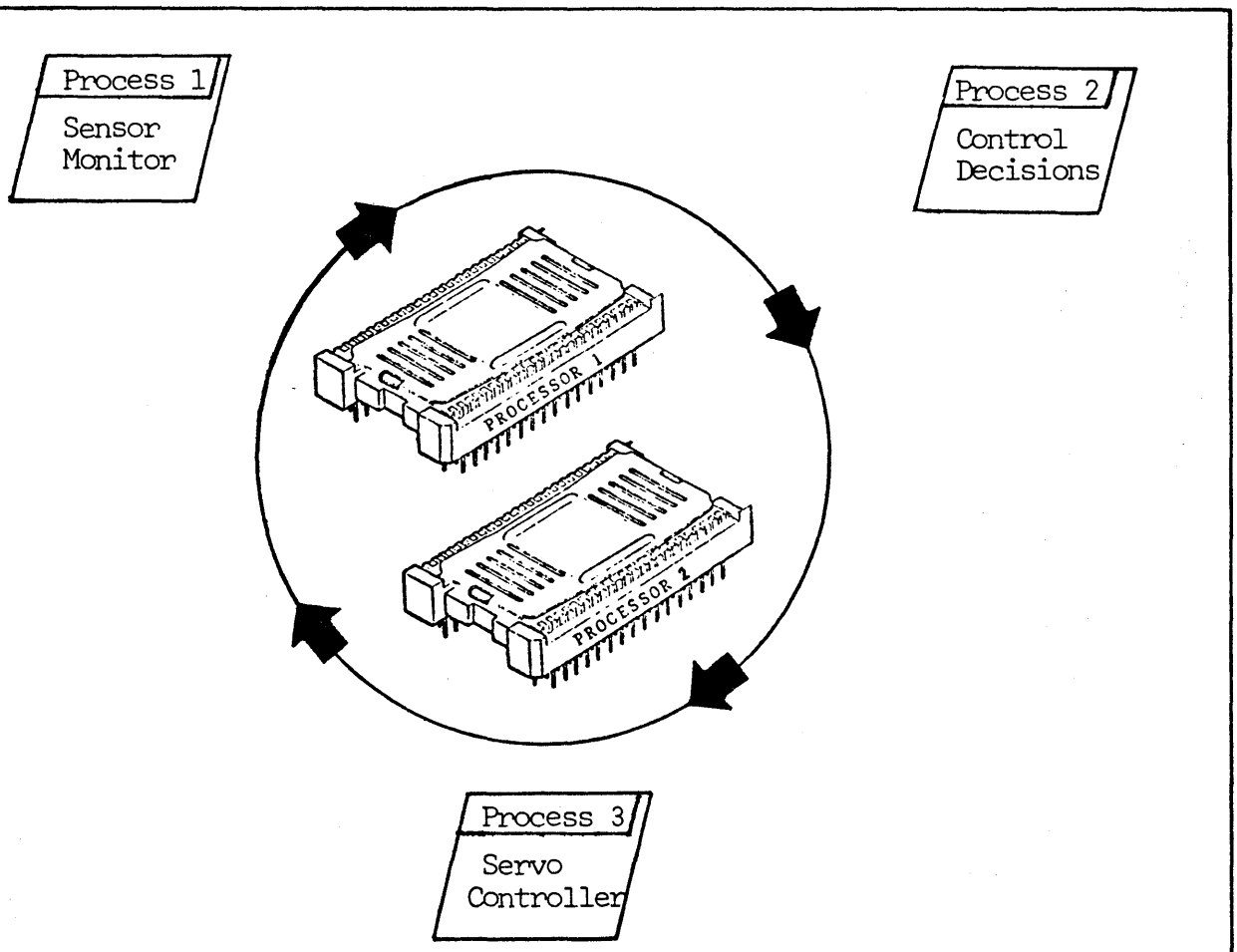


Figure 3.7 -- A 3-process industrial control program running on 2 processors

This is called transparent multiprocessing because processors can be added without making any software changes. There is more about it in Chapter 5. For now, just realize that a process is the unit of work that can be scheduled to run on a processor.

PROCESS OBJECTS

A process object is the data structure in memory that contains information about a particular process. There is one for each process in the system and it contains information like how the process should be scheduled and what the process status is (running, waiting, etc.) (see Figure 3.8).

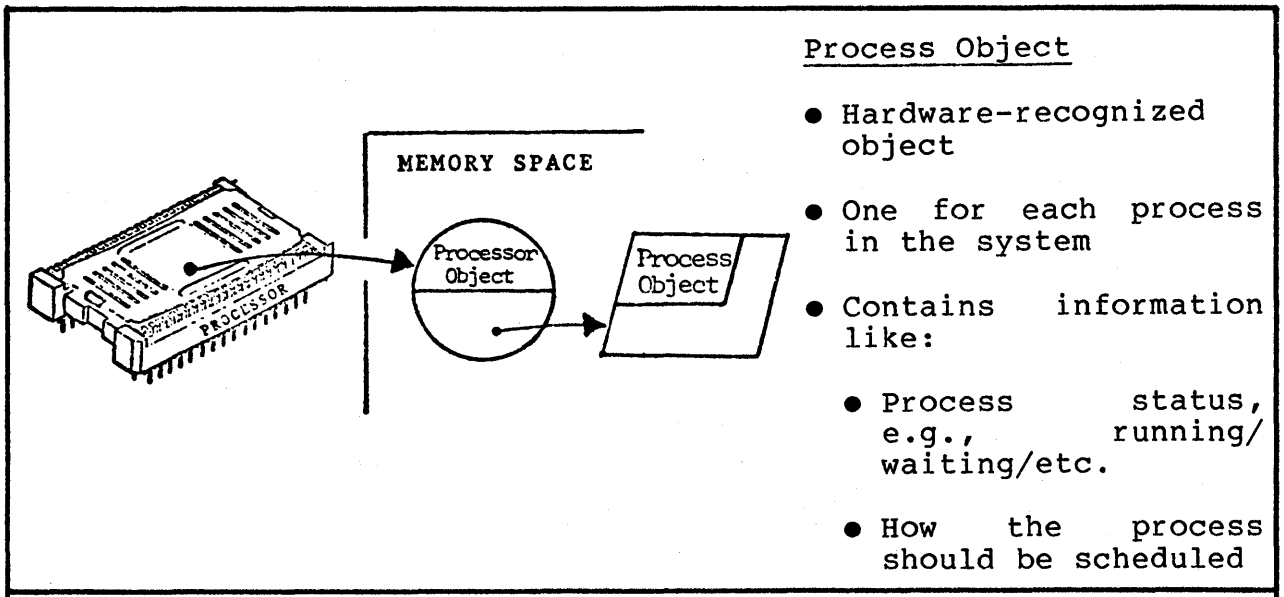


Figure 3.8 --The process object is part of the structure of a program.

Figure 3.8 shows that the processor object contains an object reference for the process object currently running on the processor. This is important because the 432's object-oriented addressing and protection mechanism does not allow a processor to access an object unless it has an object reference for it.

This object reference from the processor object to the process object is not fixed. It is changed dynamically as the processor switches among different processes.

As an example, reconsider Bob and Tom, the two users timesharing at computer terminals. In Figure 3.9 at Time 1, Bob's process is running and the processor object has a reference for Bob's process. At Time 2, Tom's process is running and the processor object has a reference for Tom's process.

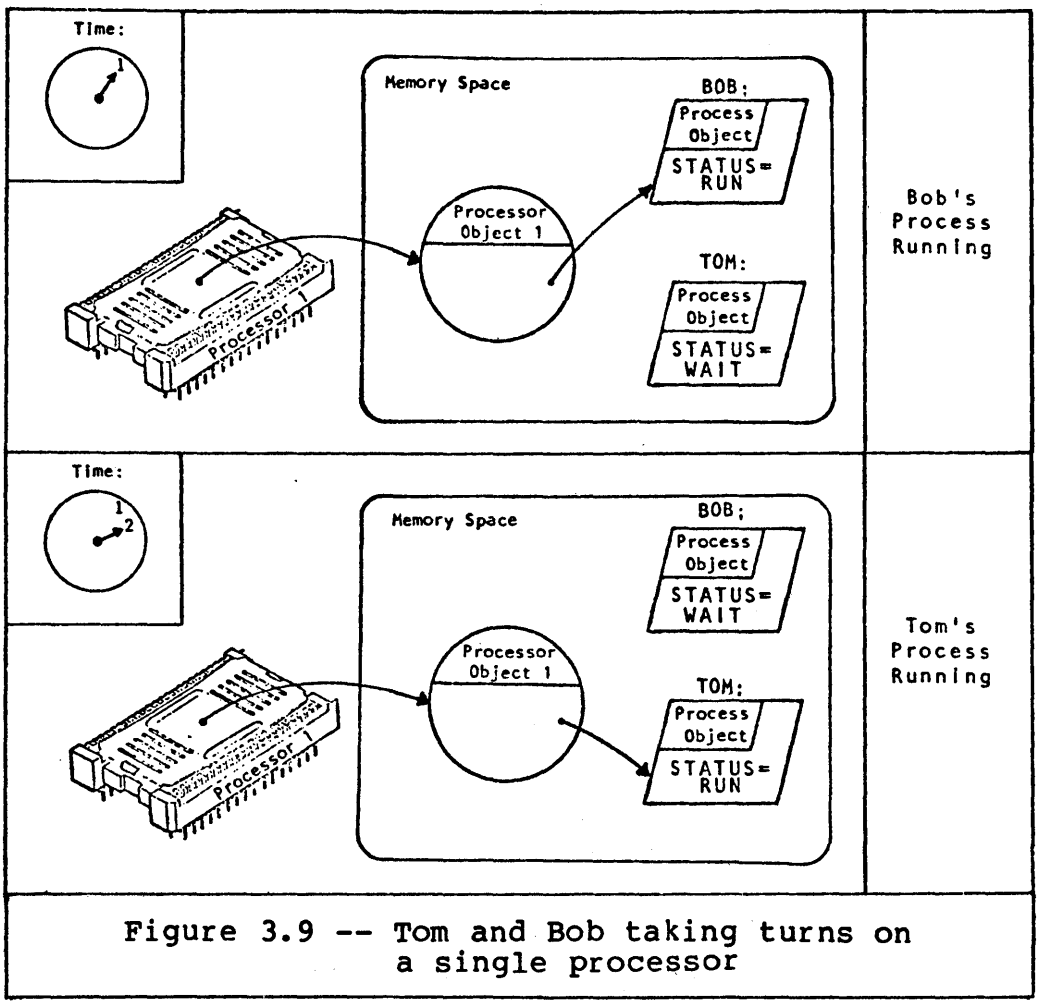


Figure 3.9 -- Tom and Bob taking turns on a single processor

Note also that the process object contains the status information showing whether its process is running or waiting.

One last note about Figure 3.9 -- it is oversimplified. It does not show how the object reference gets switched from Tom's process to Bob's process, so don't get confused. The processor at Time 2 doesn't just arbitrarily give itself the ability to execute a different process. The details of how this works are explained in Chapter 5, "Transparent Multiprocessing". For now, just remember one thing:

> A processor object has a reference for the process object of the process currently being executed. This reference changes dynamically as the processes being executed change.

Figure 3.9 shows how Bob and Tom can both share one processor. Figure 3.10 shows how things are structured if Bob and Tom each have their own processor. There is a processor object for each processor and each processor object has a reference for the process currently running on its processor.

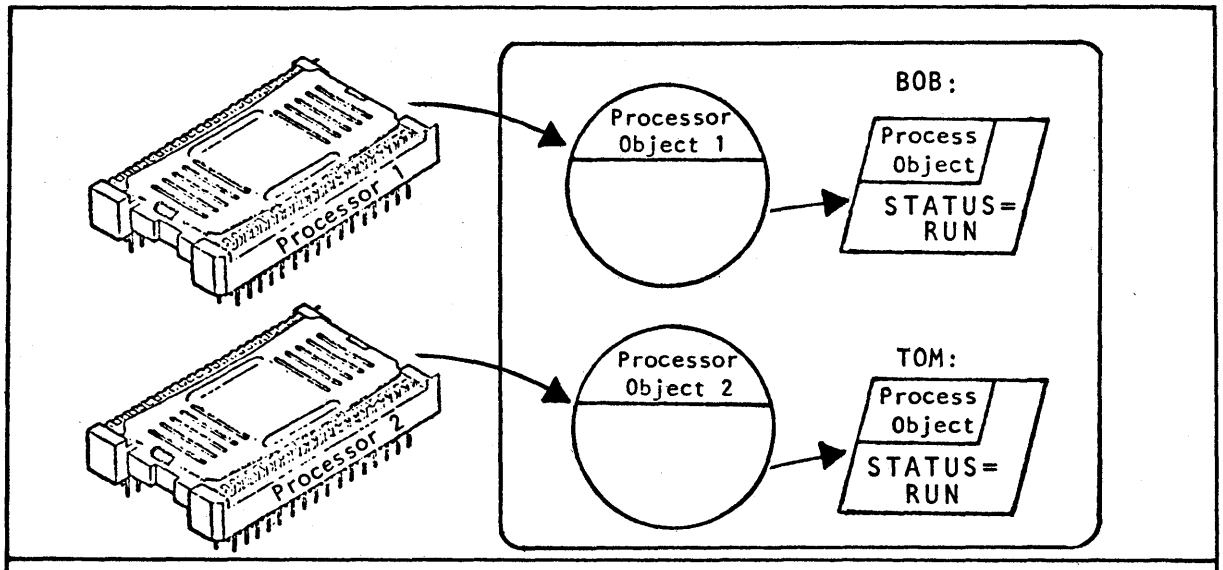


Figure 3.10 -- A two processor system

PROCEDURES

Each process is made up of one or more procedures and, as you might expect, the 432 has a hardware-recognized object that represents a procedure. It is called a context object, but before I explain what a context object is I need to say a few things about procedures.

A procedure is a collection of instructions to perform some operation, such as sine of x. A procedure is called with some parameters, performs some operation, and optionally returns a result.

In a computer system, several processes that need to perform the same operation (e.g., sine of x), and they may want to share the same procedure. Procedures are shared among several different processes by giving each process a "copy" (an instance) of the procedure. Actually, each "copy" does not duplicate all of the information because some of the information (e.g., the instructions) can be used simultaneously by more than one process. Each instance of a procedure (each "copy") is called a context.

CONTEXT OBJECTS

A context object is the hardware-recognized data structure that contains information about a particular instance of a procedure. For example, the context object contains the instruction pointer (This points to the current instruction that is being executed. It is roughly equivalent to a program counter on more conventional machines.), the stack pointer, and the list of objects that the procedure can access.

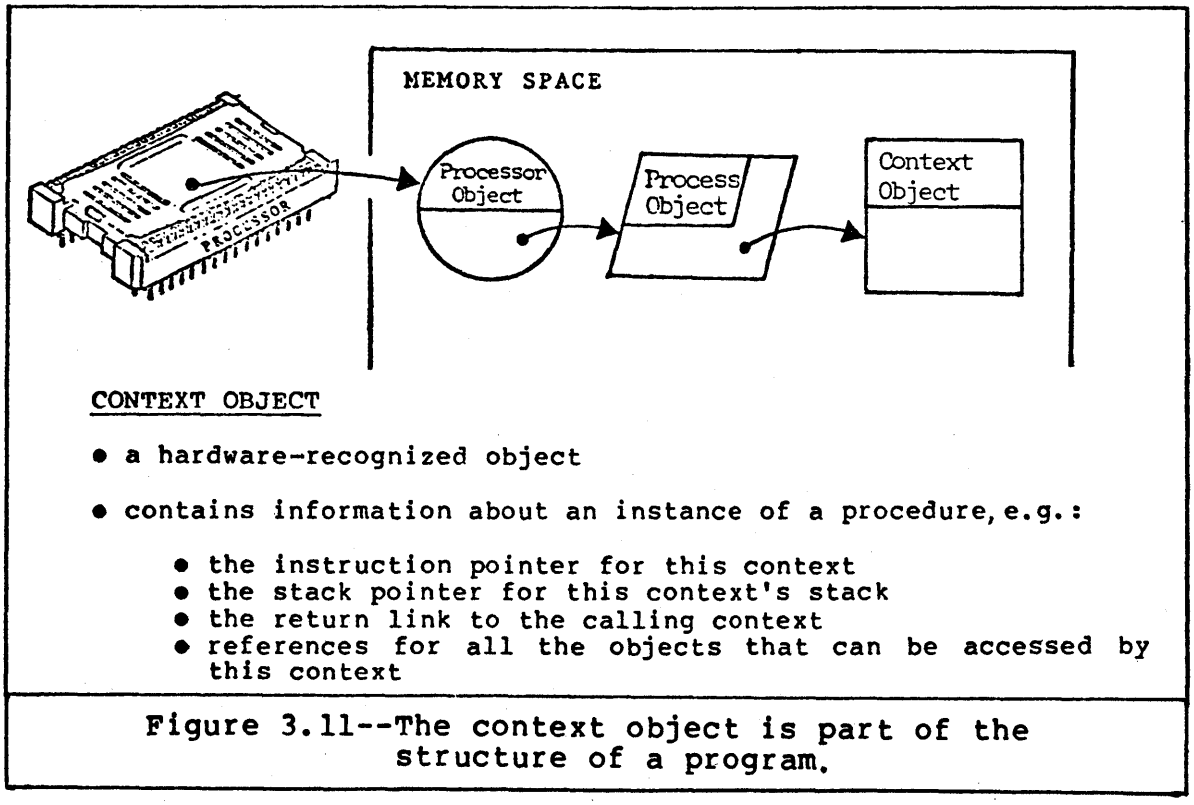


Figure 3.11--The context object is part of the structure of a program.

The last bullet in Figure 3.11 deserves special attention:

- The context object contains a reference for all the objects used by the procedure (see Figure 3.12).

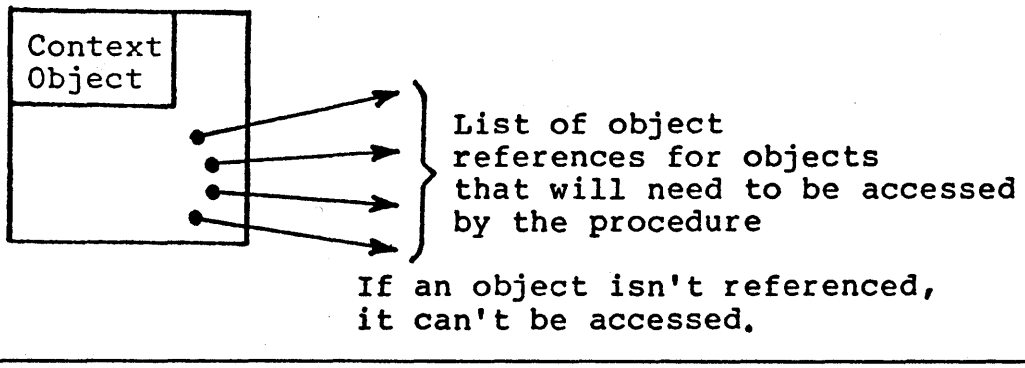


Figure 3.12 -- A context lists the objects accessible by an instance of a procedure.

The complete list of objects accessible by a procedure is called its access environment. Remember, the 432's object-oriented protection mechanism does not allow a program to access an object unless it has a reference for it. Well, all the objects referenced by a context object are part of the context's access environment. Any object listed in the access environment (i.e., that has an object reference in the context object or in one of the objects referenced by the context object) can be accessed; all objects that are not listed cannot be accessed.

When one procedure calls another procedure, the access environment changes because the called procedure has a different context, i.e., a different list of objects which can be accessed (see Figure 3.13).

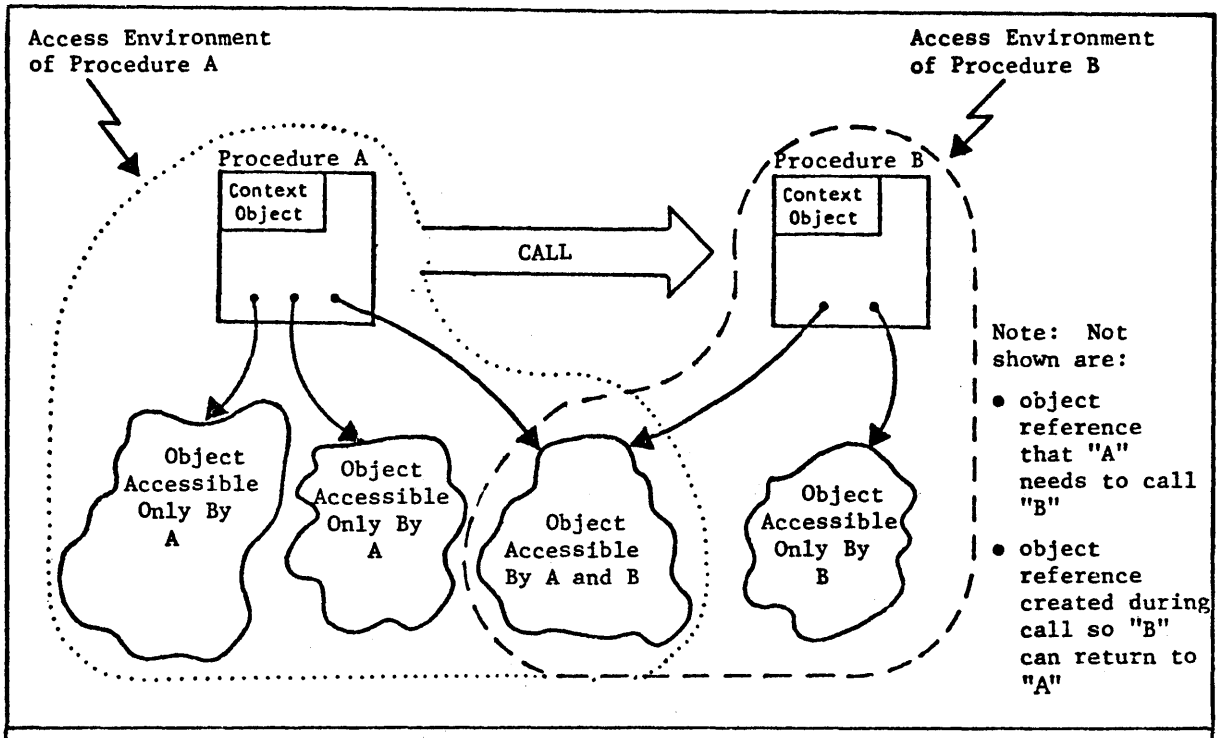


Figure 3.13 -- The access environment changes with each procedure call.

Note that this means that the 432 has a finer "protection granularity" than most machines. The protected access environment is at the procedure level, not at the process or job level as on most machines that have protection. We'll look at the advantages this provides in Chapter 6.

> In summary, then, a context is the root of the access environment for an instance of a procedure. It contains a list of references for all the objects used by this particular instance of the procedure.

Note that in Figure 3.11 one of the things contained by the process object is a reference for the current context object. I say current, because this reference changes dynamically as procedures are called and return.

Here is an example. In Figure 3.14.A, the MAIN procedure is running. Note that the process object has a reference for the context being executed.

In Figure 3.14.B, MAIN has called the procedure SUB using the CALL CONTEXT WITH MESSAGE operator. This single hardware instruction did four things:

- SUB is now the current context being executed, as indicated by the reference from the process object to SUB.
- The access environment was changed. The objects listed in MAIN's context cannot be accessed now (unless they are also listed in SUB's context); only the objects listed in SUB's context can be accessed.
- A return link was created. This is simply an object reference in the called context (SUB) for the calling context (MAIN). It is used to locate MAIN when SUB executes a RETURN instruction.
- A message was passed. There's more about this in a minute.

The MAIN procedure must have a reference for the subroutine SUB in order to call SUB; this reference is not shown in Figure 3.15 -- so as not to confuse it with a reference for the context object that represents a single activation of SUB.

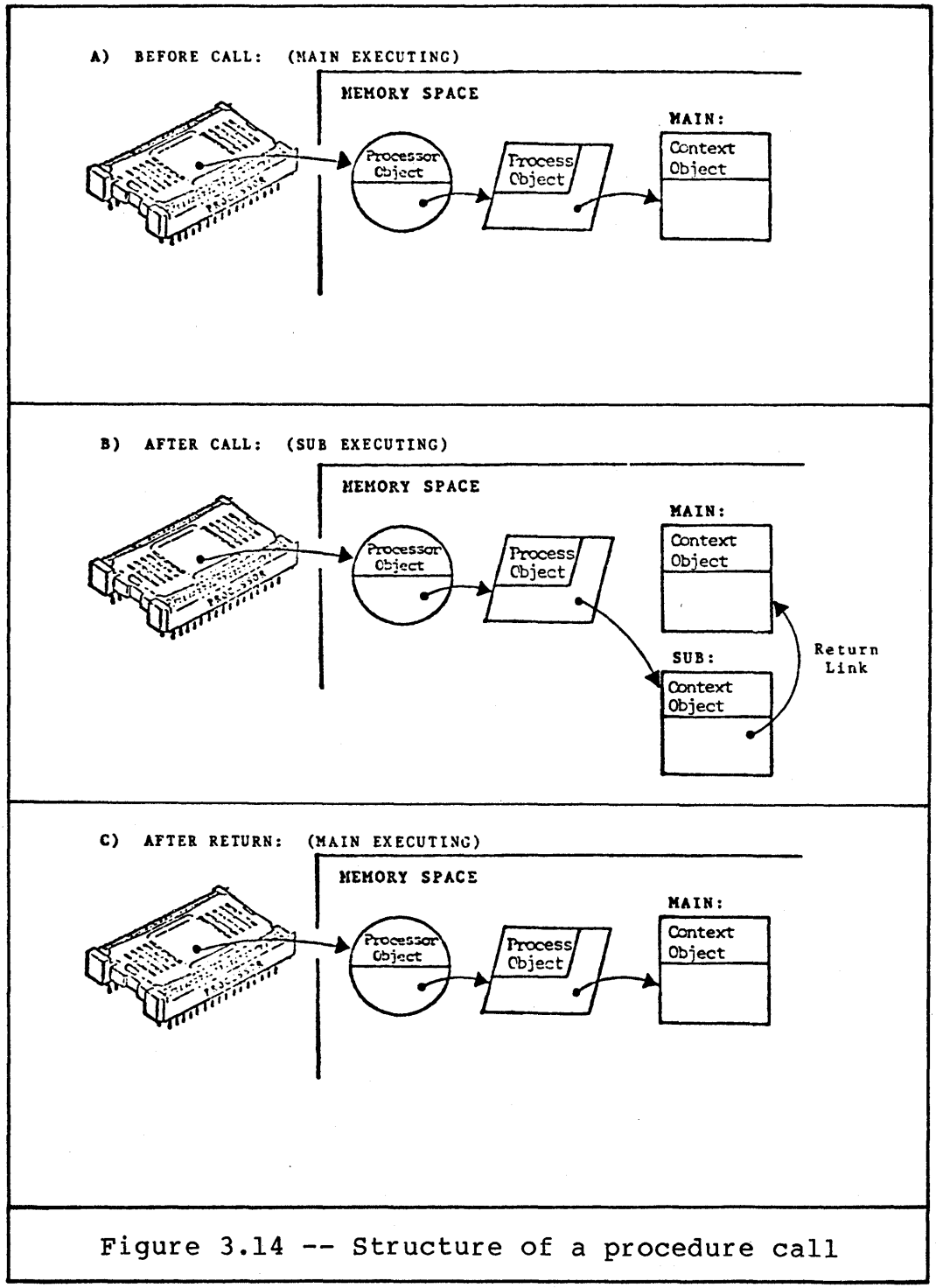


Figure 3.14 -- Structure of a procedure call

In Figure 3.14.C, SUB has returned control to the calling procedure MAIN using the RETURN operator. This single hardware instruction did three things:

- MAIN is now the current context being executed as indicated by the reference from the process object to MAIN.
- The access environment was changed. The objects listed in SUB's context cannot be accessed now (unless they are also listed in MAIN's context); only the objects listed in MAIN's context can be accessed.
- The return link was removed. The object reference to MAIN from SUB no longer exists.

This method of calling procedures and returning has two major differences from most machines:

1. The access environment changes.
2. The message-passing mechanism is very general.

Let's look at these differences in more detail.

The access environment changes. Each procedure has its own access environment which is limited to the set of objects needed by the procedure. This provides much finer control of protection than is found on most other machines. Some of the advantages are discussed in a later chapter.

The message-passing mechanism is very general. The CALL and RETURN operators pass messages by moving the object reference for the message to the context which is to receive the message.

Let's look at the CALL CONTEXT WITH MESSAGE operator as an example of how messages are passed. In Figure 3.15.A, the MAIN context is executing and has prepared a message for SUB. Note the object reference for the message that is in MAIN's context object.

In Figure 3.15.B MAIN has called SUB and has passed it the object reference for the message. Note that the object reference has been removed from MAIN's context and placed in SUB's context.

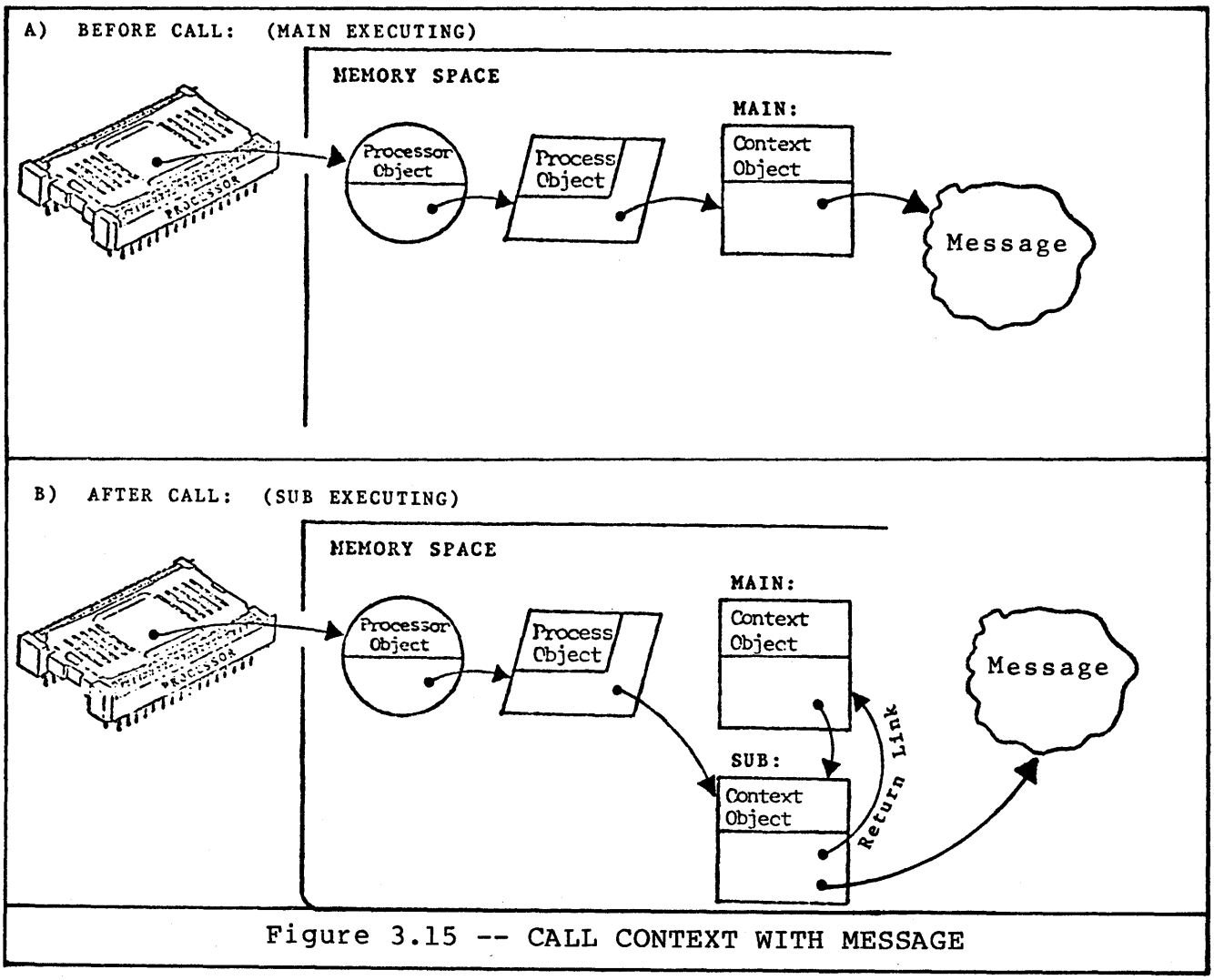


Figure 3.15 -- CALL CONTEXT WITH MESSAGE

This is a powerful parameter-passing mechanism because it is general. Any object can be a message. For example, if the called procedure is sine of x, the message is simple -- it contains the value x. But, what if the called procedure is the part of the operating system that schedules processes? The message might be a process object that needs to be scheduled.

In summary, the message-passing mechanism is powerful because it is general. It can pass a reference for a simple object that contains the parameter values, or it can pass a reference for an object that contains many object references (see Figure 3.16). Messages are discussed in more detail in Chapter 4.

One final note. This discussion is somewhat simplified, because context objects are actually allocated and deallocated by the CALL and RETURN instructions. I have not explained this because storage allocation and deallocation are beyond the scope of this book. If you are interested in this subject, please refer to the Architecture Reference Manual.

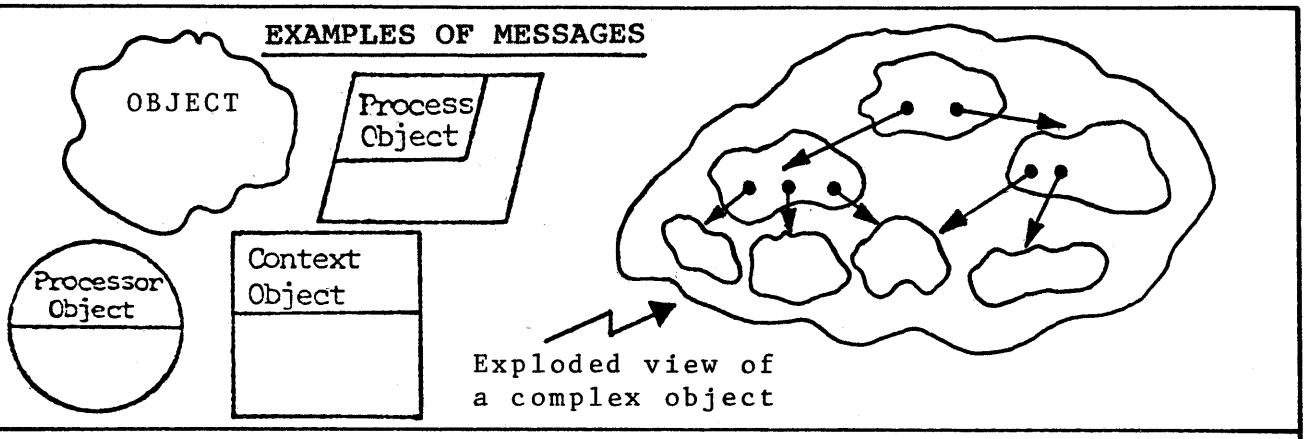


Figure 3.16 -- Any object can be a message.

INSTRUCTION AND DATA OBJECTS

You are nearing the end of your journey through the structure of a program. We've talked about physical processors, processor objects, process objects, and context objects. Now its time to talk about the last two objects which are part of the basic structure of a program.

Every computer does two very basic things:

1. It fetches and executes instructions.

2. It manipulates data.

Therefore, a context (an instance of a procedure) needs to be able to access objects that contain instructions and data.

An instruction object is exactly what its name implies -- an object that holds instructions. Pretty simple, but two things make it interesting:

1. Instruction objects only hold instructions (and a little bit of system information) -- they don't hold data.

2. Instruction objects are the only type of objects that a processor will use as a source of instructions to be fetched and executed.

This second feature is nice because it means that other kinds of objects such as data objects, process objects, etc. cannot be accidentally mistaken for instructions and executed. This is an example of how the 432 uses type checking to produce more reliable systems.

Having instructions isolated in their own object has another advantage. All 432 programs are fully reentrant. Instruction objects can be shared by several instances of a program.

A data object is exactly what its name implies -- an object that holds data, e.g., integers, reals, characters, tables, etc.

"Data object", then, is a very general term. In fact, it is so general that the 432 provides a mechanism for giving data objects (as well as other objects) a software-defined type such as "telephone directory" that is more specific than "data object". This mechanism is described in Chapter 6, Designing Software Systems.

Figure 3.17 shows the complete structure of a program. Note that the context object has references for all the instruction and data objects that will be executed/manipulated by this instance of the procedure (i.e. this context).

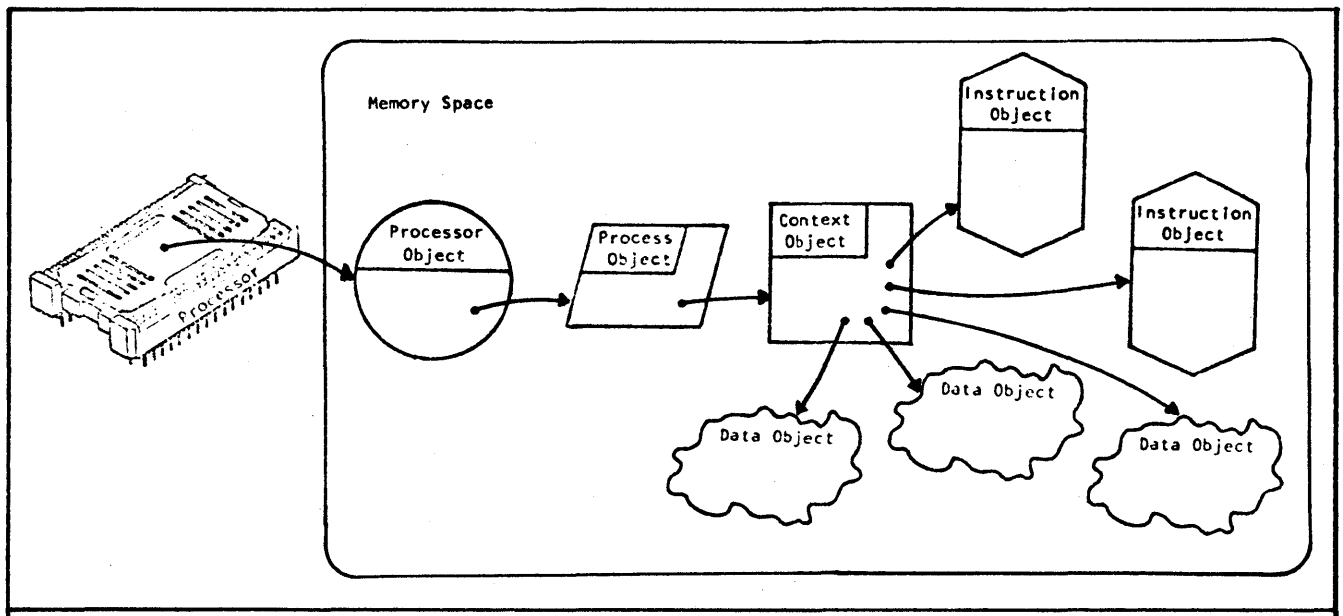
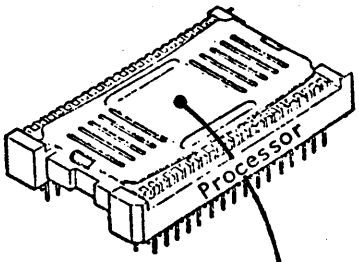


Figure 3.17 -- The structure of a program

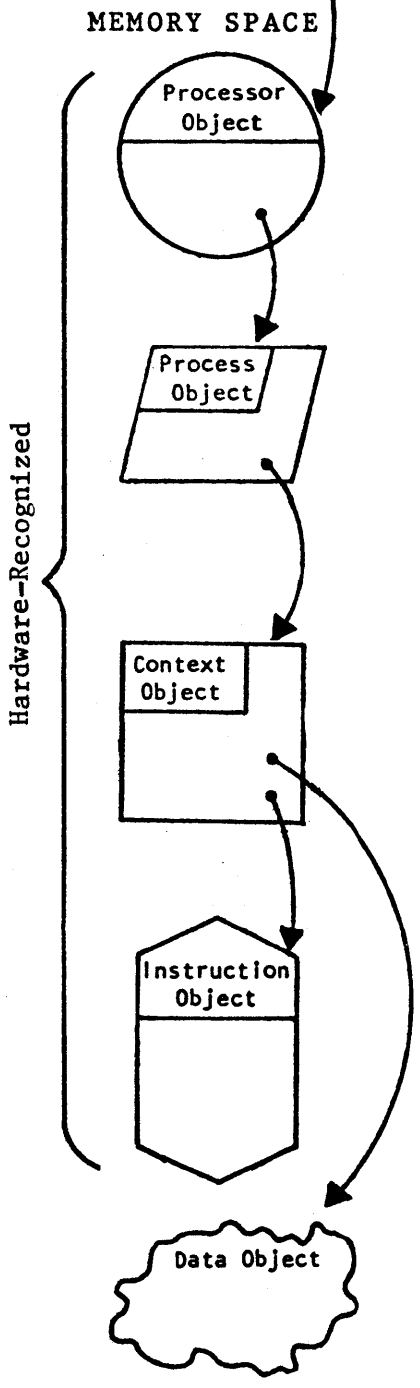
When looking at Figure 3.17, remember that this is not a static structure -- it changes dynamically as the program runs different processes, as processes execute different contexts, and as contexts create and destroy data objects.

SUMMARY

You've now had a complete overview of program structure, from processors to data objects. Figure 3.18 summarizes the important points about each object.



PHYSICAL PROCESSOR
- Made of silicon (not an object)
- Fetches and executes instructions



PROCESSOR OBJECT
- Each processor has one
- Contains information like:
  - Processor status (e.g., running or halted)
  - Diagnostic & machine check information
  - A reference for the process currently being executed

PROCESS OBJECT
- One for each process in the system
- Contains information like:
  - Process status (e.g., running, waiting, etc.)
  - How the process should be scheduled
  - A reference for the context currently being executed

CONTEXT OBJECT
- One for each instance of a procedure
- Contains information like:
  - Instruction pointer for this context
  - Stack pointer for this context's stack
  - Return link to the calling context
  - References for all the objects that can be accessed by this context

INSTRUCTION OBJECT
- Contains only instructions -- does not contain data
- Is the only type of object that a processor will use as a source of instructions to be fetched and executed

DATA OBJECT
- Contains data (e.g., integers, reals, characters, or a combination of several primitive data types)

Figure 3.18 -- The objects that make up a program

ADVANTAGES AND BENEFITS

The major advantages of building these structures into the architecture are that:

- Frequently-used operations can be performed by the hardware, and
- a major portion of the system design is already complete.

This means that the 432 architecture has the following benefits:

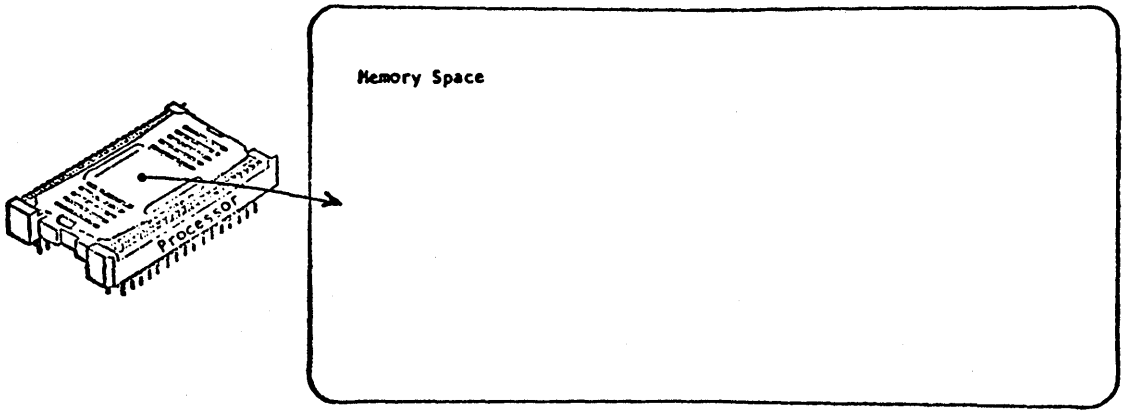
- Execution of frequently-used operating system and high-level language operations is faster.
- The operating system is more secure because support for it is built into the hardware.
- System design is faster because more of it is already done.

PROGRAM STRUCTURE QUIZ

1. These objects are part of a 432 program:

   - Context Object
   - Instruction Object
   - Process Object
   - Processor Object
   - Data Object

   Finish the picture below to show how these objects fit together to form the structure of a 432 program.



2. In one sentence, what is a processor object? Give 3 examples of the kind of information that is found in a processor object.

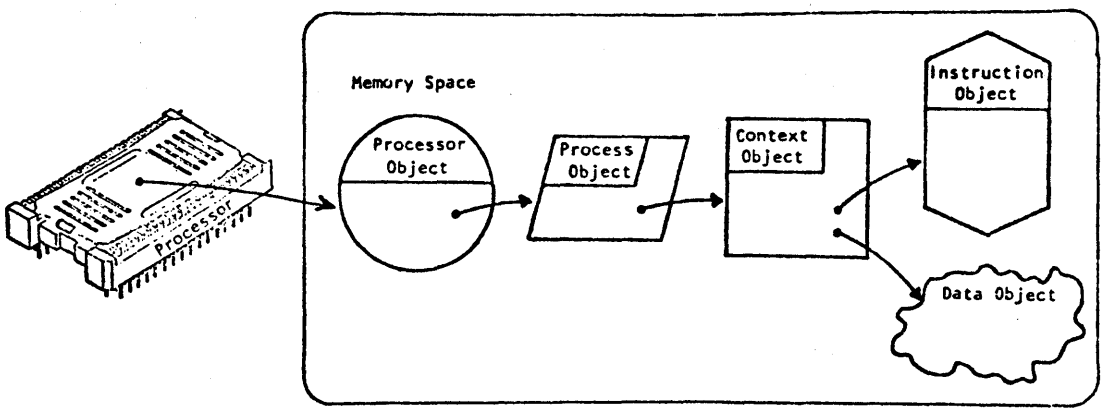
3. In one sentence, what is a process? Give 3 examples of the kind of information that is found in a process object.

KEY TO PROGRAM STRUCTURE QUIZ

1. These objects are part of a 432 program:

   - Context Object
   - Instruction Object
   - Process Object
   - Processor Object
   - Data Object

   Finish the picture below to show how these objects fit together to form the structure of a 432 program.



2. In one sentence, what is a processor object? Give 3 examples of the kind of information that is found in a processor object.

   A processor object is a data structure that contains information about a specific processor. It contains information like:

   - processor status (e.g., running or halted)
   - diagnostic and machine check information
   - an object reference for the process currently being executed

3. In one sentence, what is a process? Give 3 examples of the kind of information that is found in a process object.

   A process is a unit of work for a processor, i.e., the smallest unit of programming activity that can be scheduled to run on a processor.

   A process object is a data structure that contains information about a specific process. It contains information like:

   - process status (running, waiting, etc.)
   - scheduling parameters that describe how the process should be scheduled to run on a processor
   - a reference for the context currently being executed

Chapter 4

INTERPROCESS COMMUNICATION

Introduction

Now that you are familiar with the basic structure of a 432 program, it is time to consider the support that the 432 provides for concurrent programming.

This chapter discusses programs with more than one process, i.e., two or more parts that can execute at the same time. Special support is required for synchronization and communication between concurrent processes, and the 432 provides this support in the hardware.

The major topics covered by this chapter are:

• PROGRAMS WITH MORE THAN ONE PROCESS	Sometimes a program has 2 or more parts that can execute at the same time. This section considers an example of such a program and looks at the support required.
• COMMUNICATION PORTS	Communication ports are the 432 hardware-defined objects that support interprocess communication and synchronization. This section gives an overview of how they work and defines the term message.
• SEND AND RECEIVE	This section explains how the SEND and RECEIVE operators allow processes to communicate by sending and receiving messages.
• AN EXAMPLE OF INTERPROCESS COMMUNICATION	This section walks through an example of how parallel processes communicate.

PROGRAMS WITH MORE THAN ONE PROCESS

A process is a unit of work for a processor -- it is the unit of software that can be scheduled to run on a processor. A program is a collection of instructions and data that runs on a processor and tells it what to do to provide some service, such as computing a sine, firing a missile, or managing an airline reservation system. A program can be a process, part of a process, or several processes.

The important difference between the two terms is that the term program is used when we are describing a unit of software that provides some service, and the term process is used when we are talking about a unit of software that is schedulable. A process is always a program, i.e., it always provides a service, but a program is not always a process. A process is always schedulable; a program may be schedulable, may not be schedulable by itself, or may have several parts (processes) each of which is schedulable.

Here is an example to clarify the distinction between the two terms. The example involves the totally automated "office of the future" where all the people have been replaced by computer programs. REPORTER is a program that continuously produces reports describing the accomplishments of this automated office.

The REPORTER program is made up of 2 smaller programs called ROGER and PATTY. The first program, ROGER, is a writer program which writes the reports and then gives them to the second program, PATTY. PATTY is a secretarial program which types the reports and then gives them back to ROGER for proofreading. When a report is correct, Roger can print it (not shown).

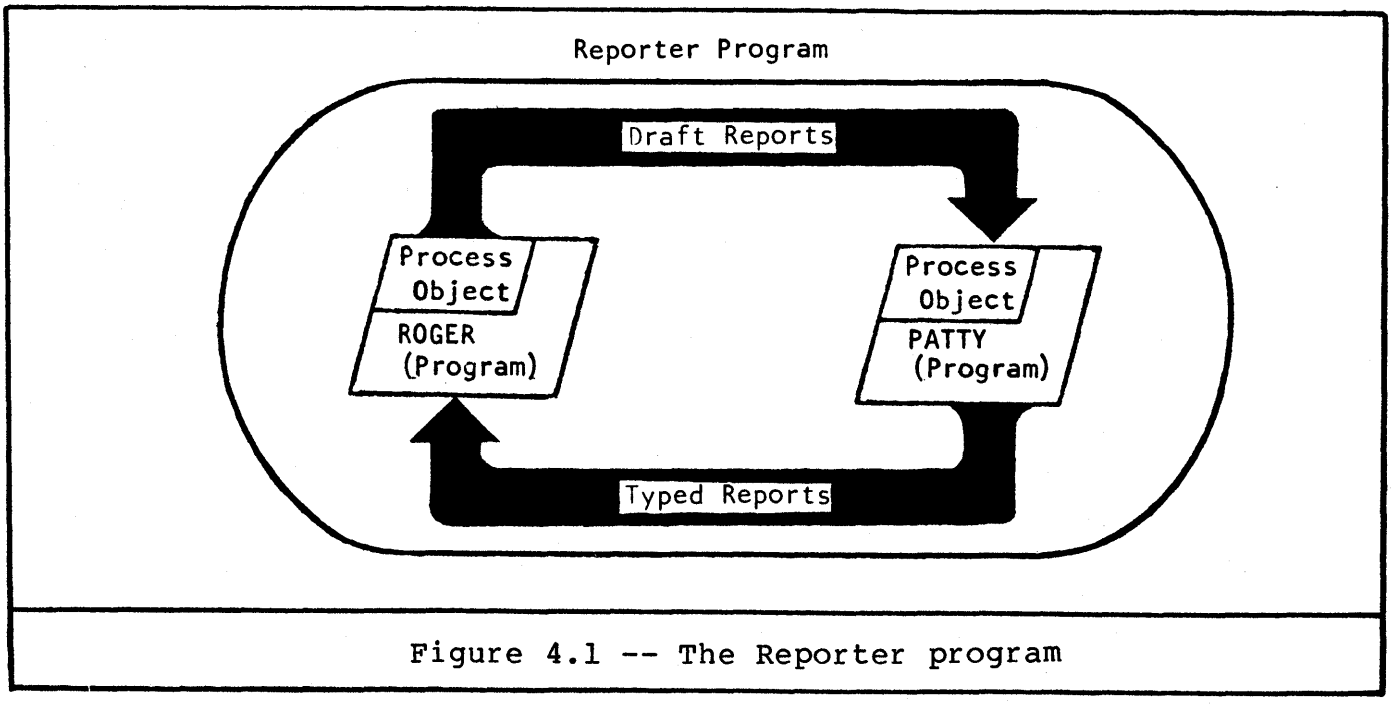


Figure 4.1 -- The Reporter program

Figure 4.1 shows this office of the future. Note that it shows three programs: REPORTER, ROGER, and PATTY.

ROGER and PATTY, in addition to being programs, are also processes, i.e., they can be independently scheduled for execution on a processor. If there are two processors in our system, both ROGER and PATTY can be executing at the same time on different processors. PATTY can be typing ROGER's first report at the same time that ROGER is writing a second report.

When a program is made up of more than one process (like the REPORTER program), a mechanism is needed for the separate processes to communicate with each other. In the REPORTER program, ROGER needs to communicate with PATTY to give her draft reports to type, and PATTY needs to communicate with ROGER to give him typed reports to proofread.

COMMUNICATION PORTS

432 processes communicate with each other by using communication port objects.

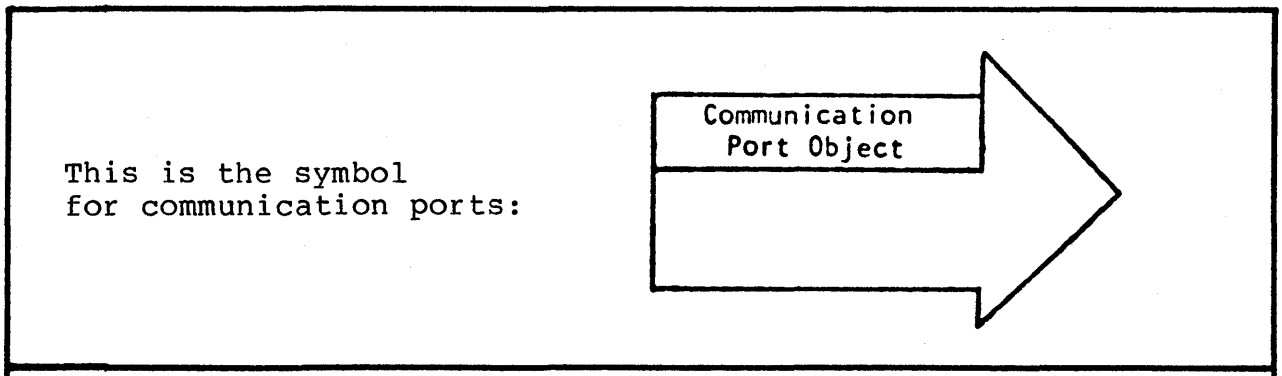


Figure 4.2 -- A communication port

For example, communication ports can be used to implement the REPORTER program.

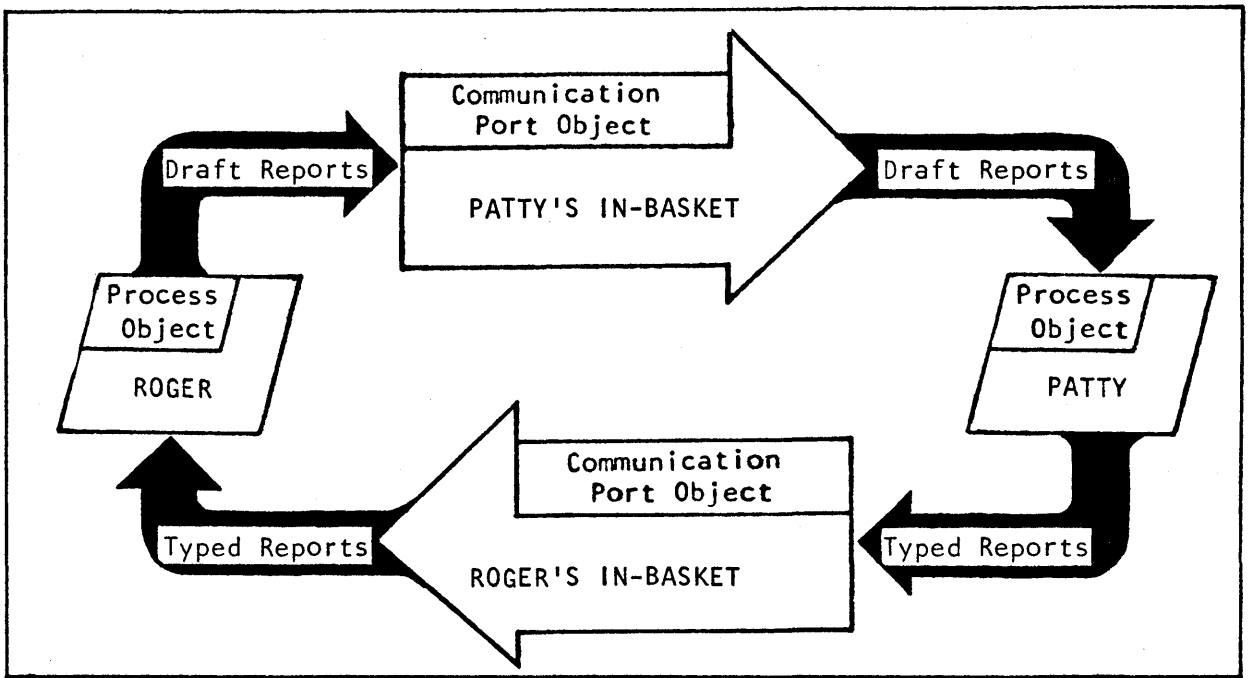


Figure 4.3 -- The Reporter program

In Figure 4.3, communication ports are used as "in-baskets" for ROGER and PATTY. Whenever ROGER finishes writing a report, he puts it in PATTY's in-basket. Reports accumulate in PATTY's in-basket (in a first-in-first-out (FIFO) queue) until she is ready to type one. Whenever PATTY needs work, she removes a report from her in-basket, types it, and places the typed report in ROGER's in-basket. When ROGER is ready to proofread a report, he removes a typed report from the FIFO queue of reports in his in-basket. (The 432 also supports other (non-FIFO) queuing disciplines, but they are not covered in this primer).

The communication port acts as a buffer between the two asynchronous processes, allowing each to proceed at its own pace. ROGER does not need to wait until PATTY finishes typing Report 1 in order to place Report 2 in her in-basket. Similarily, PATTY does not need to worry about what ROGER is doing. She just keeps typing a report until she finishes. When she is done, she puts it in ROGER's in-basket and takes another draft from her in-basket.

Communication ports are a very powerful and flexible mechanism for interprocess communication because they can be used to send just about any kind of message between two processes. In the REPORTER program, the draft reports and the typed reports are objects that are sent as messages. But, communication ports allow any object to be a message. This includes all hardware-defined objects (e.g., process objects and context objects) as well as all software-defined objects (e.g., reports and telephone books).

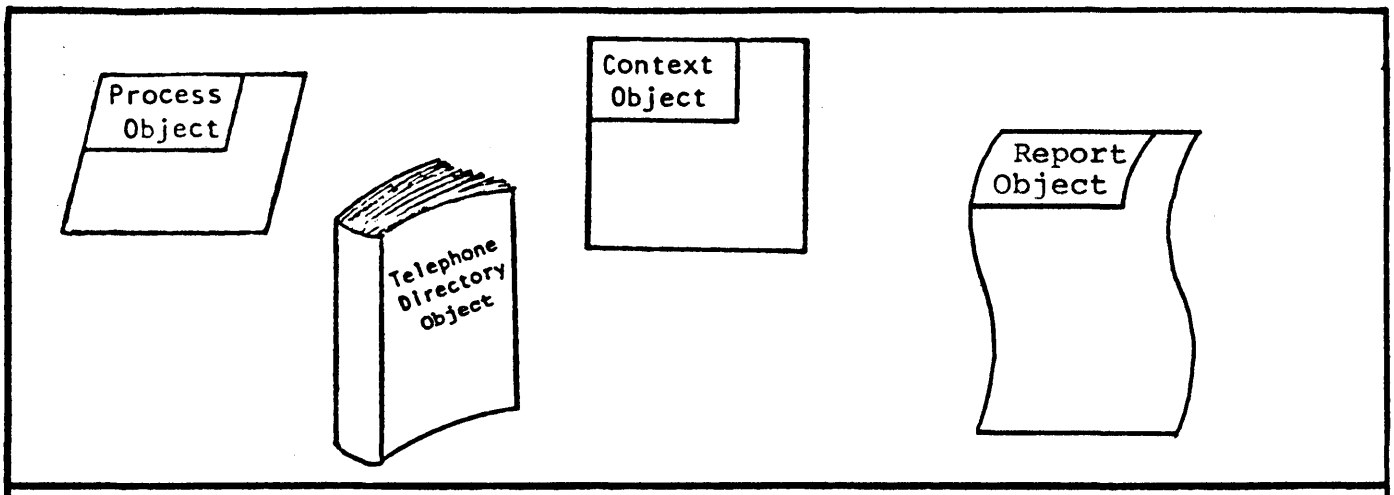


Figure 4.4 -- Any object can be a message.

There are two kinds of hardware operators that manipulate messages and communication ports -- SEND and RECEIVE. The next two sections take closer looks at these operators and at examples of how they are used.

SEND and RECEIVE

When ROGER wants to give a message to PATTY, he sends it to her by putting it in her in-basket. Later, when PATTY is ready to receive the message, she removes it from her in-basket. The 432 hardware operation that places the message in the in-basket is called SEND. Figure 4.5 gives a conceptual explanation of what the instruction does.

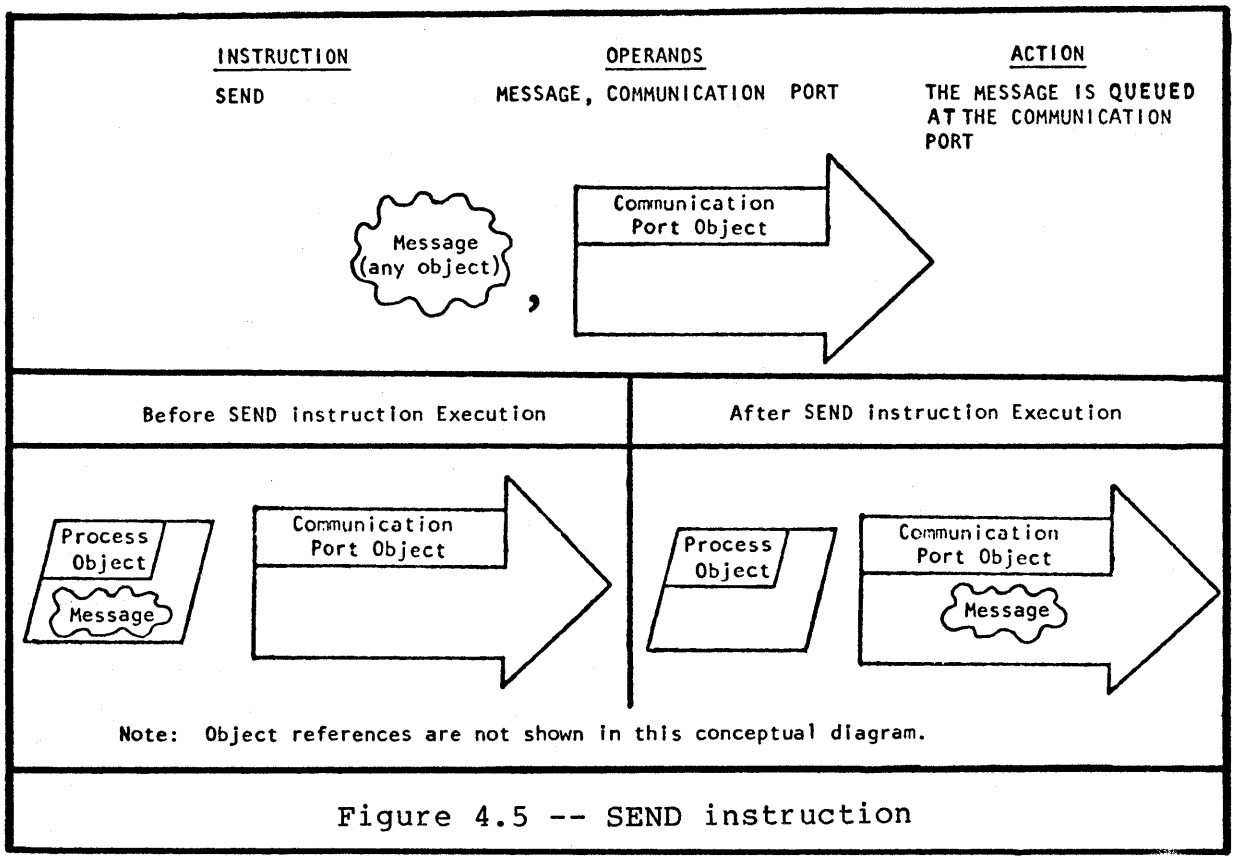


Figure 4.5 -- SEND instruction

The SEND instruction has 2 operands: the message being sent and the communication port it is being sent to. Figure 4.5 summarizes the action of the instruction. Before SEND is executed, the process has a message it wants to send. After SEND is executed, the message has been sent to the specified communication port (the message is not actually copied; only an object reference is copied).

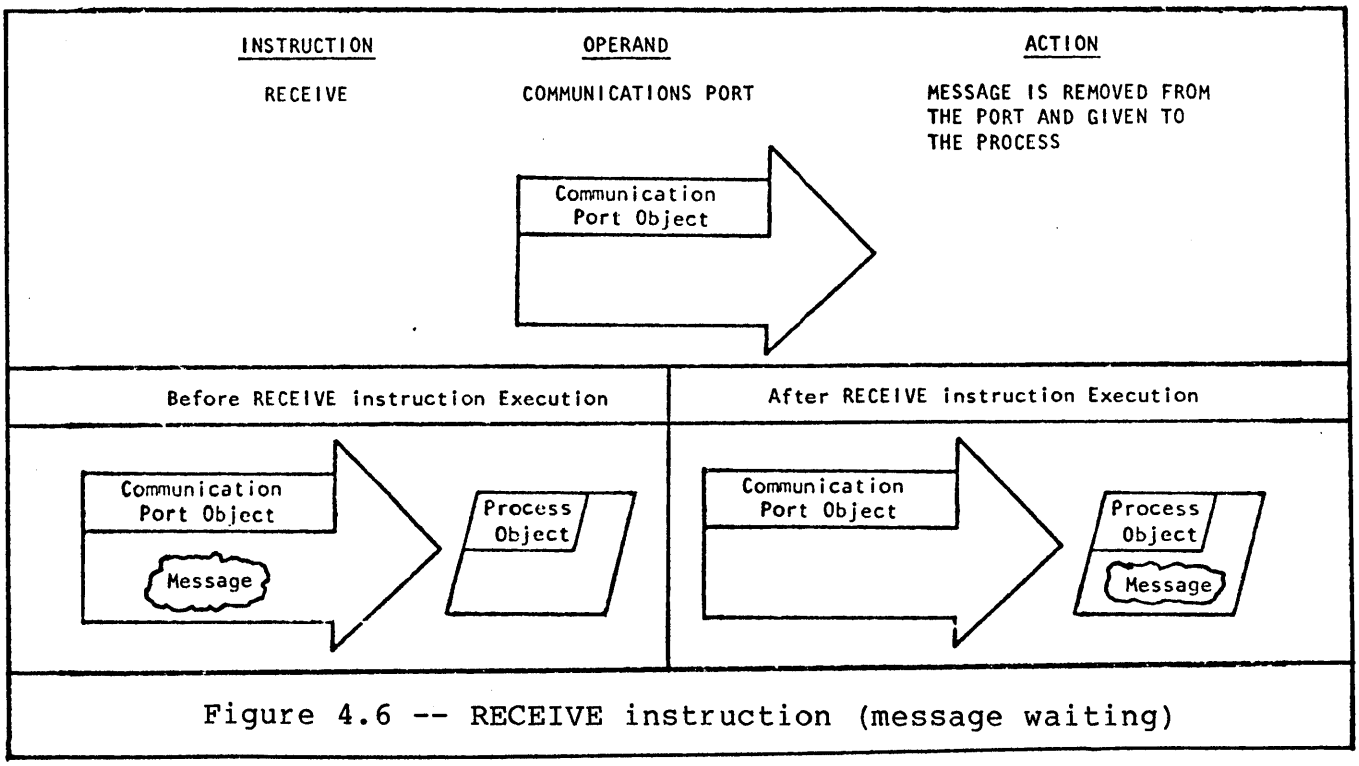


Figure 4.6 -- RECEIVE instruction (message waiting)

The RECEIVE instruction has one operand: the communication port where the message is to be received. Figure 4.6 summarizes the action of the instruction if there is a message waiting. Before RECEIVE is executed, there is a message waiting in the communication port (placed there earlier by a SEND instruction). After RECEIVE is executed, the message is moved from the port to the process executing the RECEIVE.

The RECEIVE instruction is straightforward if there is a message waiting in the communication port -- the message simply moves from the port to the process executing the RECEIVE instruction. What happens if there are no messages waiting in the communication port?

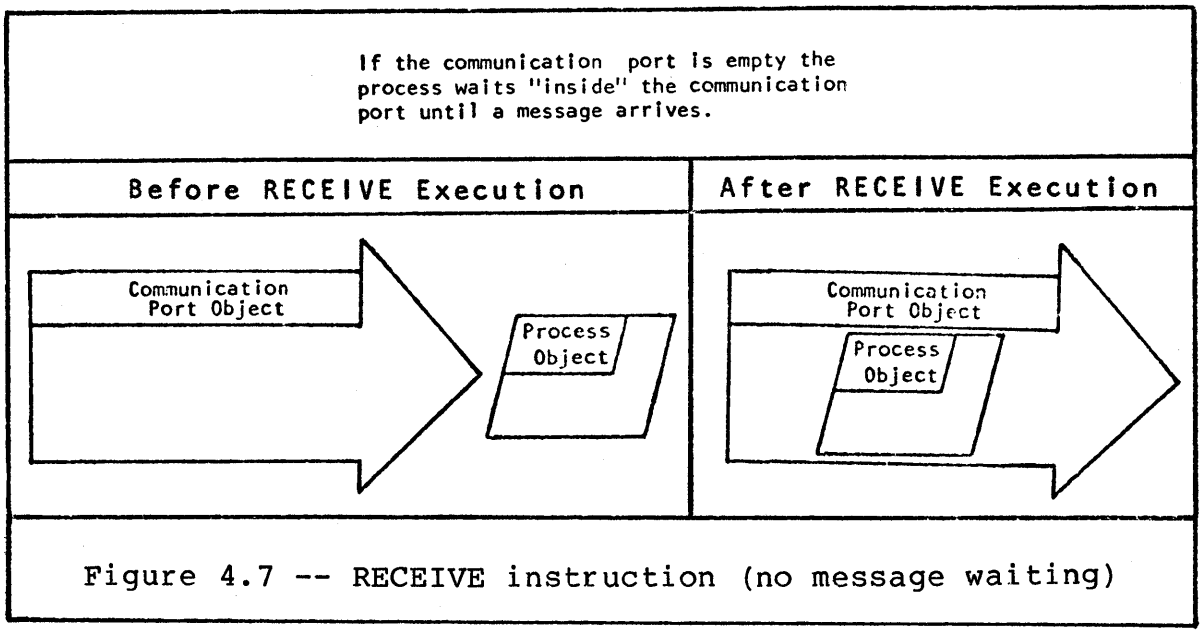


Figure 4.7 -- RECEIVE instruction (no message waiting)

If a process executes a RECEIVE instruction and the communication port is empty, the process waits "inside" the communication port until a message is sent. Remember that messages are objects and that a process is just a special type of object. Therefore, processes can queue up waiting for messages just like messages can queue up waiting to be received by processes.

Execution of instructions from the waiting process stops until a message is sent to the port. When a message is sent to a port that has a waiting process, the message is immediately given to the waiting process so that it can begin execution again.

Note that while a process waits at a communication port the processor that is executing the process may not. It can immediately begin executing some other process that is ready to execute. How processors handle this scheduling and dispatching of processes is covered in Chapter 6.

Actually, RECEIVE is only one of several receive instructions. It is useful when you want a process to stop and wait when there are no messages. But, sometimes it is useful for a process to receive a message if there is one, but to continue doing something else if there is not. The CONDITIONAL RECEIVE instruction allows it to do just that.

The SEND instruction may also require the executing process to wait "inside" the communication port -- if the part of the port that holds messages (the port message buffer) is "full" and cannot accept any more messages when the SEND is issued. When this happens, the SENDing process waits until some other process executes a RECEIVE on the port, removing a message and freeing a slot in the message buffer. The SEND is then completed and the SENDing process no longer waits at the port. Just as there are variants of RECEIVE which do not wait if the port is empty, there are variants of SEND (e.g., CONDITIONAL SEND), that do not wait if the port is full.

The next section returns to the REPORTER program and looks at an example showing how communication ports and the SEND and RECEIVE operators are used.

AN EXAMPLE OF INTERPROCESS COMMUNICATION

The example program has much the same structure as the REPORTER program discussed earlier, but with one slight difference: a second secretarial process named TOM. TOM does the same work as PATTY and even uses the same instruction objects as PATTY. The only difference is that there are now 3 processes (ROGER, PATTY, and TOM) that can execute at the same time instead of just 2 (ROGER and PATTY). ROGER does not care who types his reports; both PATTY and TOM are equally competent.

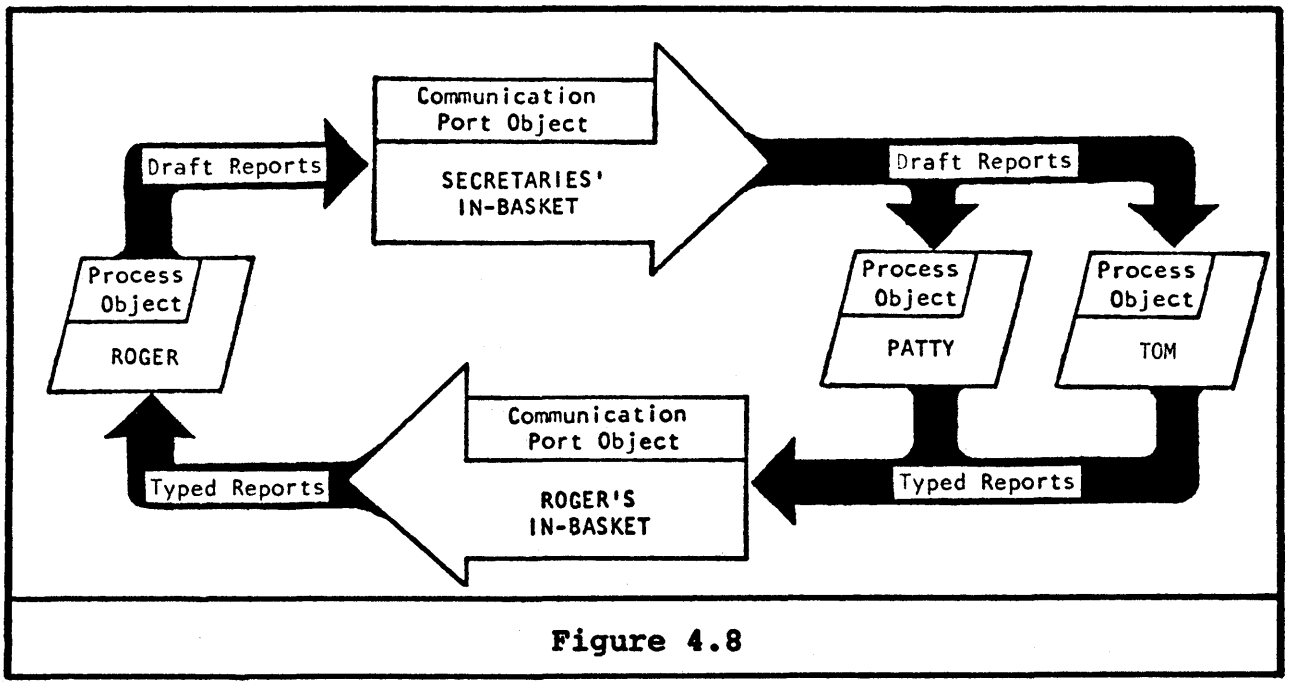


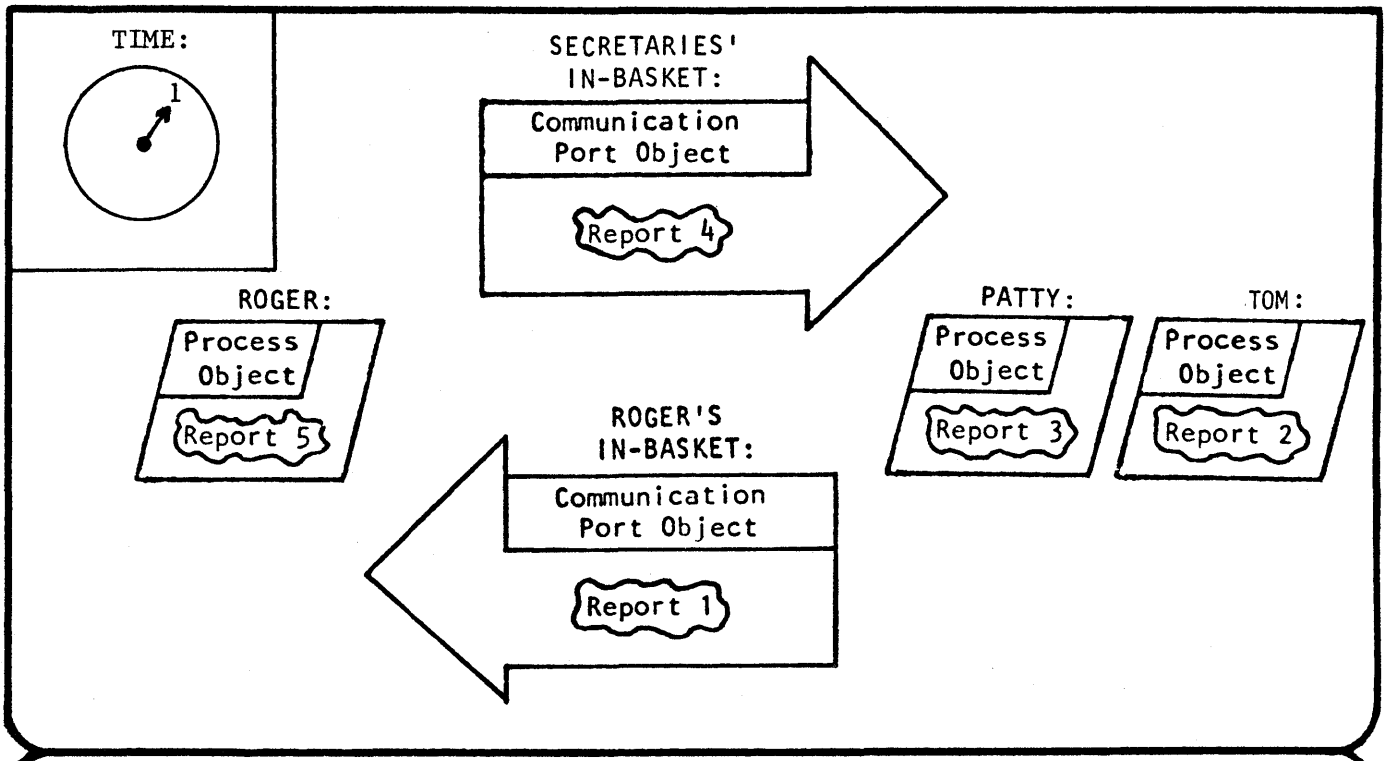
Figure 4.8

In Figure 4.8, you will note that both PATTY and TOM share one secretarial in-basket. Whenever ROGER wants a report typed, he places it in the secretarial in-basket and either PATTY or TOM removes it, types it, and places it in ROGER's in-basket.

In reading the example, please remember two things: First, the diagrams show objects being moved around. Actually, only references for the objects are moved; the objects themselves stay in the same locations.

Second, unlike most of the diagrams in earlier chapters, the diagrams in this example do not show object references. The object references are described later, but are left out of this example for simplicity.

Figure 4.9.1 is the first "snapshot" of what is happening as the modified REPORTER program executes. Note that the program has been executing for some time, because ROGER has already written 4 reports and is now writing Report 5.



Three processes are executing concurrently:

1. ROGER writing Report 5
2. PATTY typing Report 3
3. TOM typing Report 2

Two reports are in communication ports:

1. Report 4 has been written.
2. Report 1 has been typed.

Figure 4.9.1

At Time 1 (Figure 4.9.1), there are three processes running in parallel. ROGER is writing Report 5, PATTY is typing Report 3, and TOM is typing Report 2. Report 4 has been written by ROGER and is ready to be typed by either PATTY or TOM. Report 1 has been written by ROGER and typed by either PATTY or TOM; it is now ready for ROGER to proofread.

At Time 2 (Figure 4.9.2), TOM has finished typing Report 2, so he executes a SEND instruction to send the report to the communication port which is being used as ROGER's in-basket. This instruction moves the report from TOM's process to ROGER's in-basket.

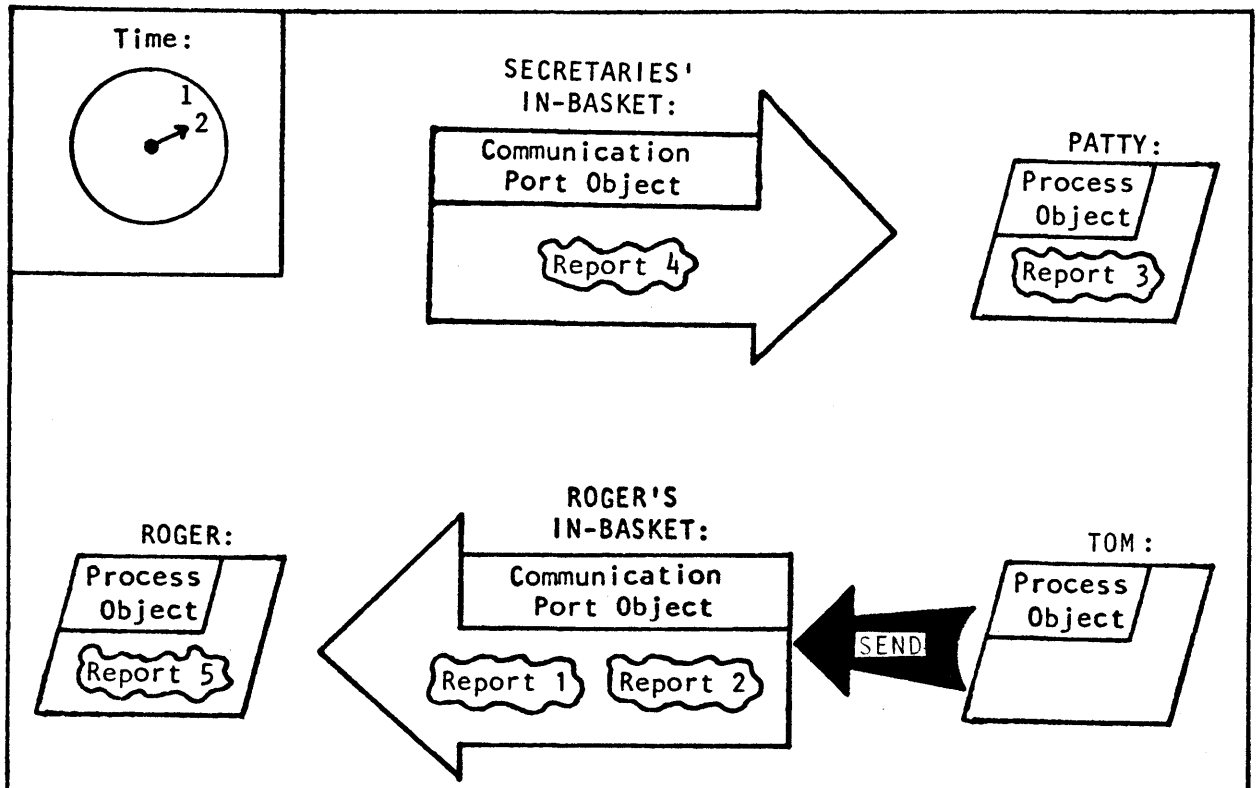


Figure 4.9.2 - TOM finishes Report 2 and sends it to ROGER's in-basket.

Now that TOM has finished with Report 2, he is ready to type another report. In Figure 4.9.3, TOM executes a RECEIVE instruction and receives a message (Report 4) from the secretaries' in-basket. TOM can now start typing Report 4.

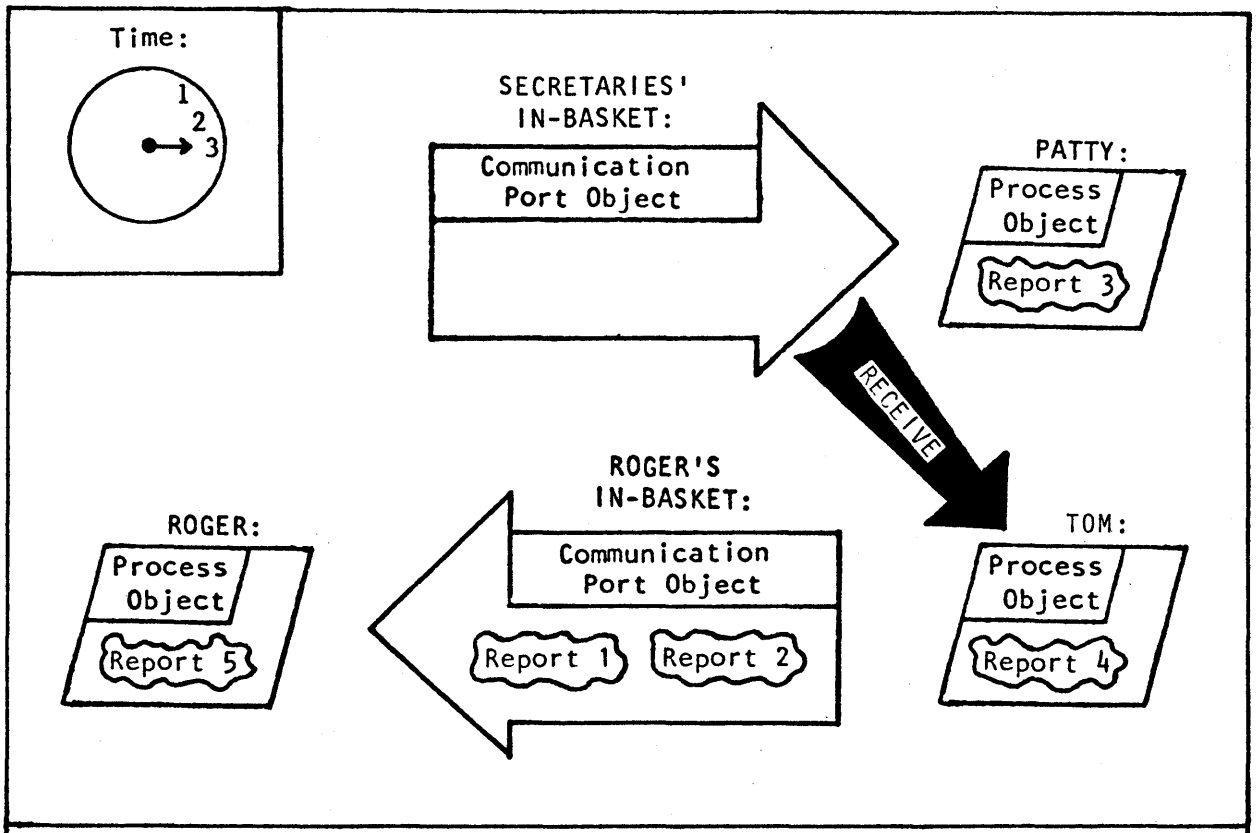


Figure 4.9.3 -- TOM looks for more work in the secretaries' in-basket and receives Report 4.

At Time 4 (Figure 4.9.4), TOM is typing Report 4, ROGER is writing Report 5, and PATTY has finished typing Report 3. PATTY executes a SEND instruction and sends Report 3 to ROGER's in-basket.

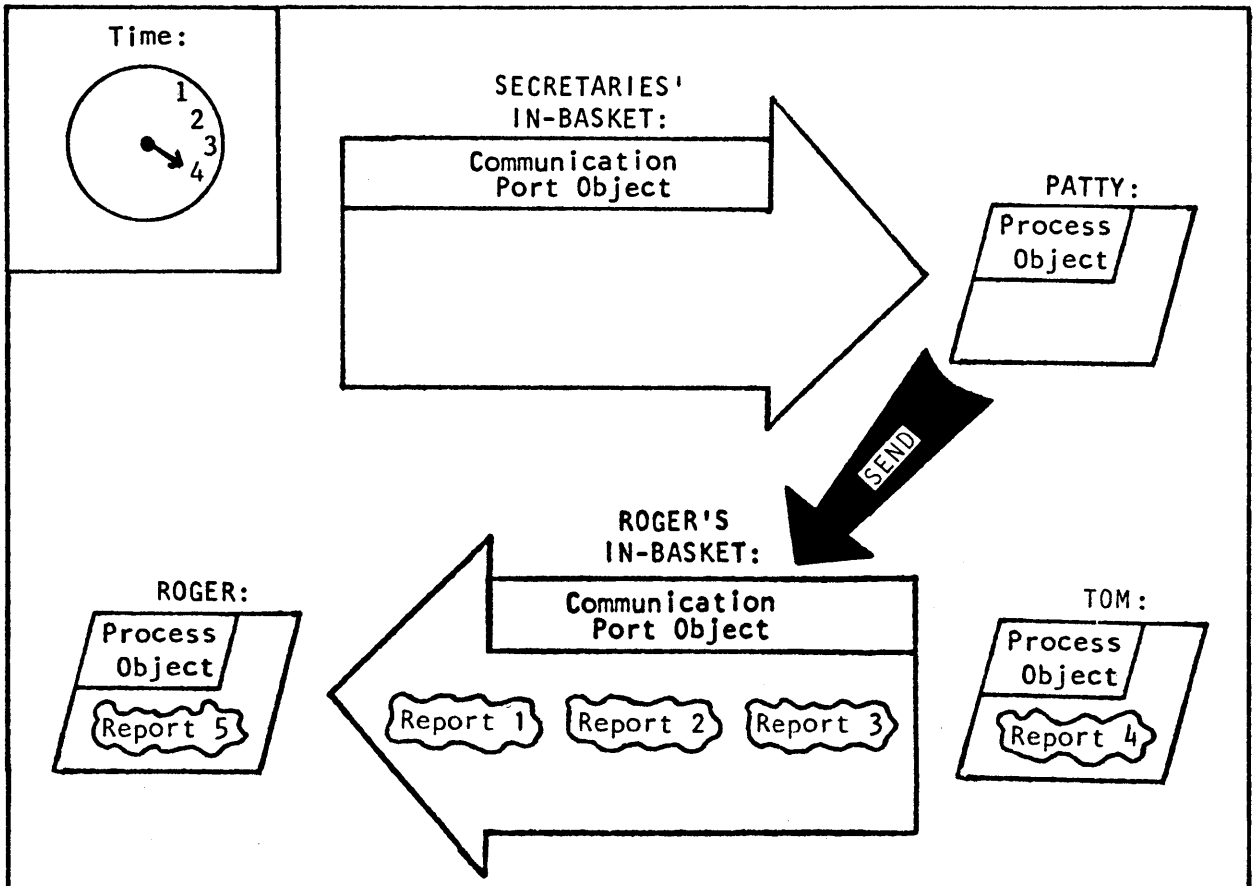


Figure 4.9.4 -- PATTY finishes Report 3 and sends it to ROGER's in-basket.

There are now 3 reports queued in ROGER's in-basket. When ROGER is ready to proofread, he will execute a RECEIVE instruction to remove a report from his in-basket. He will receive reports one at a time (one every time he executes a RECEIVE) in the same order that they were sent to the port.

Now that PATTY has finished typing Report 3 and has sent it to ROGER's in-basket, she is ready to type another report. At Time 5 (Figure 4.9.5), PATTY executes a RECEIVE instruction to get another report from the secretaries' in-basket. But this time the communication port is empty. Because there is nothing for PATTY to receive, she "climbs inside" the secretaries' in-basket to wait for a message. Remember that processes are objects and can be queued up to wait for messages, just as messages are objects and can be queued up to wait for processes to receive them.

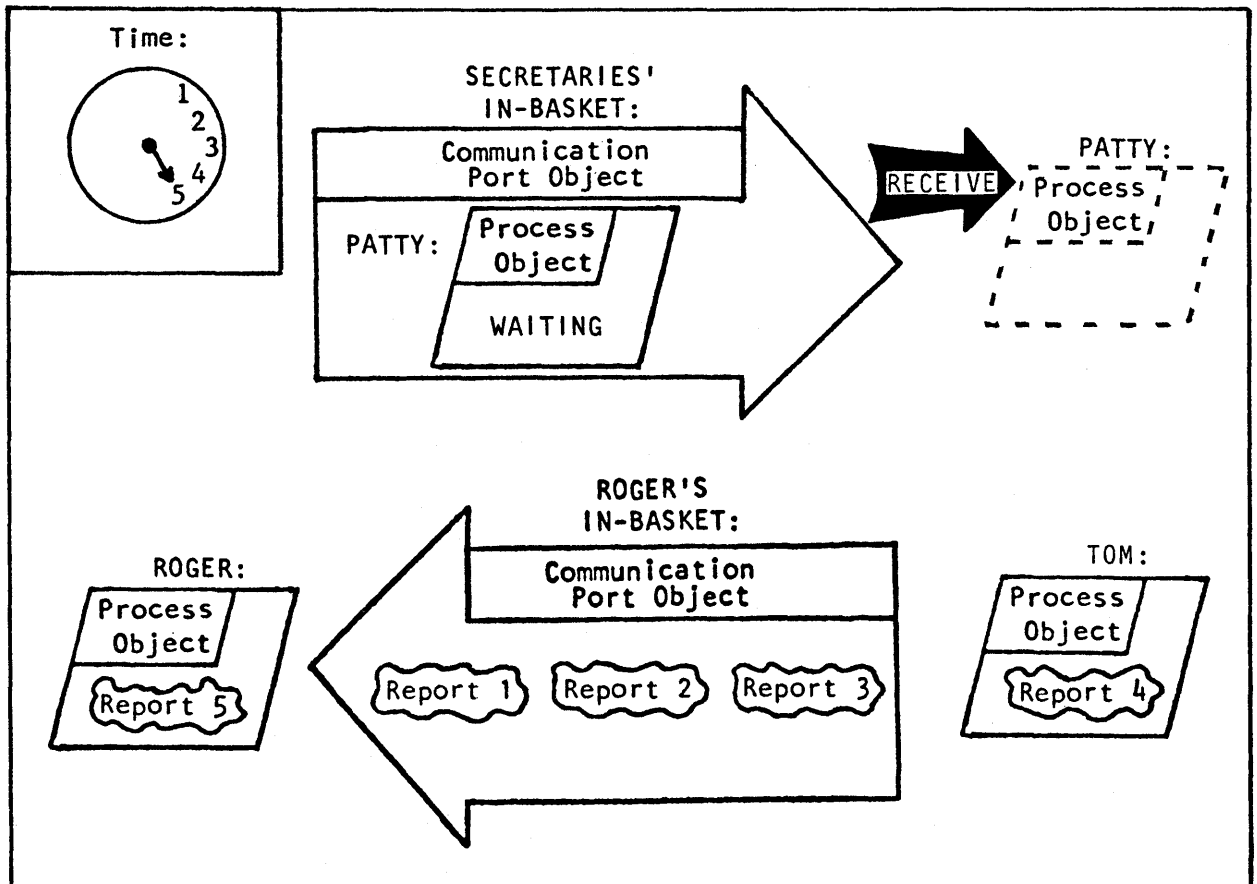
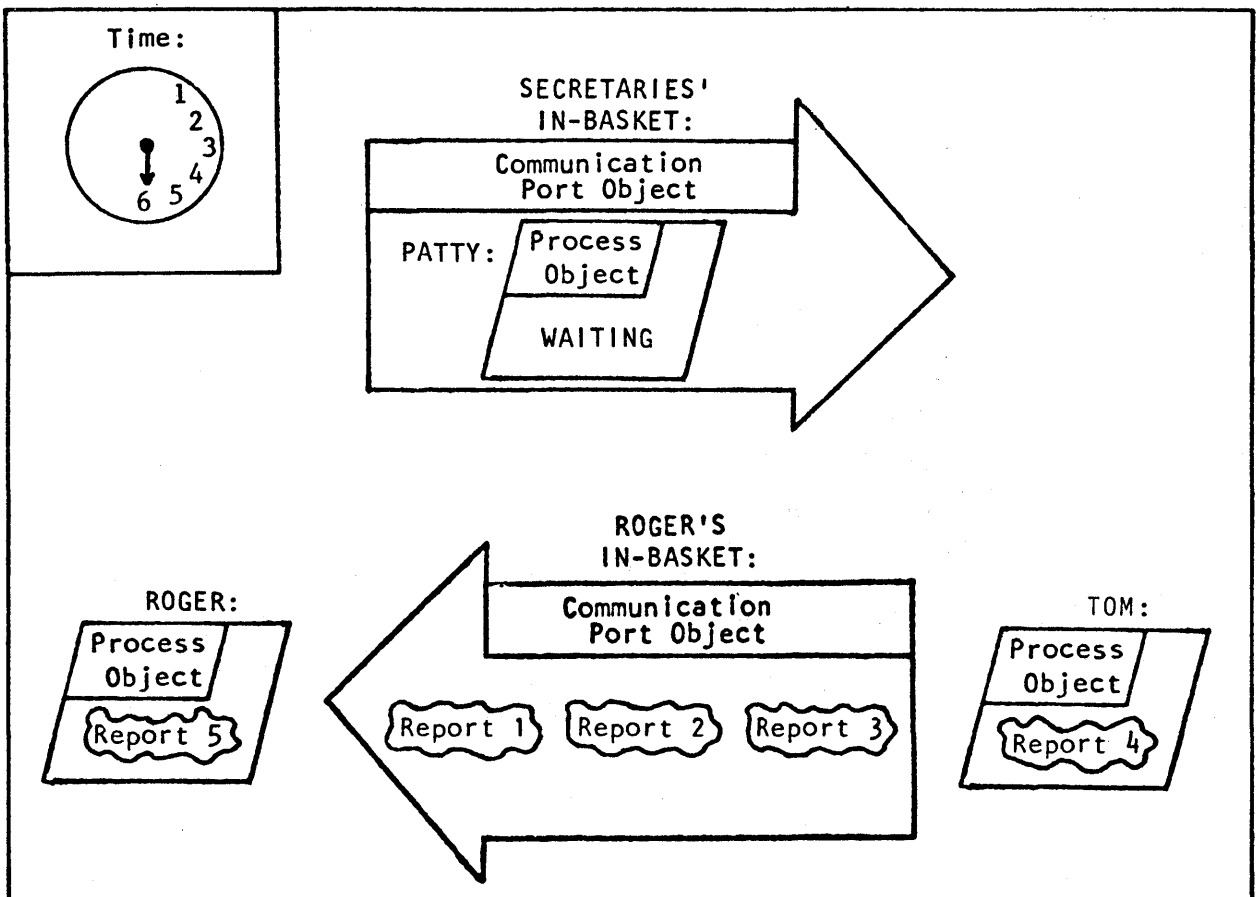


Figure 4.9.5 -- PATTY looks for more work in the secretaries' in-basket, but finds it empty, so she "climbs inside" the in-basket to wait for a message.

At Time 6 (Figure 4.9.6), only 2 processes are executing: ROGER is writing Report 5, and TOM is typing Report 4. PATTY is not executing right now, because she does not have anything to type. PATTY is waiting in the secretaries' in-basket and will not start executing until ROGER sends a report to the secretaries' in-basket.



- Two processes are executing concurrently:
  1. ROGER is working on Report 5.
  2. TOM is working on Report 4.
- PATTY is waiting for a message.

Figure 4.9.6

At Time 7 (Figure 4.9.7), ROGER has finished writing Report 5 and has executed a SEND instruction to send it to the secretaries' in-basket. PATTY, who has been waiting in the in-basket for a message, receives Report 5 immediately.

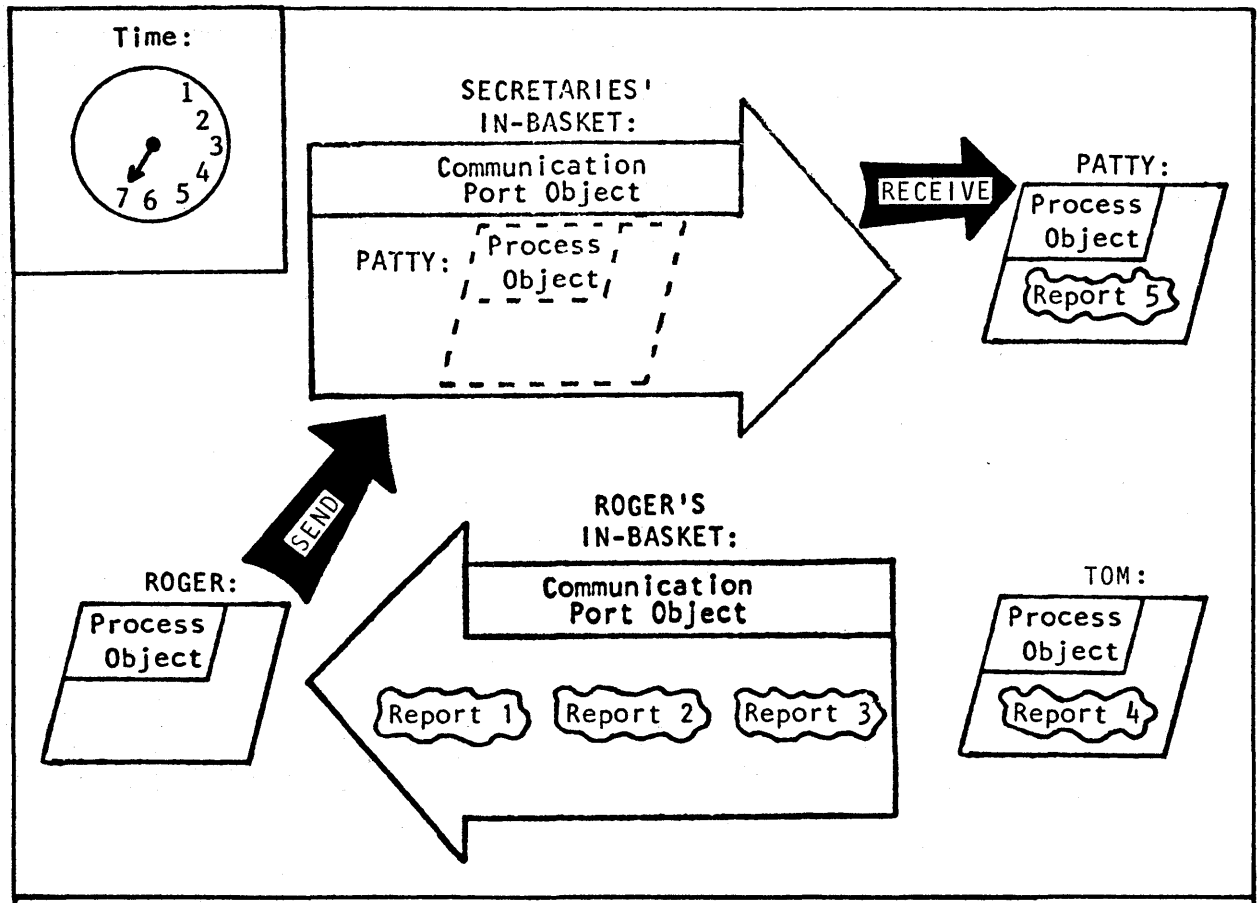
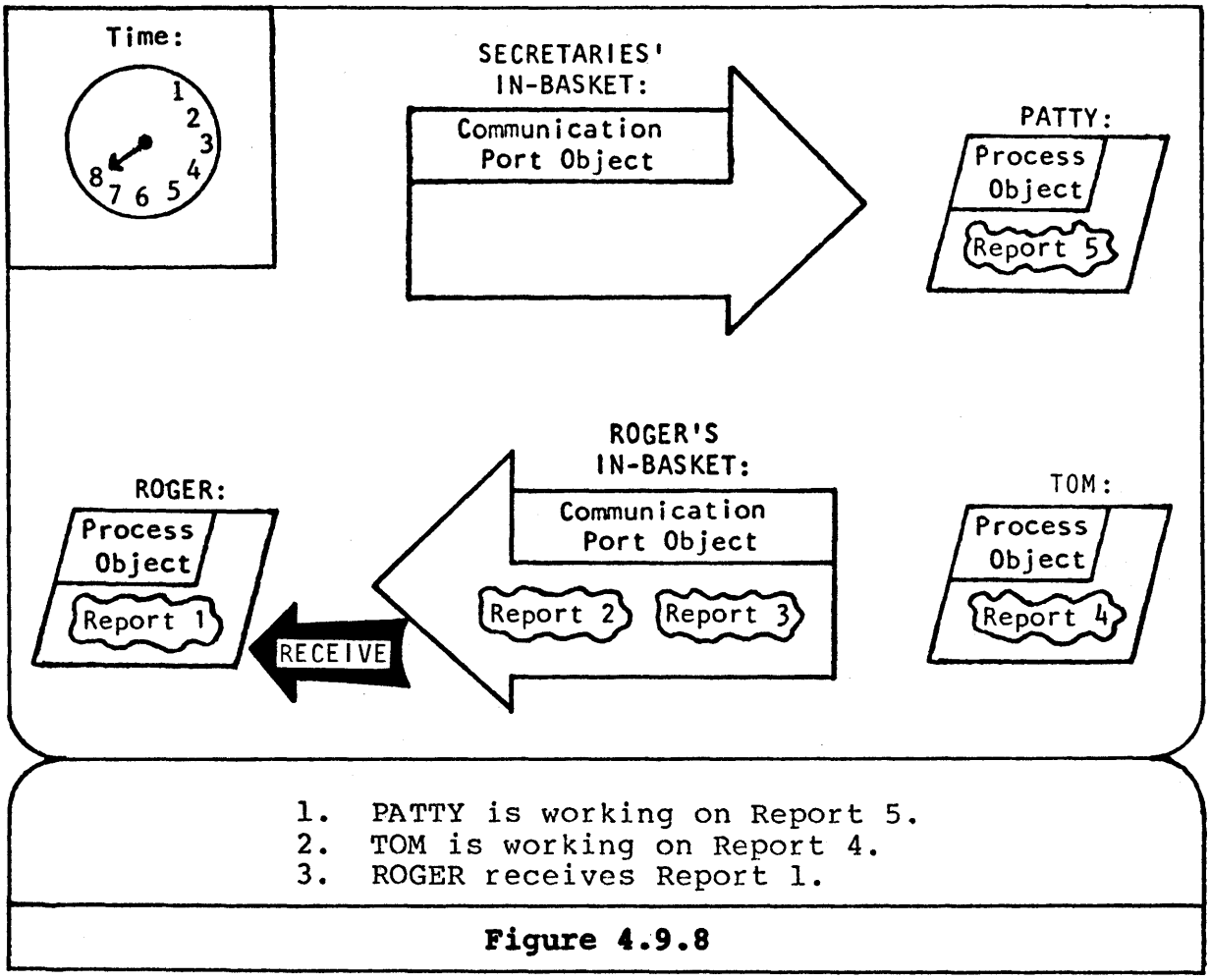


Figure 4.9.7 -- ROGER finishes Report 5 and sends it to the secretaries' in-basket where PATTY receives it immediately.

There are now 3 processes executing: 1) ROGER, which will either start writing another report or proofread one of the reports that is already typed; 2) TOM, which is still typing Report 4; and 3) PATTY, which has just received Report 5 and is starting to type it.

Finally, at Time 8 (Figure 4.9.8), TOM and PATTY are typing Reports 4 and 5, and ROGER is executing a RECEIVE instruction to receive Report 1 for proofreading.



1. PATTY is working on Report 5.
2. TOM is working on Report 4.
3. ROGER receives Report 1.

Figure 4.9.8

This concludes our example of interprocess communication; you should now have a good conceptual view of how processes use communication ports and SEND and RECEIVE operators to communicate with each other.

IT'S ALL DONE WITH OBJECT REFERENCES.

This section takes a closer look at the structure involved in interprocess communications, using the same REPORTER program example.

Figure 4.10 is a more detailed view of part of the REPORTER program. For simplicity, ROGER's in-basket is left out. For clarity, all the domain objects, instruction objects, and processor objects are left out. These objects are needed to run the program, but they are not needed to understand what's going on.

At Time 1 (Figure 4.10.1), ROGER is writing Report 1. Note that ROGER's process object has a reference for the current context, and the current context is able to access Report 1 because it has an object reference for it. This context can also access the communication port serving as the secretaries' in-basket, and can therefore send a message to this port. Note that PATTY's current context has an object reference for the secretaries' in-basket communication port and can therefore receive a message from this port.

At Time 2 (Figure 4.10.2), ROGER has finished writing Report 1 and has already sent it to the secretaries' in-basket. Note that the report itself has not moved, but the object reference for it has. ROGER can no longer access Report 1 because he no longer has a reference for it. PATTY cannot access Report 1 either because she does not have a reference for it.

At Time 3 (Figure 4.10.3), PATTY has executed a RECEIVE instruction. Note that Report 1 has not moved, but the object reference for it has. The object reference is no longer queued in the port; it is now in PATTY's context object. PATTY can now access the report.

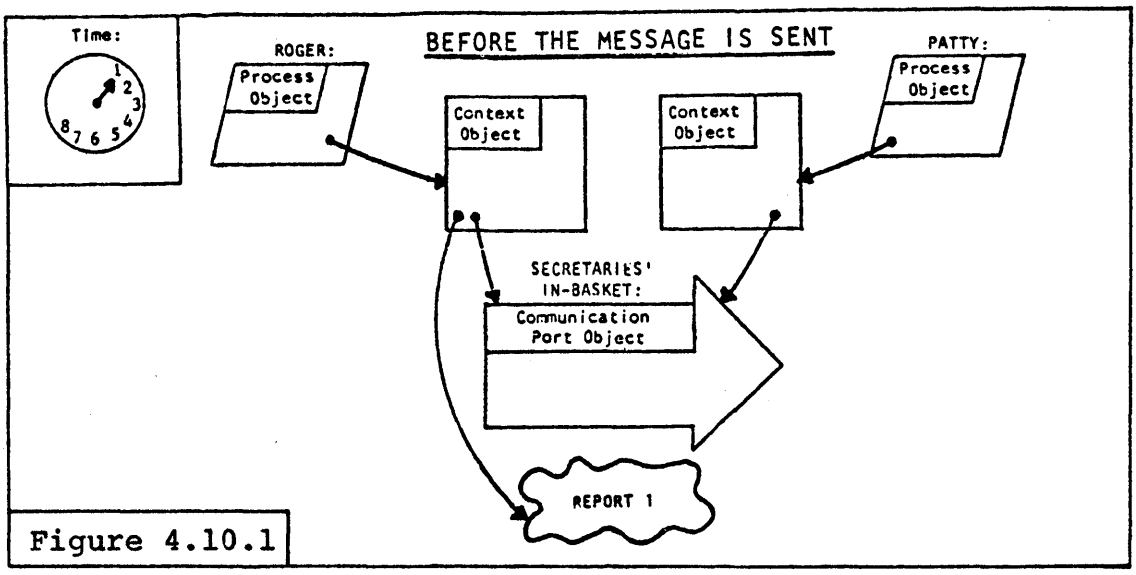


Figure 4.10.1

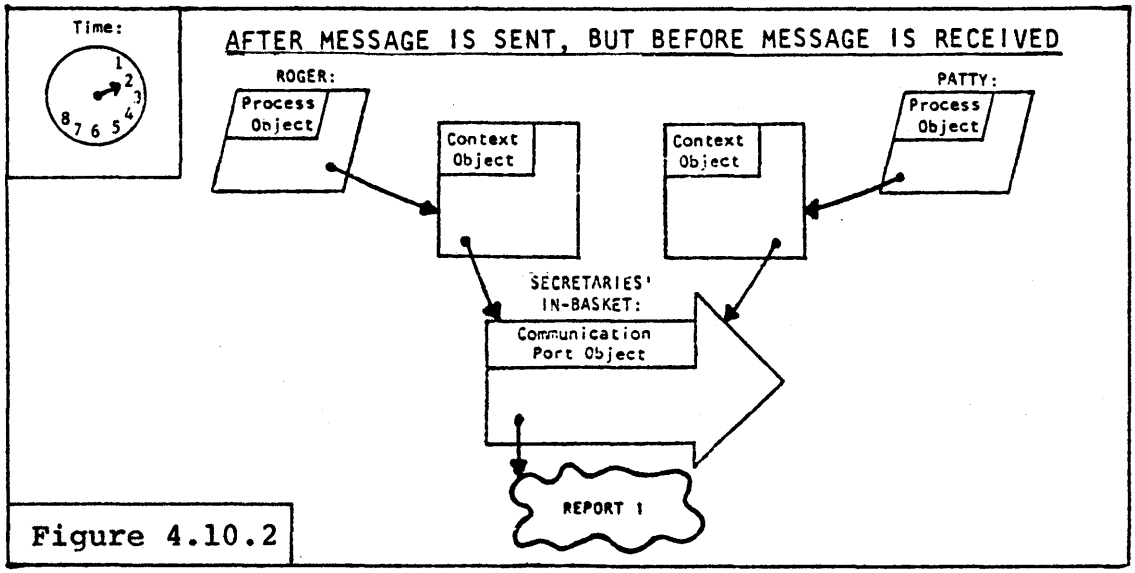


Figure 4.10.2

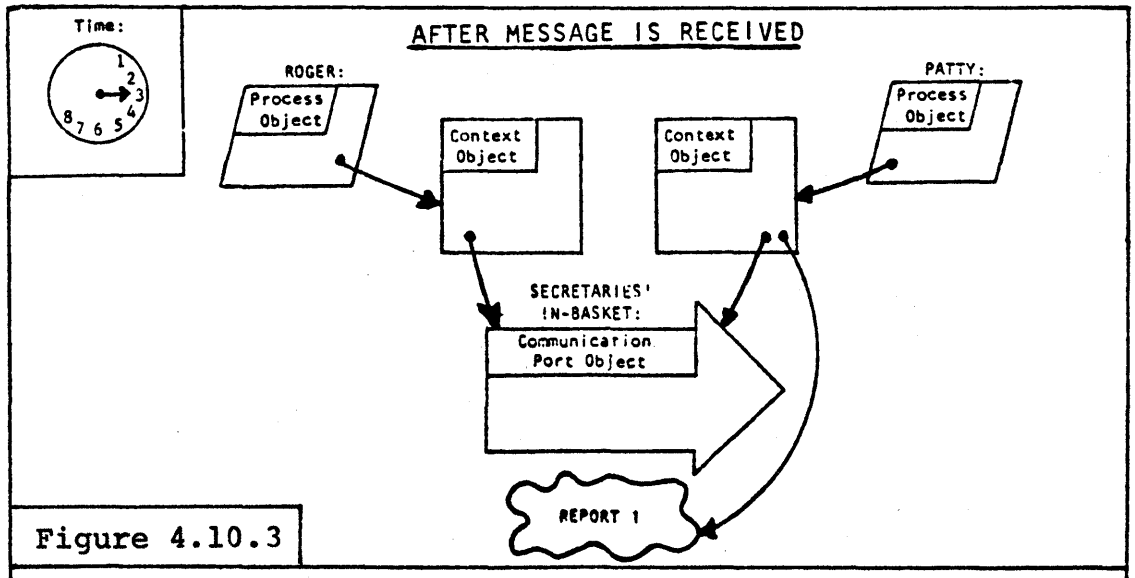


Figure 4.10.3

Figure 4.10 -- It's all done with object references.

INTERPROCESS COMMUNICATION QUIZ

1. The SEND and RECEIVE operators allow communication between concurrent _____.

2. Which of the following objects cannot be a message? Why?

   a. Communication port object
   b. Process object
   c. Context object
   d. Domain object

3. What happens when a SEND instruction is executed? (Describe the operands and the operation.)

4. What happens when a RECEIVE instruction is executed? (Explain both cases: (1) message waiting and (2) no message waiting.)

KEY TO INTERPROCESS COMMUNICATION QUIZ

1. The SEND and RECEIVE operators allow communication between concurrent processes.

2. Which of the following objects cannot be a message? Why?

   a. Communication port object
   b. Process object
   c. Context object
   c. Domain object

   All of the items listed are objects and any object can be a message. Therefore, all of the items above can be messages.

3. What happens when a SEND instruction is executed? (Describe the operands and the operation.)

   Operands: the message to be sent and the communication port it is to be sent to
   Action: The object reference for the message is moved from the context executing the send to the communication port specified as an operand.

4. What happens when a RECEIVE instruction is executed? (Explain both cases: (1) message waiting and (2) no message waiting.)

   Operand: the communication port from which the message is to be received
   Action:
   (1) Message waiting - The object reference for the message is moved from the communication port to the context object of the context executing the RECEIVE.
   (2) No message waiting- An object reference for the process executing the RECEIVE instruction is copied into the communication port. When a message is sent to the port, execution continues as in (1) above and the object reference for the process is deleted from the port.

Chapter 5

TRANSPARENT MULTIPROCESSING

The 432 is specially designed to support parallel execution of programs by multiple processors.

The earlier chapters of this book talked about two of the objects in the 432 architecture that support multiple processors. The first object discussed was the processor object. Chapter 3 explained that the 432 keeps processor-specific information in a processor object associated with each processor. This allows a system to have many processors without the problem many third-generation mainframe computers had, where each processor expected its diagnostic information and processor status to be kept in a particular physical memory location.

The second object discussed was the communication port. Chapter 4 talked about the support this object provides for communication between concurrent processes. When there is more than one processor in a system, it makes sense to structure a program so that it can run on more than one processor at a time. If you do, the program will run faster when additional processors are added. The interprocess communication facilities are important tools in structuring a program for simultaneous execution on multiple processors.

This chapter covers a third object that supports multiple processors: the dispatching port. The facilities provided by this object allow any software that executes on a 432 system to be run on systems with 1, 2, or many processors. Absolutely no software changes are required to move programs from single-processor systems to multiple-processor systems or vice versa. This is called transparent multiprocessing: The number of processors in a system is totally transparent to the software.

Transparent multiprocessing offers many advantages to users of 432 systems. The most obvious advantage is the flexibility provided by a machine that offers a range of performance. Processors can be added to or removed from a configuration to tune it to meet the desired performance or price.

A second advantage is that multiprocessor systems are inherently more reliable. If one processor fails, the rest of the system may be able to continue running.

This chapter has six sections that describe how dispatching ports support transparent multiprocessing:

- THE THREE STAGES OF PROCESSOR MANAGEMENT

  This section explains what the 432 does to manage multiple processors. The rest of this chapter explains how the 432 manages multiple processors.

- POLICY MAKING

  This section explains how the policy decisions are made that determine how processes share processors.

- SCHEDULING

  Scheduling and dispatching are the mechanisms that implement the policy. Scheduling is determining the order of process execution that implements the policy. This section explains how the 432 hardware performs process scheduling.

- DISPATCHING

  Dispatching is assigning a process to a particular processor for execution. This section explains how the 432 hardware does this.

- A DISPATCHING EXAMPLE

  This section reviews the REPORTER program example from the last chapter and walks through an example of how the 432 scheduling and dispatching mechanisms work.

- DISPATCHING PORTS AND PROGRAM STRUCTURE

  This section explains how dispatching ports fit into the basic program structure described in Chapter 3.

THE THREE STAGES OF PROCESSOR MANAGEMENT

Before I explain how the 432 manages multiple processors and multiple processes, I would like to sidetrack for a minute and explain what it does.

The effective management of multiple processors and processes requires three things:

- Policy Making
- Scheduling
- Dispatching

Policy making is setting the criteria that determine how processes share the processors. For example, a first-come, first-served policy allows the first process that is ready to run to grab a processor and use it until it is finished. If a second process is ready to run, it must wait until the first process completes before it gets a chance to run on the processor.

Another example of a scheduling policy is round-robin. With a round-robin policy, when a process is ready to run, it is placed at the end of the queue of processes waiting to run. Processes are removed from the front of the queue and given a short turn on a processor. If a process uses up the time allocated for its turn before it is finished running, it is placed at the end of the queue and the second process in the queue is given a chance to execute. The first process will get a second turn after all of the other waiting processes have had a first turn.

There are many other policies (priority, deadline, weighted round-robin, etc.), and no one policy is perfect for all applications. Many different policies can be used with a 432. The important thing to remember is that policy making determines how processes share processors.

The second element of processor management is scheduling. Both scheduling and dispatching are part of the mechanism that carries out the policy. Scheduling is the ordering of processes to run on processors in a manner that realizes the policy. For example, suppose there are two processes, Ann and Bob, in a queue waiting for execution by a processor. If a third process, Charlie, becomes ready to run, the round-robin policy can be implemented by scheduling Charlie to run after Ann and Bob. If Ann is still ready to run after she uses up her turn on the processor, the round-robin policy can be implemented by scheduling Ann to run after Bob and Charlie.

Thus, scheduling is the ordering of processes to run on processors in a manner that implements the policy.

The third element of processor management is dispatching. Dispatching is the assignment of processes to processors in the order in which the processes have been scheduled. In the example above, Ann has been scheduled to execute ahead of Bob and Charlie, but she has not been told which processor will execute her process. Ann is dispatched when she is assigned to a particular processor, e.g., if Processor 4 is looking for work and Ann is first in line, then Ann will be dispatched to begin execution on Processor 4.

In summary, the three elements of processor management are:

Policy Making -	Setting the policy that determines how processes share the processors
Scheduling -	Queueing the processes to run on a processor in an order that implements the policy
Dispatching -	Assigning (in the order scheduled) the processes to particular processors for execution

Having covered what the 432 needs to do in order to manage processes and processors, you can now look at how policy making, scheduling, and dispatching are done on the 432.

Before going on, I should warn you that this chapter covers only the facilities provided for short-term process/ processor management (policy making, scheduling, and dispatching) and does not cover long-term process/processor management. Short-term process/processor management is handled by the hardware, while long-term process/processor management is handled by software.

Let me explain what I mean by short-term process/processor management. Multi-user or multi-process systems need a mechanism for giving several users many small slices of processor time, so that each user sitting at a terminal has a reasonable response time for his/her requests. These time slices are typically 10 to 100 milliseconds long, and it is not unusual for a process to run for only a fraction of its allocated time.

Computers which do not provide a hardware mechanism for performing process switching must do so using software. Because this software is typically executed every 10 milliseconds or so, it can use a large share of the processor's time. It is not unusual for a processor to spend 30% or more of its time executing short-term scheduling software. This is why the 432 provides short-term scheduling hardware. Short-term scheduling is done so often that it makes sense to make it as fast as possible.

Long-term scheduling (anything over 3 seconds or so) is done much less frequently (about once per 100-1,000 short-term schedulings in a time-sharing system and even less often in other types of systems), so it has been left for the software.

POLICY MAKING

All policy making (i.e., deciding how processes share processors) is done by software. This is because it is very important for policies to be flexible. Different applications require different policies, and some applications require different policies under different conditions.

But, while it is important to keep the policy flexible by placing it in the software, it is also desirable to make the implementation of the policy efficient by moving the mechanism that carries out the policy into the hardware.

This has been done with the 432. The policy is set by operating system software, but the mechanism for scheduling and dispatching is built into the hardware.

Policy decisions made by the operating system are communicated to the hardware scheduling and dispatching mechanisms by means of scheduling parameters. The operating system sets scheduling parameters to tell the hardware how the processes should share the processor(s). Figure 5.1 shows that the scheduling parameters for a particular process are part of the information contained in the process object.

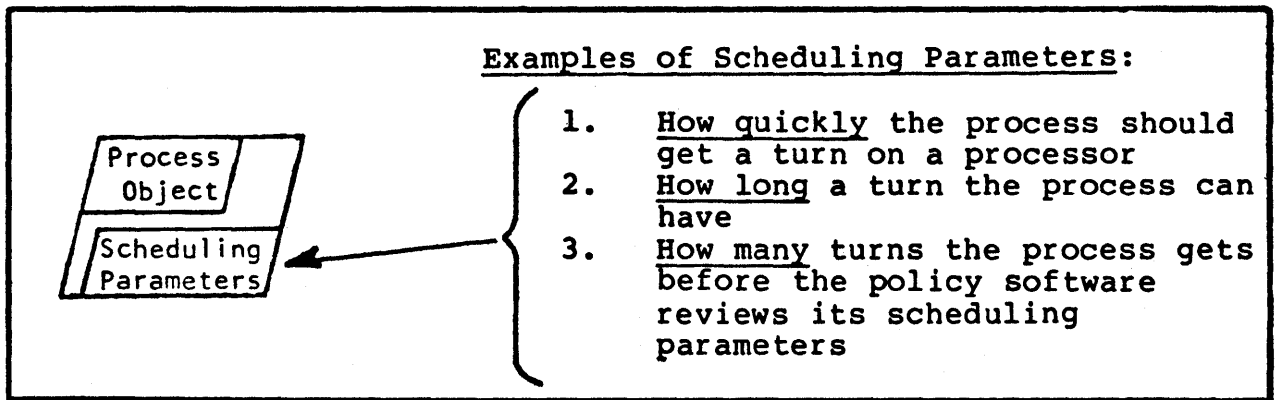


Figure 5.1 -- A process object contains scheduling parameters.

By setting these parameters to different values, the policy software can use the scheduling and dispatching hardware to implement a wide variety of policies. Figure 5.2 shows an operating system process (called POLICY PROCESS) setting the scheduling parameters on PROCESS A. Note that POLICY PROCESS can access the process object of PROCESS A because it has an object reference for it.

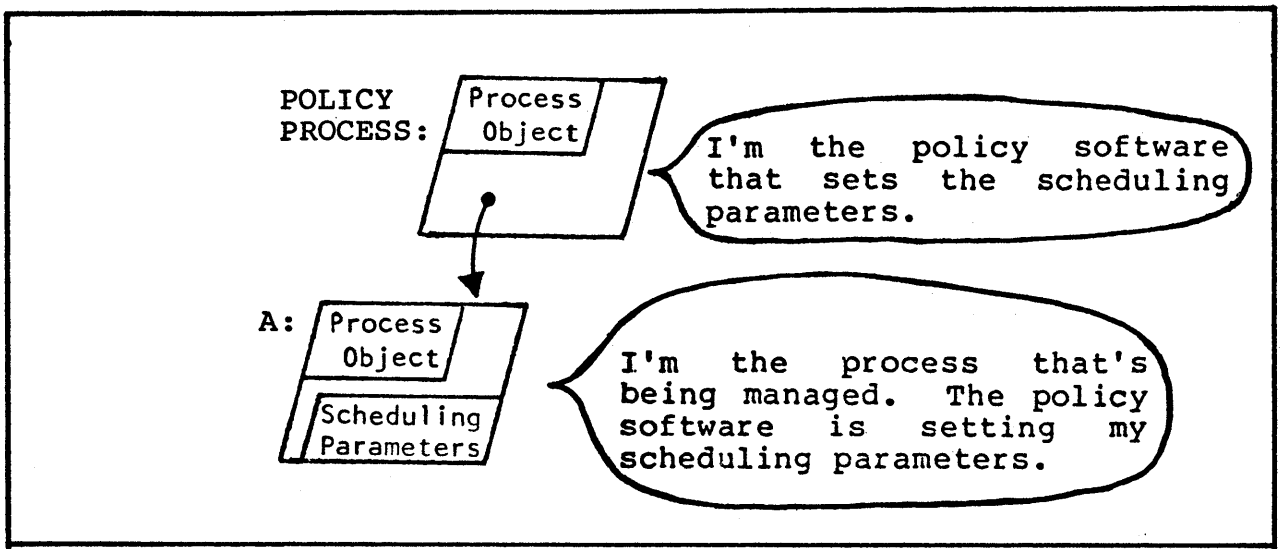


Figure 5.2 -- Policy software sets the scheduling parameters.

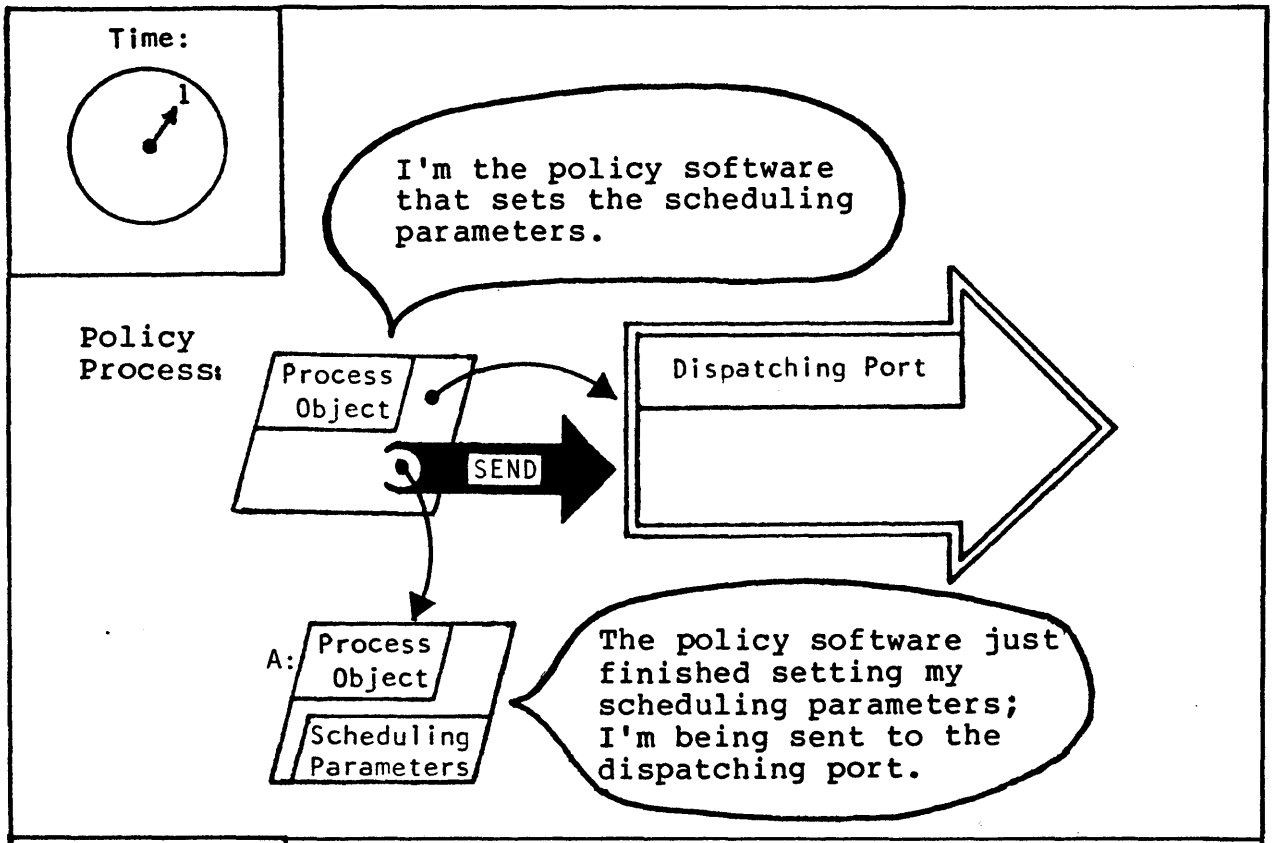
Notes: 1. This diagram is somewhat simplified. Actually, the POLICY PROCESS process object has an object reference for its current context, and the current context has the object reference for PROCESS A. I have left out the context object in Figures 5.2 and 5.3 to make them simpler.

2. POLICY PROCESS itself also has scheduling parameters that must be set, but I am not going to discuss the "chicken and the egg" problem.

SCHEDULING

The last section discussed how the first element of 432 process/processor management (i.e., policy making) is handled by the operating system software. This section and the next section explain how the 432 hardware mechanisms perform scheduling and dispatching.

Once the policy software has finished setting a process's scheduling parameters, it is ready to hand the process to the hardware for scheduling and dispatching. It does this by sending the process as a message (using the same SEND operator described in Chapter 4) to a hardware-defined object called a dispatching port. Figure 5.3 shows an operating system POLICY PROCESS sending PROCESS A to a dispatching port for scheduling and dispatching.



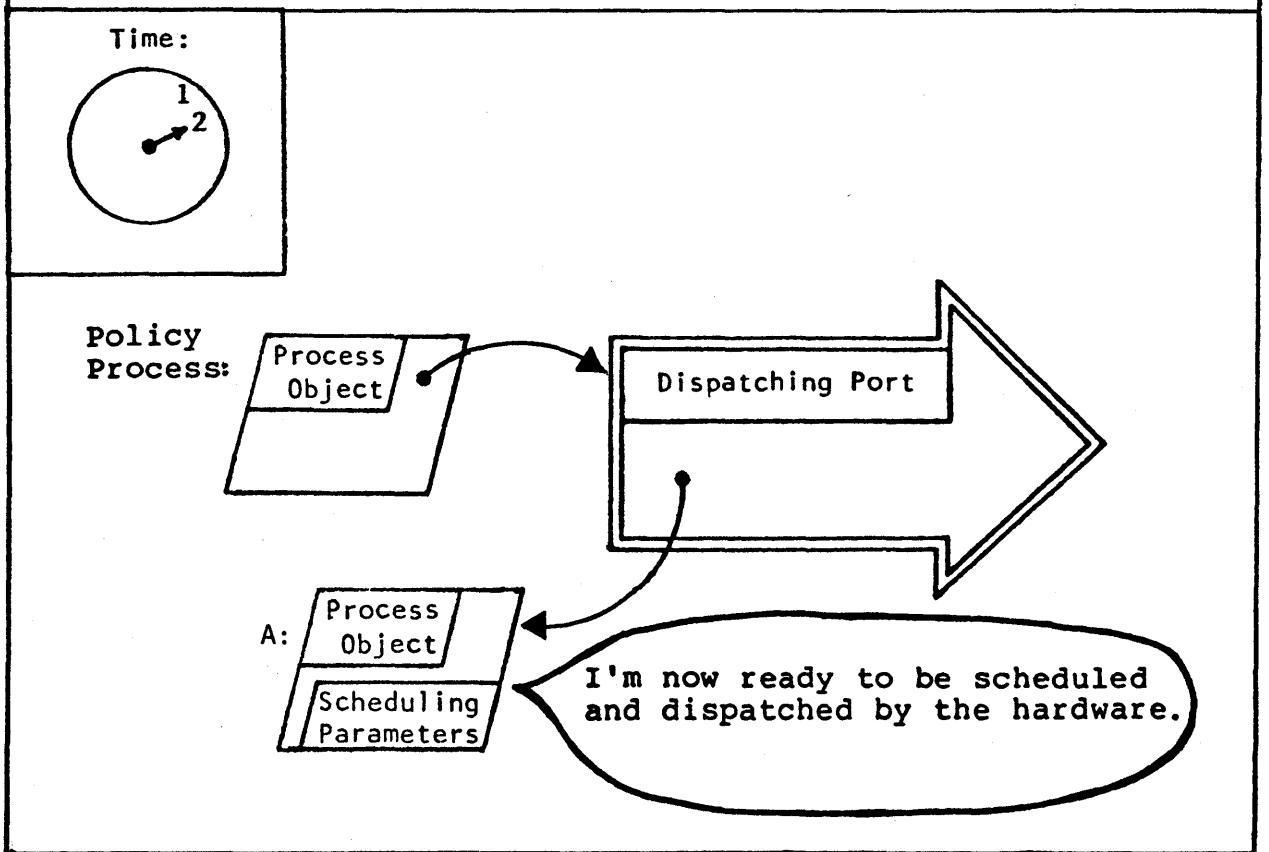


Figure 5.3 -- Processes are sent to dispatching ports for scheduling and dispatching.

Dispatching ports are where processes "line up" (are scheduled) for execution on a processor. The first process in line is the first process that will be assigned to a processor.

Dispatching ports are very similar to communication ports as you will see in a minute. But first, I'll explain one of the differences.

Messages sent to a communication port are normally placed in a first-in-first-out (FIFO) queue. When a process is sent to a dispatching port, its scheduling parameters are used to determine where it belongs in line. It may be placed anywhere in the line: the front, the back, or somewhere in the middle. Its position totally depends on its scheduling parameters and the scheduling parameters of the other processes in line at the port. This is illustrated in Figure 5.4.

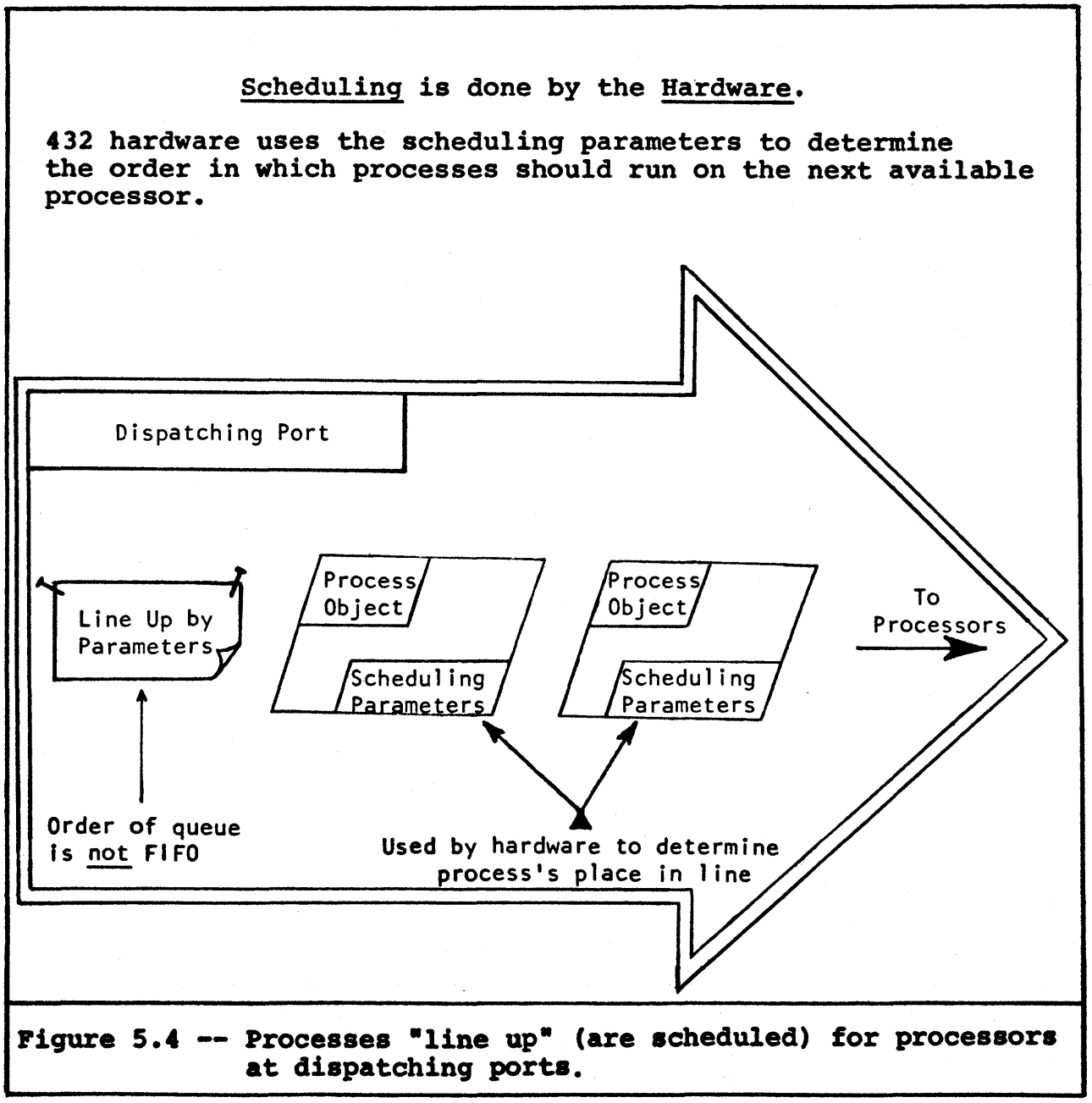


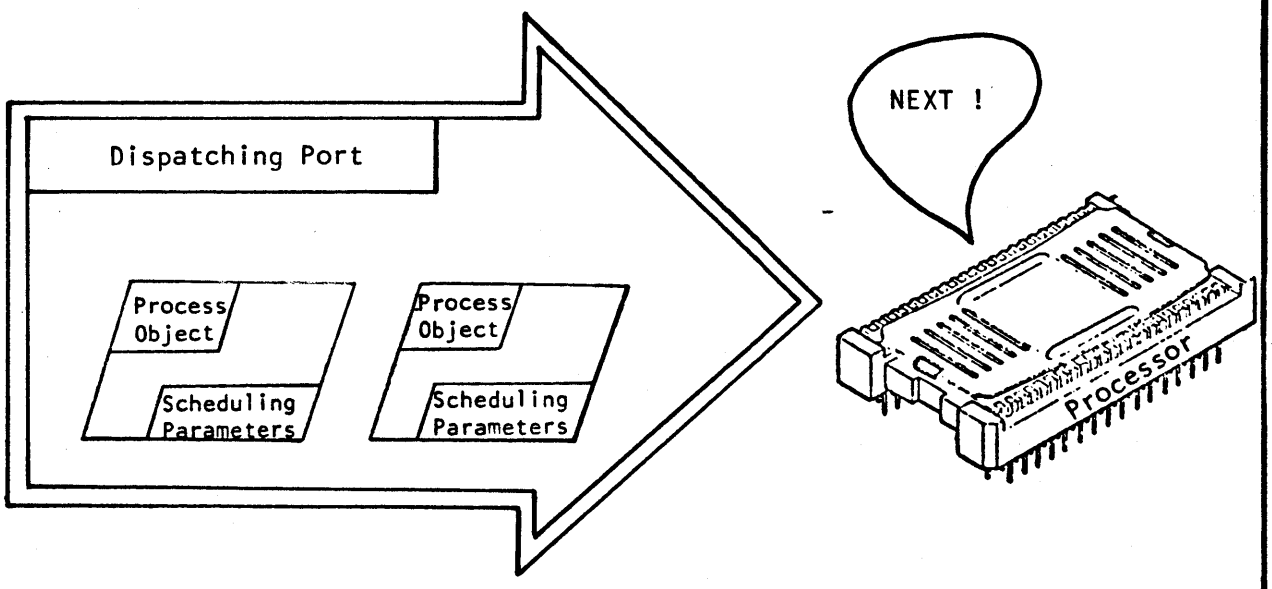
Figure 5.4 -- Processes "line up" (are scheduled) for processors at dispatching ports.

In summary, a process is scheduled as soon as it is sent to the dispatching port. The hardware compares its scheduling parameters with the scheduling parameters of the other processes already lined up at the port, and schedules the process by placing it in line at the appropriate position.

DISPATCHING

Figure 5.4 shows two processes lined up at a dispatching port. They have already been scheduled because they are already lined up in the order in which they will execute on a processor. All that remains to be done is to dispatch the processes, i.e., assign them to a physical processor for execution.

We have already seen that sending a process to a dispatching port is very similar to sending a message to a communication port. As we will see in a minute, a processor dispatching a process from a dispatching port is very similar to a process receiving a message from a communication port. Dispatching ports are the "meeting places" for processes and processors. Processes "stand in line" at dispatching ports to wait for service by a processor, and processors go to dispatching ports when they need a process to execute (see Figure 5.5).



- PROCESSES "stand in line" here for service by a PROCESSOR.
- PROCESSORS "stand in line" here for PROCESSES to serve.

Figure 5.5 -- Dispatching ports are meeting places for processes and processors.

When a processor needs a process to execute, it simply removes the first process in line at its dispatching port and begins to execute it. If there are no processes waiting at the dispatching port, then the processor waits at the dispatching port until one is sent to the port (just like a process waits for a message at a communication port), see Figure 5.6.

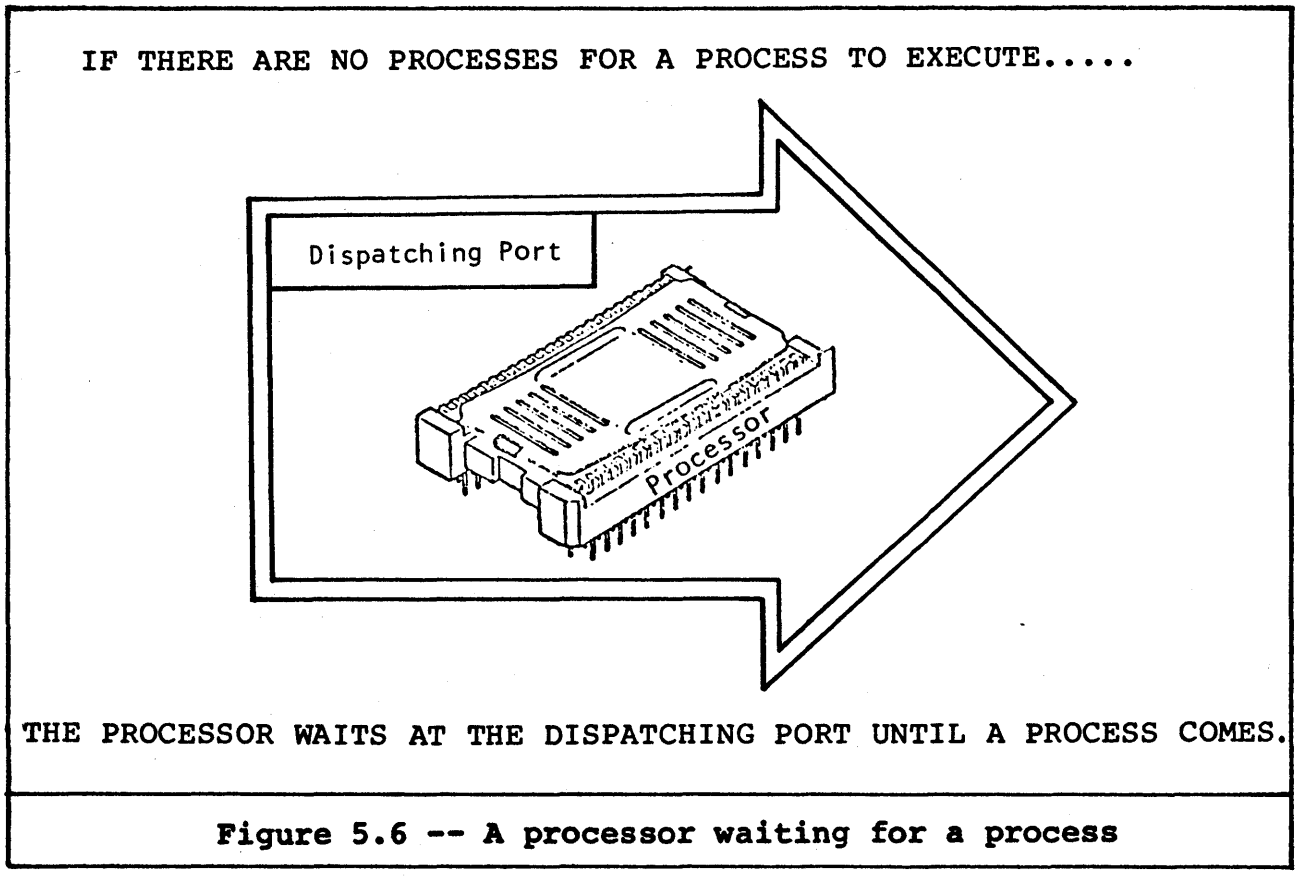


Figure 5.6 -- A processor waiting for a process

Figure 5.6 is a conceptual view of a processor waiting at a dispatching port. Figure 5.7 shows what really happens. An object reference for the waiting processor's processor object is queued at the dispatching port.

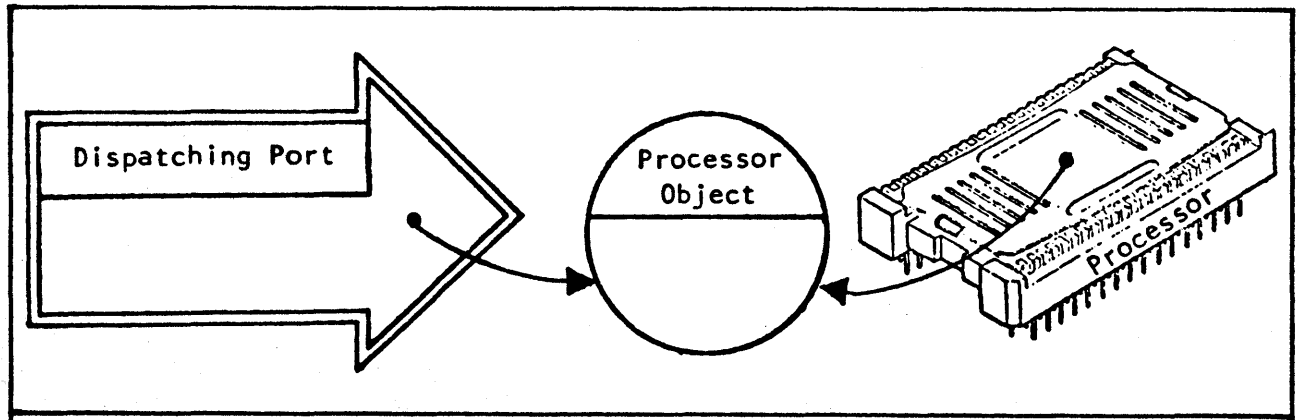


Figure 5.7 -- A processor waiting for a process

A DISPATCHING EXAMPLE

This section returns to the REPORTER program example discussed in the last chapter and adds more detail. This time, a dispatching port is included to see how it works. Figure 5.8 shows the structure of the REPORTER program and the dispatching port: There are three processes (ROGER, PATTY and TOM), two processors (1 and 2), and one dispatching port.

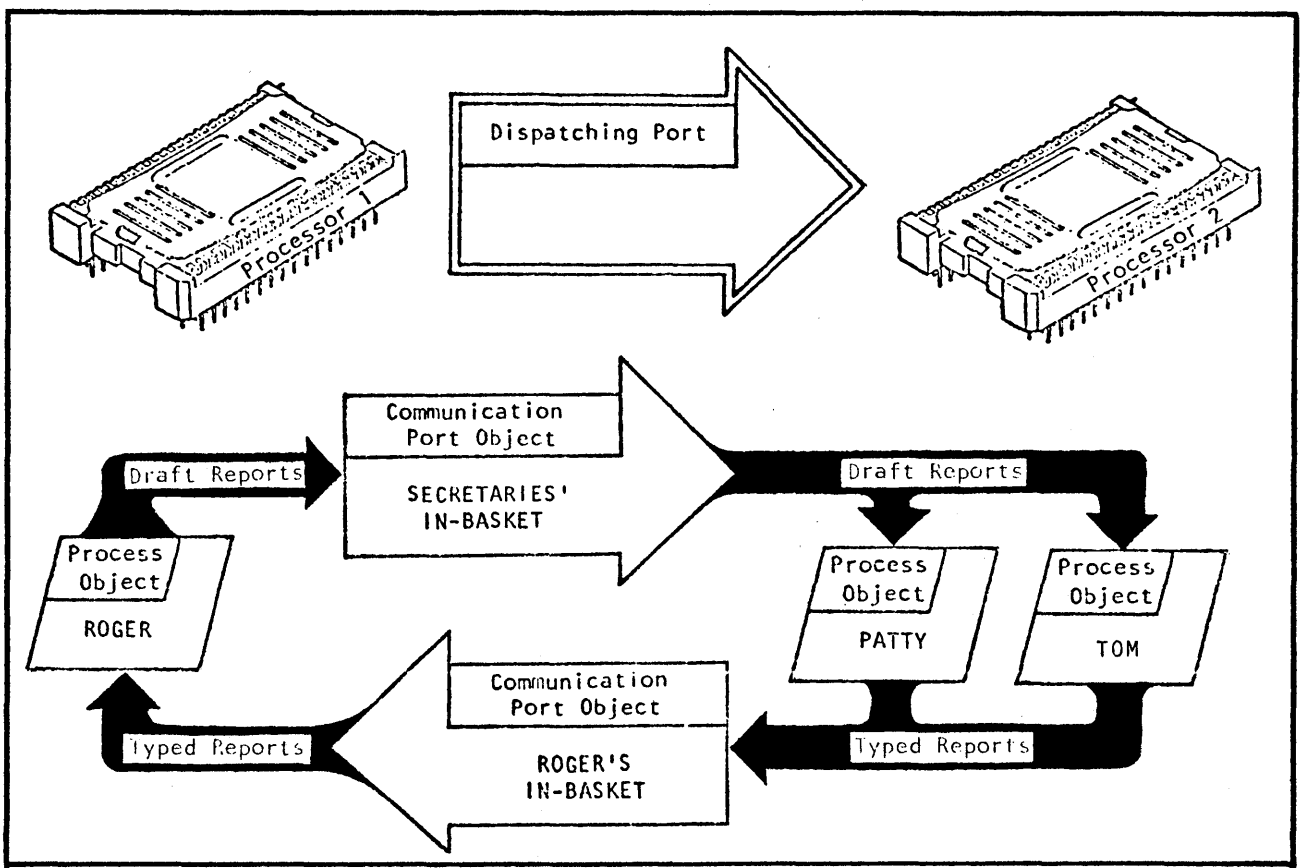


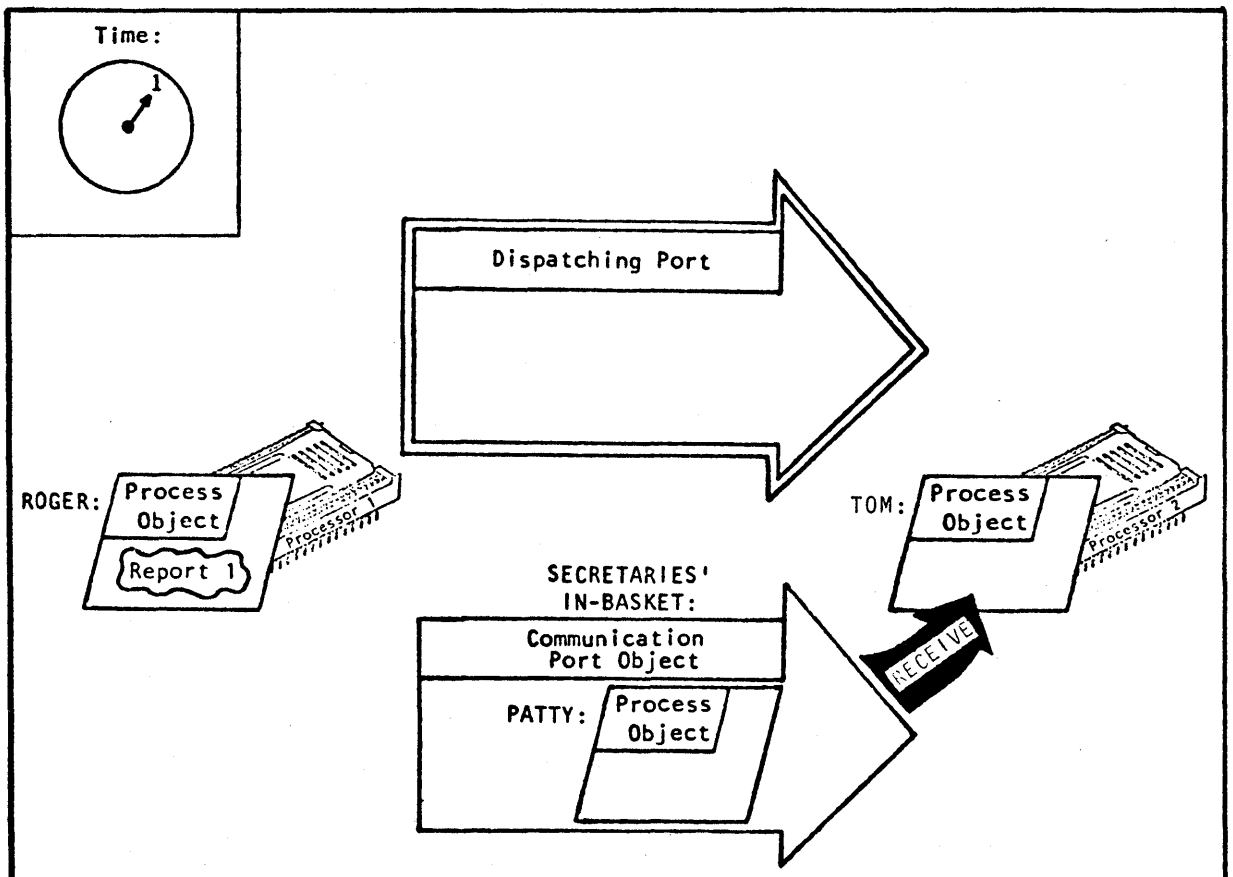
Figure 5.8 -- The Reporter program

Watch for two things in this example:

- There are two reasons for a processor to switch processes, i.e., stop executing one process and start executing another. What are these two reasons?
- The 432 hardware performs scheduling and dispatching, yet there are no 432 instructions called SCHEDULE or DISPATCH. Why?

The example begins with Figure 5.9.1. This figure does not show all the objects in Figure 5.8. I have left out ROGER's in-basket because it is not used in this example.

At Time 1 (Figure 5.9.1), there are two processes executing. ROGER is executing on Processor 1 and TOM is executing on Processor 2. PATTY is not executing; she is waiting in the secretaries' in-basket for a report to type. TOM is ready to type another report and has just started to execute a RECEIVE instruction.



TWO PROCESSES ARE EXECUTING

- ROGER is executing on Processor 1.
- TOM is executing on Processor 2.

ONE PROCESS IS WAITING

- PATTY is waiting for a report to type.

Figure 5.9.1

At Time 2 (Figure 5.9.2), TOM'S RECEIVE instruction has been partially executed. Because there were no messages in the communication port, TOM'S process has been queued up behind PATTY'S process to wait for a message. At this point, Processor 2 is finished with TOM'S process. There is nothing more for Processor 2 to do until TOM receives a message. Therefore, Processor 2 goes to the dispatching port to hunt for another process that is ready to execute.

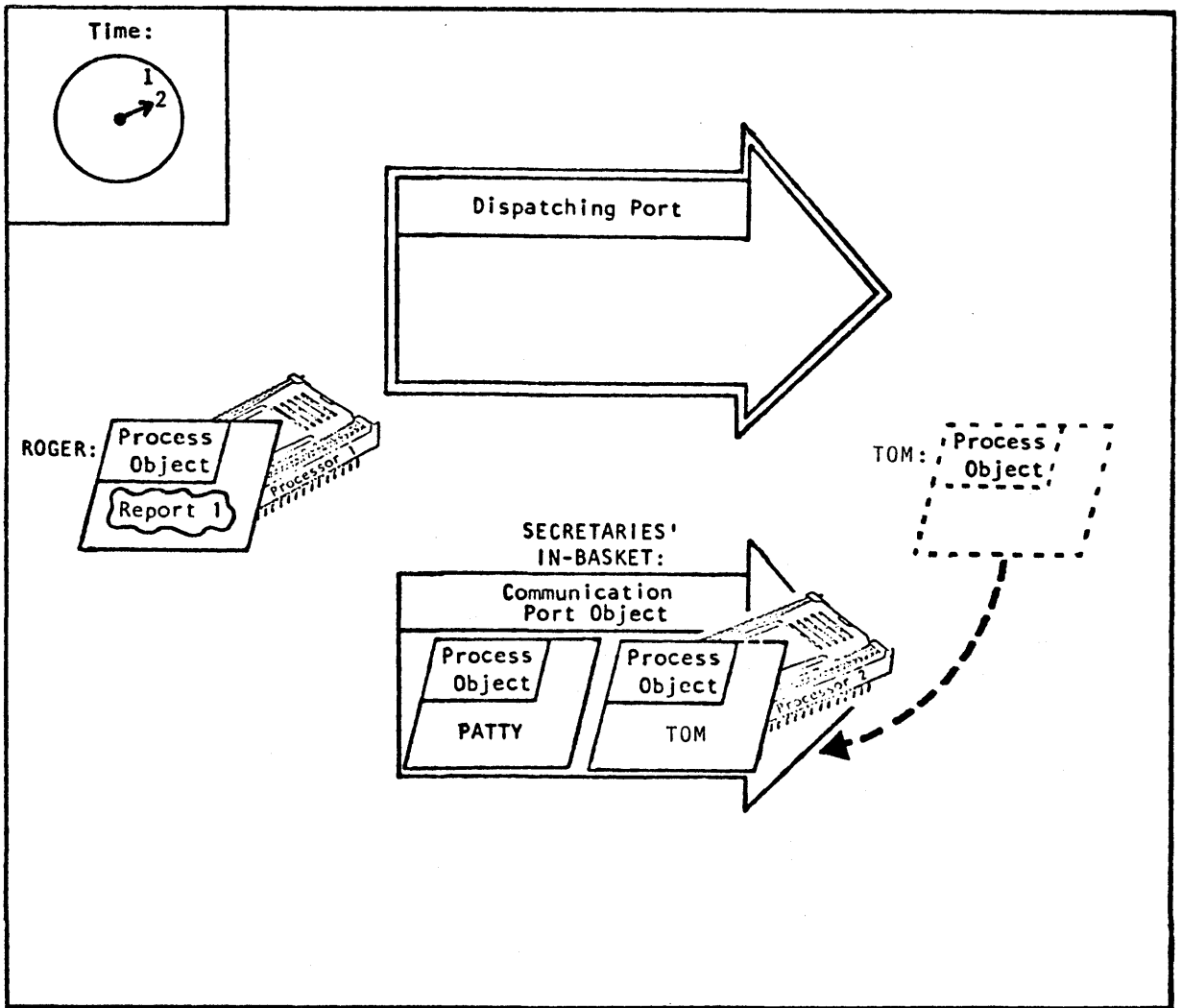


Figure 5.9.2 - TOM is queued to wait for a message.

At Time 3 (Figure 5.9.3), Processor 2 has checked the dispatching port and found it empty, i.e., no processes are ready to run. Processor 2 has, therefore, "climbed inside" the dispatching port to wait for a process that is ready to run. Processor 1 is still executing ROGER's process.

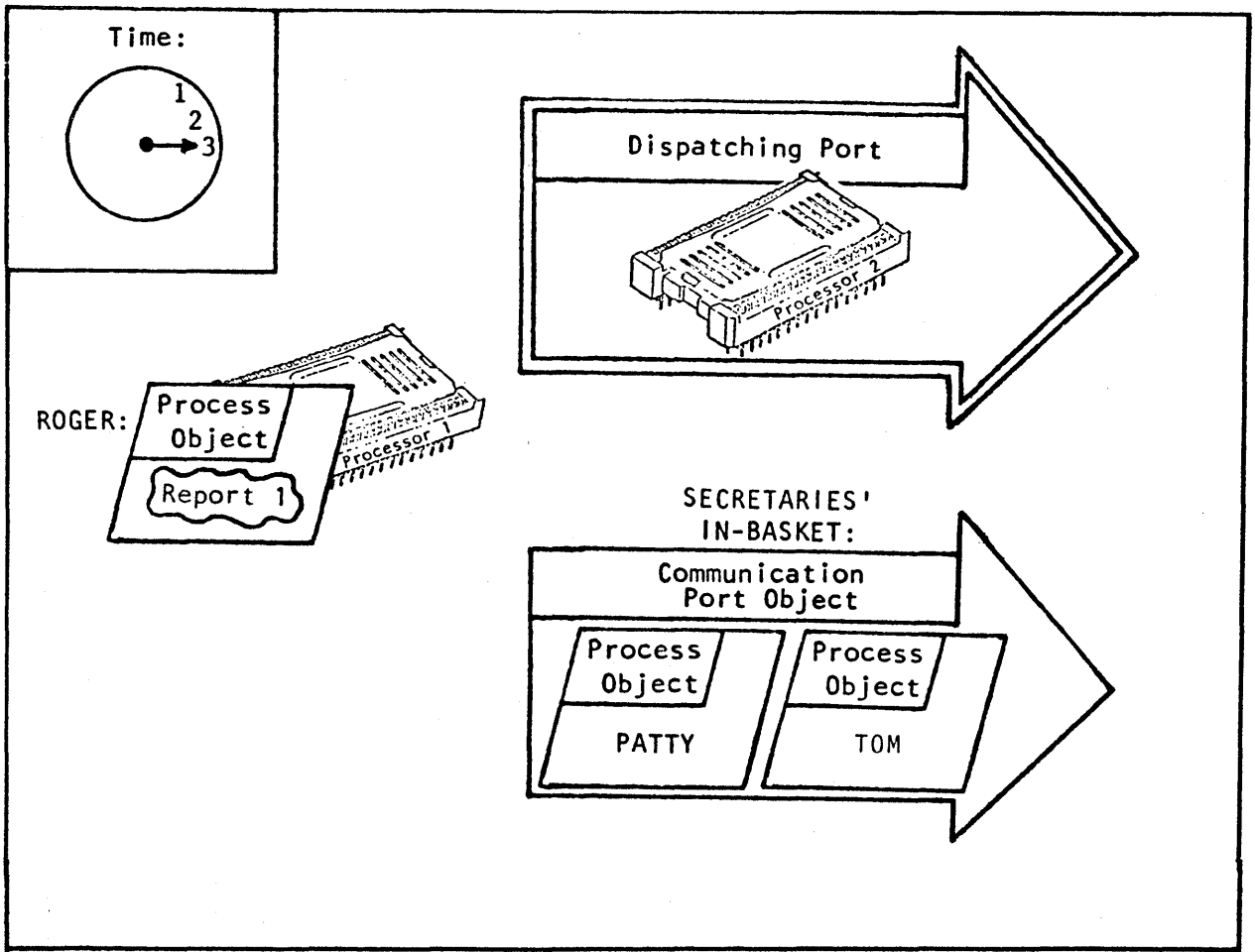


Figure 5.9.3 -- Processor 2 is waiting for a process that is ready to execute.

At Time 4 (Figure 5.9.4), ROGER finishes writing Report 1 and sends it to the secretaries' in-basket, where both PATTY and TOM are waiting for a report to type. Processor 2 is idle; it is waiting for a process to run.

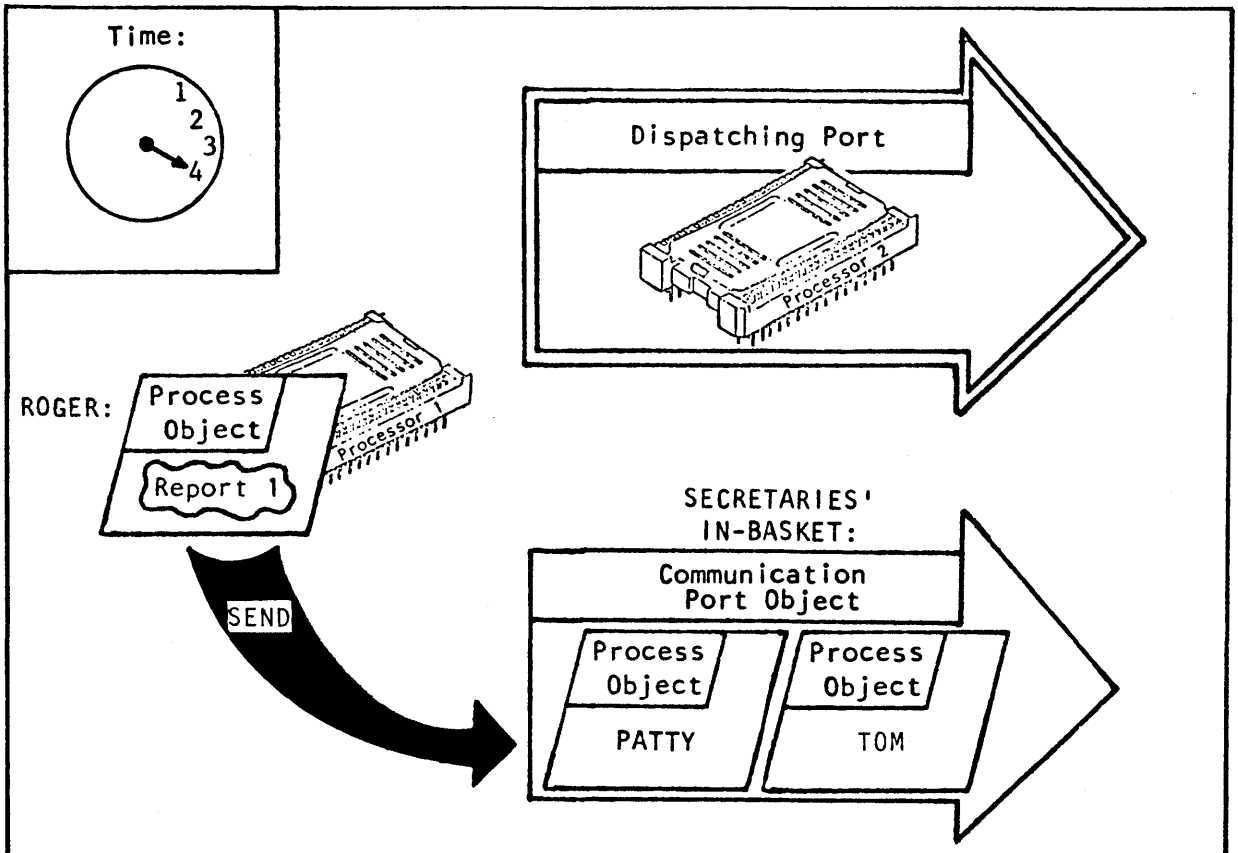
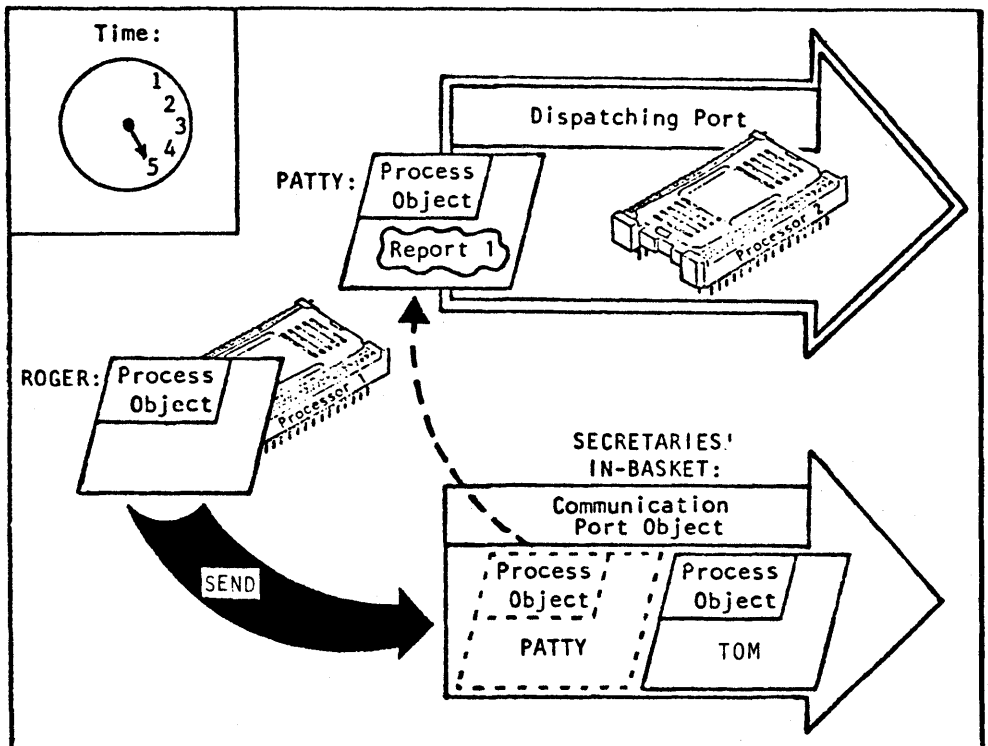


Figure 5.9.4 -- ROGER sends Report 1 to the secretaries' in-basket.

At Time 5 (Figure 5.9.5), ROGER'S SEND instruction has been partially executed. The message (Report 1) has been given to PATTY'S process, which was first in line at the port, and PATTY is now ready to run. As part of the SEND instruction, Processor 1 moves PATTY'S process to the dispatching port so that it can be scheduled to run on a processor. But, when Processor 1 moves PATTY to the dispatching port, it finds Processor 2 waiting for a process to execute. Processor 1 gives PATTY'S process to Processor 2 and tells it to start executing PATTY.

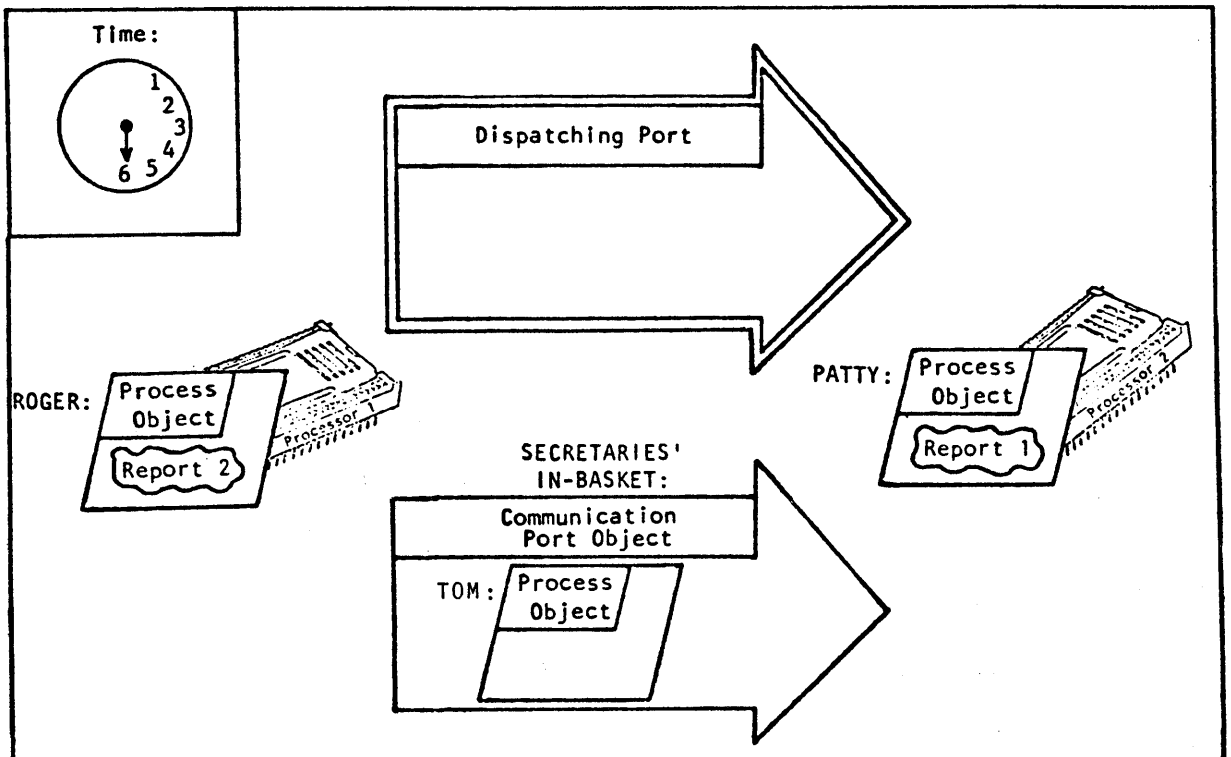


- When ROGER sends Report 1 to the secretaries' in-basket, it is given to PATTY.
- PATTY is now ready to run and is moved to the dispatching port, where Processor 2 is waiting.

Figure 5.9.5

Note that ROGER'S processor, Processor 1, not only sends the message to the communication port, it also gives the message to the first process waiting in line and moves that process to the dispatching port so that it can be scheduled and dispatched. In this case, because Processor 2 is waiting for a process to execute, Processor 1 also gives the process to Processor 2 and tells it to start working. Note that, during this time, ROGER'S processor, Processor 1, is really working for PATTY'S process so that it can get started. After Processor 1 gets PATTY started, it goes back to work for ROGER.

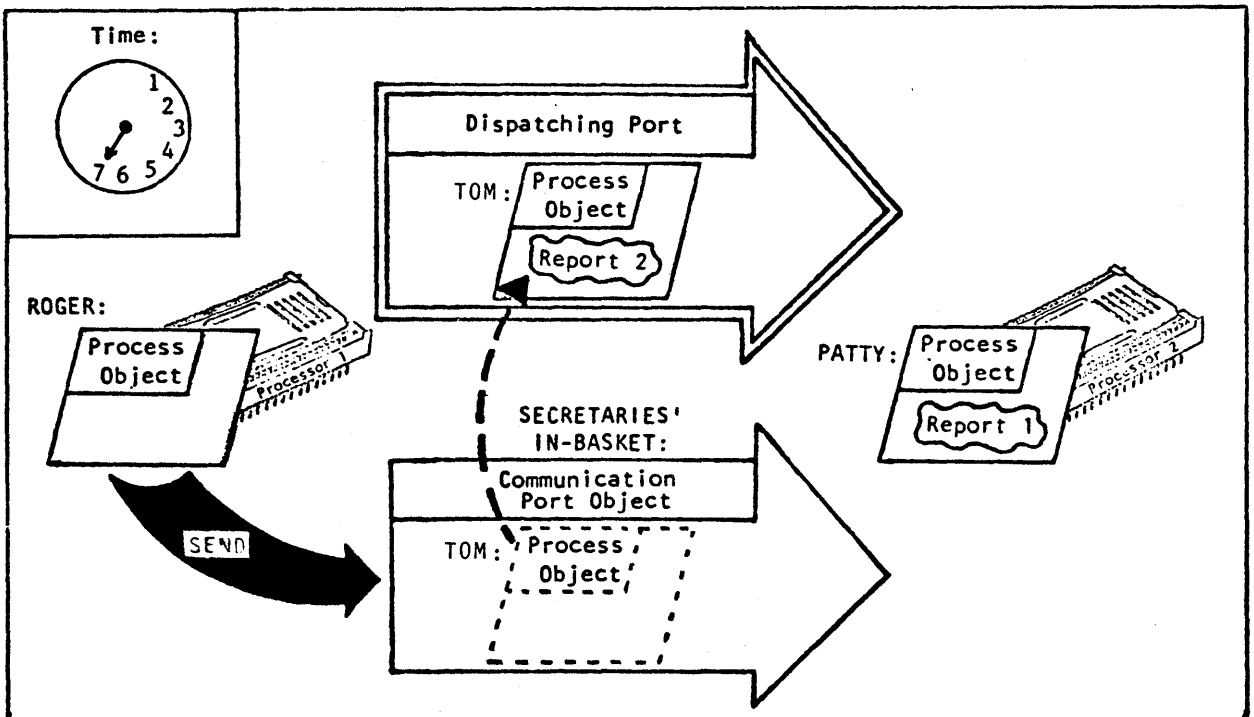
At Time 6 (Figure 5.9.6), there are two processes executing. ROGER is still executing on Processor 1 and PATTY is now executing on Processor 2. TOM is still not ready to execute; he is waiting in the communication port for a report to type.



- PATTY is now running on Processor 2.
- ROGER is still running on Processor 1.
- TOM is still waiting for a report.

Figure 5.9.6

At Time 7 (Figure 5.9.7), ROGER'S process is in the middle of executing a SEND instruction to send Report 2 to the communication port being used as the secretaries' in-basket. When Processor 1 moved the message to the communication port, it discovered that TOM was waiting for a message and therefore gave it to TOM. This made TOM ready to run, so Processor 1 moved TOM to the dispatching port for scheduling and dispatching. Note that moving TOM to the dispatching port and scheduling him for execution is all part of the SEND instruction.

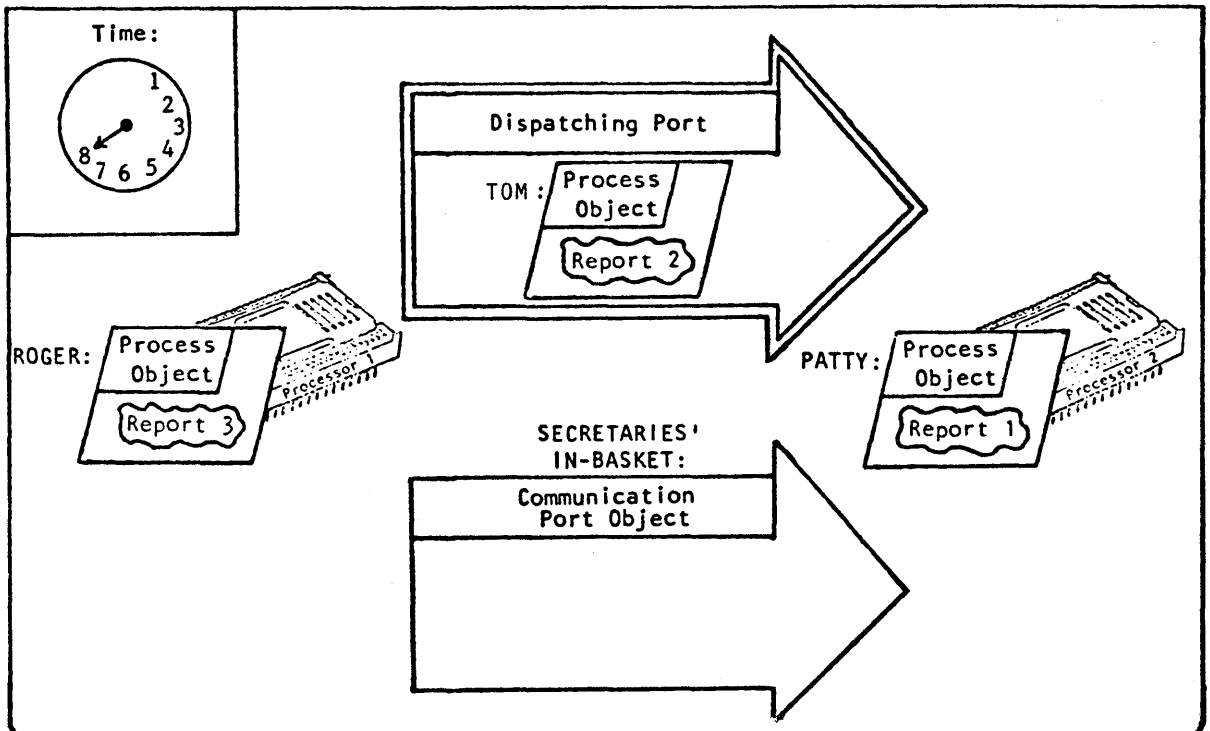


- ROGER sends Report 2 to the secretaries' in-basket.
- TOM is given Report 2 and is now ready to run.
- TOM is moved to the dispatching port to wait for a processor.

Figure 5.9.7

Here is another example where ROGER'S processor, Processor 1, has gone to work for another process (TOM'S) in order to get it scheduled and dispatched. This time, however, there was no processor waiting to execute the process, so Processor 1 had to be content with simply scheduling TOM so that the next processor that comes to the dispatching port looking for work will find TOM waiting and ready to run.

At Time 8 (Figure 5.9.8), TOM'S process is waiting in the dispatching port, Processor 1 is still executing ROGER'S process and Processor 2 is still executing PATTY'S process. Note that the major difference between Time 6 and Time 8 is that TOM is now ready to run and is waiting in a dispatching port. Whenever a process is ready to run (but a processor is not available) it waits in a dispatching port. Whenever a process is not ready to run -- because it is waiting for a message from another process -- it waits in a communication port.



- ROGER is still running on Processor 1.
- PATTY is still running on Processor 2.
- TOM is waiting for a processor in the dispatching port.

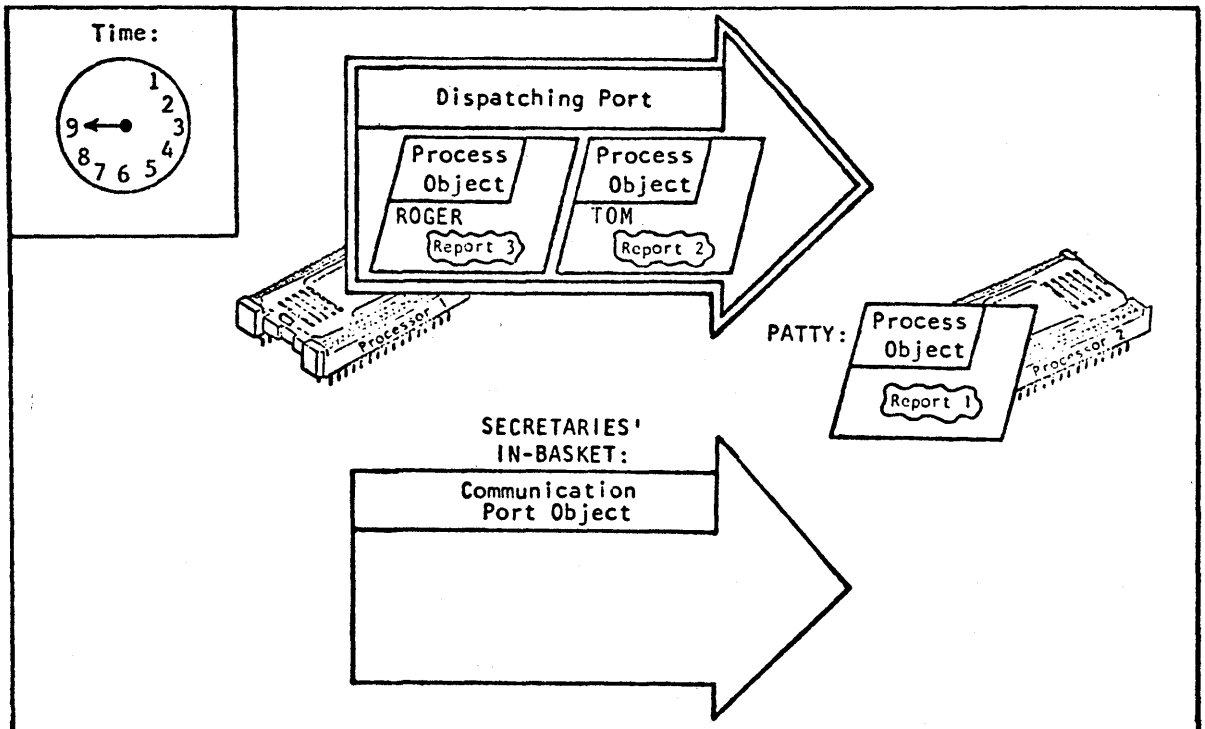
Figure 5.9.8

There have been two examples of what happens when a process is sent to a dispatching port. Either a processor is waiting at the port and starts executing the process immediately, or a processor is not waiting at the port and the process lines up to wait for a processor.

In both of the examples examined so far where a process was moved to a dispatching port, the process was moved because it had received a message and had become ready to run. In the next section of this example we will take a look at another reason for a process to be moved to a dispatching port -- because its time-slice expires.

At Time 9 (Figure 5.9.9), ROGER has used up his time-slice -- it is time for another process to get a share of the processor. Processor 1 has been keeping track of how long ROGER has been running and has noticed that ROGER has completed the amount of time specified by the scheduling parameter in ROGER'S process object that tells how long a time-slice ROGER gets.

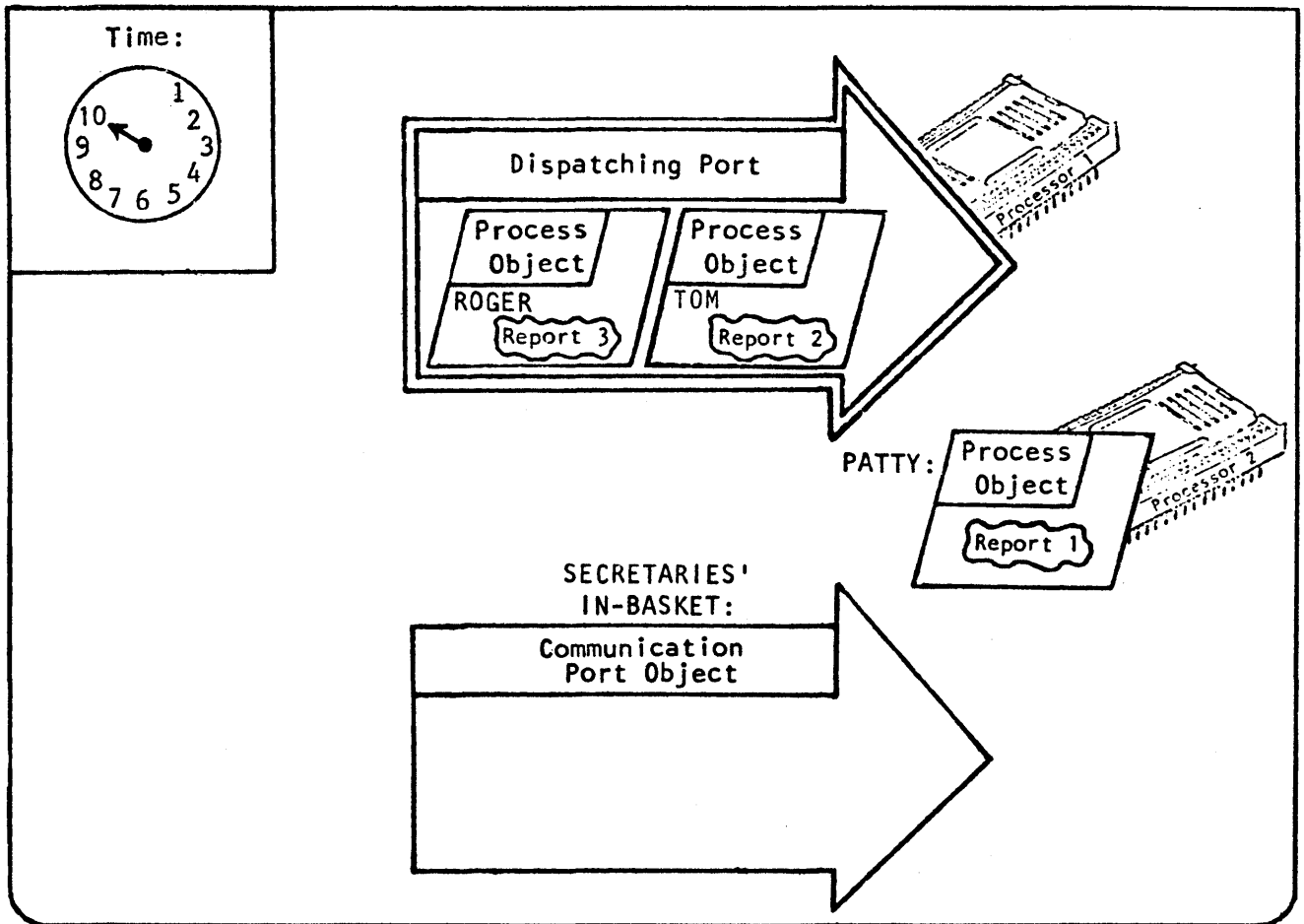
Processor 1 finishes the current instruction it is executing for ROGER and then places ROGER in the dispatching port. In the example, ROGER'S and TOM'S scheduling parameters indicate that ROGER should be scheduled to run after TOM, so ROGER'S process is placed second in line at the dispatching port.



- ROGER uses up his time period on Processor 1.
- ROGER is moved to the dispatching port to wait in line for another turn on a processor.

Figure 5.9.9

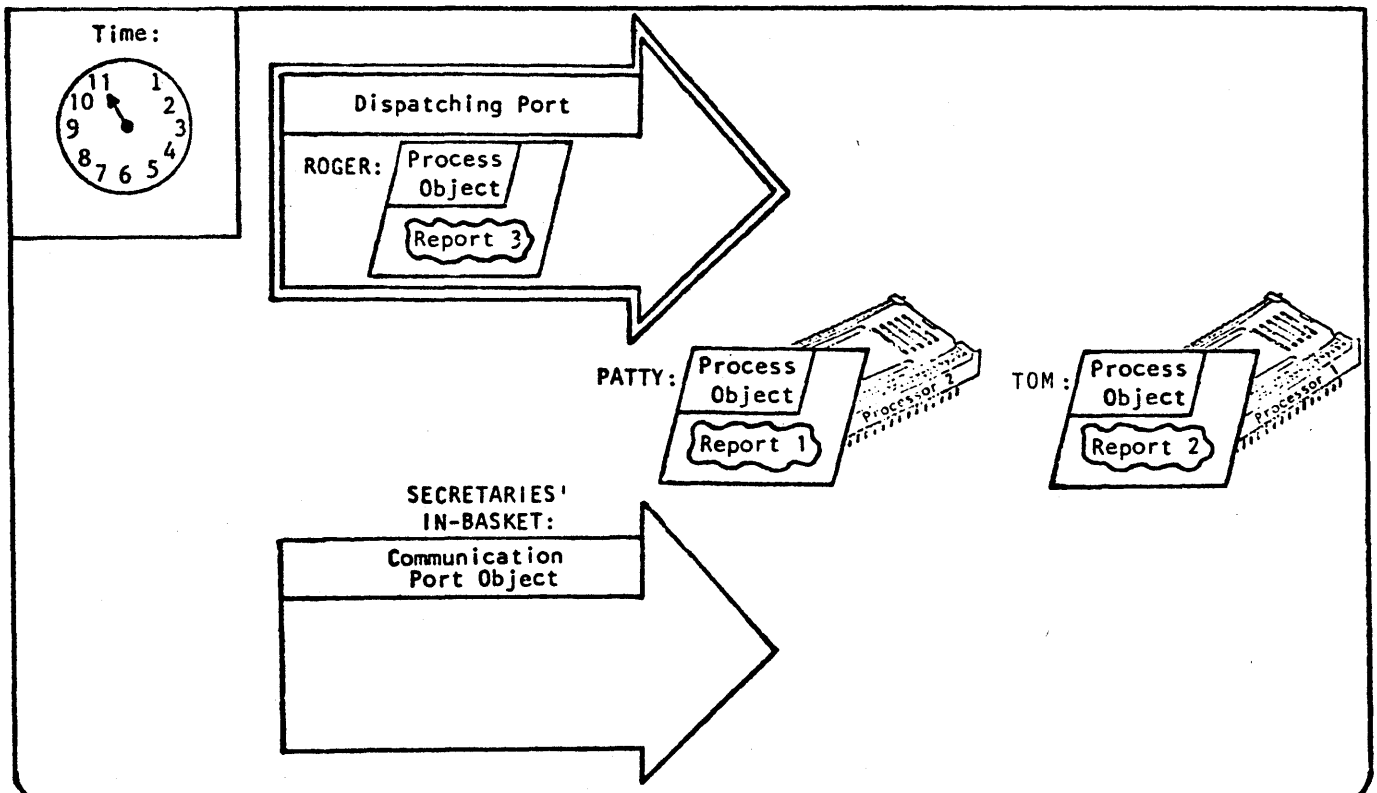
At Time 10 (Figure 5.9.10), ROGER has been inserted into the queue of waiting processes at the dispatching port. Processor 1 now needs a process to run, so it goes to the front of the queue at the dispatching port and removes TOM'S process for execution.



- Processor 1 now needs more work and goes to the dispatching port to find it.
- TOM is first in line and is removed from the dispatching port to run on Processor 1.

Figure 5.9.10

At Time 11 (Figure 5.9.11), there are two processes running. TOM is running on Processor 1 and PATTY is still running on Processor 2. ROGER is ready to run, and would be able to run if there were a third processor in the system. But, since there are only 2 processors in this example, ROGER must wait his turn at the dispatching port.



- TOM is now running on Processor 1.
- PATTY is still running on Processor 2.
- ROGER is ready for execution and is waiting in the dispatching port.

Figure 5.9.11

That concludes the example. By now, you should be able to answer the two questions I asked you earlier:

1. What are the two reasons for a processor to switch processes?

   A. _____

   B. _____

2. How can the 432 hardware perform process scheduling and dispatching, when there are no SCHEDULE and DISPATCH instructions?

ANSWERS:

1. A. The process being run executes a RECEIVE instruction on an empty communication port and must wait for a message.

   B. The process being run uses up its time-slice.

2. Process scheduling is done automatically whenever a process is sent to a dispatching port. Process dispatching is performed automatically by a processor whenever it needs to switch processes. There is no need to have SCHEDULE and DISPATCH instructions because the processor is "smart enough" to know when it needs to schedule and dispatch processes.

DISPATCHING PORTS AND PROGRAM STRUCTURE

Now that you know what dispatching ports are and how they work, you can see how they fit into the basic program structure described in Chapter 3.

By now it should be very clear that a 432 object cannot be accessed by a program unless the program has an object reference for it. This brings up two questions. First, how does a processor know which dispatching port to use when it is ready to dispatch a new process? Simple; one of the pieces of information that a processor keeps in its processor object is an object reference for its dispatching port (see Figure 5.10); it "follows" this reference to find its dispatching port.

Second question; how does a processor know which dispatching port to use when it needs to send a process to the process's dispatching port (e.g. to which dispatching port should a processor send a process that it finds waiting in a communication port when it is moving a message to the port)? Once again it is very simple. Every process object contains an object reference for its dispatching port (see Figure 5.10). The processor "follows" this reference to find its dispatching port.

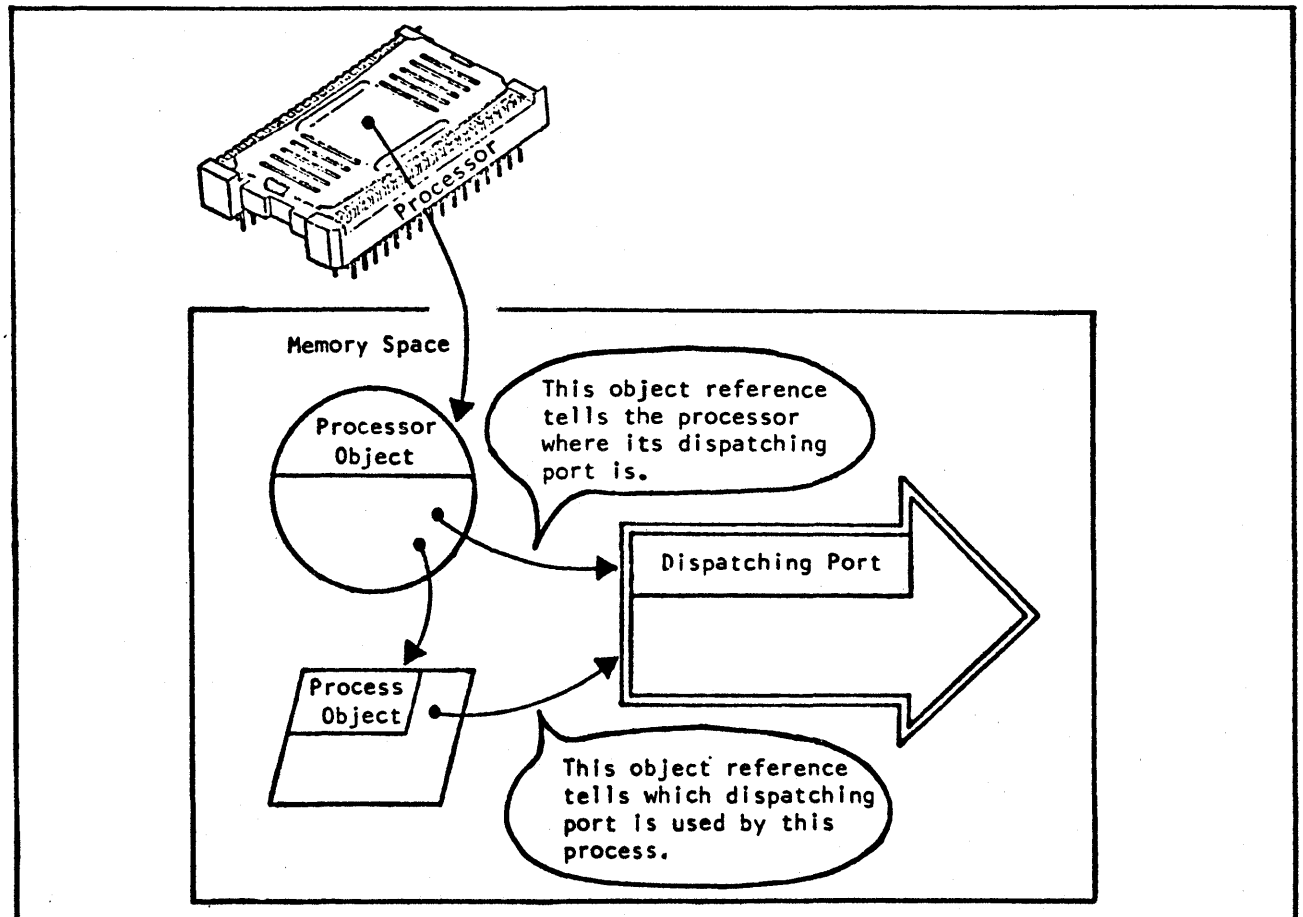


Figure 5.10 -- Each processor object and each process object has a reference for a dispatching port.

In the paragraphs above I have used phrases like "the processor's dispatching port" and "the process's dispatching port". These imply that there may be more than one dispatching port per system ... which is true; a system can have several. Usually, however, there is only one "central" dispatching port. All the processes that are ready to run go to this dispatching port, as do all of the processors that are looking for work. This sort of scheme minimizes the problem that can occur with multiple dispatching ports where processes are waiting ready to run at one port while processors are idle waiting for a process to run at another port.

Actually, the statement that there is usually only one dispatching port per system is a bit oversimplified. To be more precise, the statement should be: There is usually only one dispatching port for each type of processor. The 432 architecture not only allows for multiple processors; it allows for multiple types of processors. Different types of processors have different instruction sets and therefore different types of processes. It is important to run the right type of process on the right type of processor, so there must be at least one dispatching port for each type of processor.

TRANSPARENT MULTIPROCESSING QUIZ

1. What happens at dispatching ports?

2. What determines the order in which processes are run on a processor?

3. What happens if there are no processes for a processor to run?

4. What does the word "transparent" mean in the phrase "transparent multiprocessing"?

5. Processor management requires three things. Define each and tell where it is done by the 432 - in the hardware or the software.

	Done By	
	Hardware	Software
1. Policy Making	☐	☐
2. Short-Term Scheduling	☐	☐
3. Dispatching	☐	☐

KEY TO TRANSPARENT MULTIPROCESSING QUIZ

1. What happens at dispatching ports?

   Ready-to-run processes meet processors looking for work.

2. What determines the order in which processes are run on a processor?

   The scheduling parameters in the process object which were set by the policy software

3. What happens if there are no processes for a processor to run?

   The processor waits at the port until a process arrives. When a process arrives, the processor starts processing it.

4. What does the word "transparent" mean in the phrase "transparent multiprocessing"?

   It means that absolutely no changes need to be made to either applications or operating system software (no changes to any software) when additional processors are added to the system.

5. Processor management requires three things. Define each and tell where it is done by the 432 - in the hardware or the software.

		Done By		
		Hardware	Software	
1.	Policy Making	☐	☒	How the processes share the processors, e.g., round-robin, first-come, first-served, etc.
2.	Short-Term Scheduling	☒	☐	Ordering processes to run on processors in a manner that realizes the policy.
3.	Dispatching	☒	☐	Assigning processes (in order) to run on particular processors.

Chapter 6

DESIGNING SOFTWARE SYSTEMS

Chapter 3 described how hardware-defined objects provide the basic structure for all 432 programs. This chapter shows how the 432 allows additional objects to be defined in software and used to build modular, structured software systems.

This chapter has three major sections:

- DESIGN METHOD

  The 432 supports an object-oriented design method that is very different from (and much better than) conventional design methods. This section compares the methods and shows the advantages of the object-oriented approach.

- DOMAINS

  Domains are one of the architecture's facilities to support the object-oriented design method. This section describes domains and how they are used to build the static structure of a program.

- TYPE CHECKING

  Type checking is the architecture's second facility to support the object-oriented design method. This section describes type checking and how it fulfills two important needs.

1. DESIGN METHOD

This section describes the 432's object-oriented design method and compares it to a more conventional method.

There are three subsections in this section:

- THE SOFTWARE PROBLEM

  Why we need a good design method

- COMPARING 2 METHODS

  The major difference between an object-oriented design method and a conventional design method is the criteria used to divide a system into modules. This subsection compares the different criteria by looking at a sample system that has been "modularized" twice: once conventionally, and once with an object-oriented method.

- THE OBJECT-ORIENTED METHOD

  How the object-oriented method works and its major advantages

THE SOFTWARE PROBLEM

Managing complexity is one of the major problems facing today's system designers. As computers become more powerful, they are used for more challenging tasks. As a result, the software that controls today's computers has become very complicated.

The need for complex computer programs creates a need for a design method that allows effective management and production of large complex software systems. Historically, projects producing large software systems have had problems meeting schedules, producing reliable software, and producing maintainable software that can be changed and expanded as the system evolves.

The architecture of the 432 is designed around a better design method -- in fact, support for it is an integral part of the architecture. We call it an object-oriented design method, for reasons that you will understand shortly.

The easiest way to see how an object-oriented method differs from conventional methods is to study a sample system designed twice; once using a conventional design method and once using an object-oriented design method. This is done in the next section.

COMPARING TWO METHODS

This comparison of the two methods is a paraphrase of an article written by D. L. Parnas of Carnegie-Mellon University ("On The Criteria To Be Used In Decomposing Systems Into Modules", D.L. Parnas, December 1972, Communications of the ACM).

Summary

This section discusses modularization as a way to improve the flexibility and comprehensibility of a system while shortening its development time. The effectiveness of a modularization depends on the criteria used in dividing the system into modules. A system design problem is presented and both a conventional and object-oriented decomposition are described. It is shown that the object-oriented decomposition has distinct advantages for the goals outlined. The criteria used in arriving at the decompositions are discussed.

Introduction

A lucid statement of the philosophy of modular programming can be found in a 1970 textbook on the design of system programs by Gouthier and Pont (1, para. 10.23), which I quote below:[1]

> A well-defined segmentation of the project effort ensures system modularity. Each task forms a separate, distinct program module. At implementation time, each module and its inputs and outputs are well-defined, there is no confusion in the intended interface with other system modules. At checkout time the integrity of the module is tested independently; there are few scheduling problems in synchronizing the completion of several tasks before checkout can begin. Finally, the system is maintained in modular fashion; system errors and deficiencies can be traced to specific system modules, thus limiting the scope of detailed error searching.

---

[1]Reprinted by permission of Prentice-Hall, Englewood Cliffs, N.J.

Usually nothing is said about the criteria used in dividing the system into modules. This section discusses this issue and, by means of examples, suggests some criteria to be used in decomposing a system into modules.

A Brief Status Report

The major advance in the area of modular programming has been the development of coding techniques and assemblers which (1) allow one module to be written with little knowledge of the code in another module, and (2) allow modules to be reassembled and replaced without reassembly of the whole system. This facility is extremely valuable for the production of large pieces of code, but the systems most often used as examples of problem systems are highly-modularized programs and make use of the techniques mentioned above. Clearly, something is wrong with the way we are designing systems. We are not getting the benefits we expected.

The benefits expected of modular programming are:

1. Managerial -- development time should be shortened because separate groups can work on each module with little need for communication;

2. Product flexibility -- it should be possible to make drastic changes to one module without needing to change others;

3. Comprehensibility -- it should be possible to study the system one module at a time. The whole system can therefore be better designed because it is better understood.

The next several sections compare a conventional design modularization with an object-oriented design modularization. As you will see, the object-oriented modularization has all of the benefits listed above while the conventional modularization does not.

What is Modularization?

Below are several partial system descriptions called modularizations. In this context, "module" is considered to be a responsibility assignment rather than a subprogram. The modularizations include the design decisions which must be made before the work on independent modules can begin. Quite different decisions are included for each alternative, but in all cases the intention is to describe all "system level" decisions (i.e., decisions which affect more than one module).

The Example System: A KWIC Index Production System

A KWIC index program is exactly what its name implies -- a quick, automated way to produce an index. Every year, a tremendous number of technical journals are produced and KWIC index programs are used to produce indices that enable users to scan for key words in journal titles.

Here is a simple example of how it works. Suppose we have two titles as input to the KWIC index program:

TITLE	DOCUMENT NUMBER
X-ray Emmission By Quasars.	[1]
X-ray Diffraction Techniques.	[2]

The KWIC index produced would be:

X-ray Emmission	By	Quasars.	[1]
X-ray	Diffraction	Techniques.	[2]
X-ray	Emmission	By Quasars.	[2]
X-ray Emmission By	Quasars.		[1]
X-ray Diffraction	Techniques.		[2]
	X-ray	Emmission By Quasars.	[1]
	X-ray	Diffraction Techniques.	[2]

The center column of the KWIC index is an alphabetical listing of all the words in all the titles. The far right-hand column gives a reference number for a separate bibliography that includes information such as the author's name and where the article was published.

As an example of how one uses a KWIC index, say you are interested in locating articles on X-ray diffraction. The first thing you would do is think of key words that might appear in titles, e.g., "x-ray" and "diffraction". The second step is to scan the center column for titles containing these key words. The final step is to select interesting titles and use the document number in the right hand column to locate the full reference in the bibliography.

The output format shown above, where the key words are in the center, is only one of several ways to format the output from a KWIC index program. It is probably one of the most useful formats for actually using a KWIC index, but it is not the most useful for describing how a KWIC index is produced. As a result, the remainder of this document displays KWIC indices in a different format.

This format has two minor differences. First, it leaves off the document number. This makes the diagrams simpler. Second, and more important, the column of alphabetized words appears on the left of the page instead of in the center. Words in the title that occur before the key word are added at the end of the line. Thus, the KWIC index above would appear as:

By	Quasars. X-ray Emmission
Diffraction	Techniques. X-ray
Emmission	By Quasars. X-ray
Quasars.	X-ray Emmission By
Techniques.	X-ray Diffraction
X-ray	Emmission By Quasars.
X-ray	Diffraction Techniques.

This is a small system. Except under extreme circumstances (huge data base, no supporting software), such a system could be produced by a good programmer within a week or two. Consequently, none of the difficulties motivating modular programming are important for this system. Because it is impractical to treat a large system thoroughly, we must go through the exercise of treating this problem as if it were a large project. One modularization is given which typifies conventional approaches, and another which uses an object-oriented approach.

Conventional Modularization

The modules described below fit together to produce a KWIC index program. The function of this program is summarized by Figure 6.1. The original data file containing the titles is read by the KWIC index program which in turn produces a file with a KWIC index.

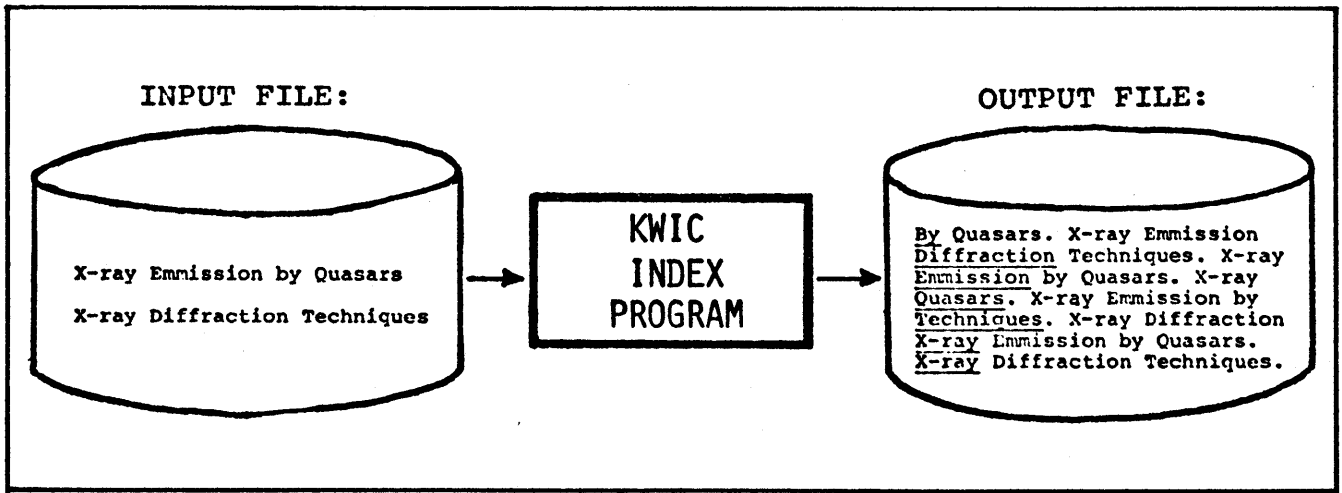


Figure 6.1 -- Overview of KWIC index program

As I explain the modules that make up this program, you get a closer look at the intermediate steps involved in producing a KWIC index.

Input Module. This module reads the data lines from the input medium and stores them in a TITLE TABLE in main memory for processing by the remaining modules. The characters are packed four to a word, and an otherwise unused character is used to indicate the end of a word. Figure 6.2 summarizes the action of this module.

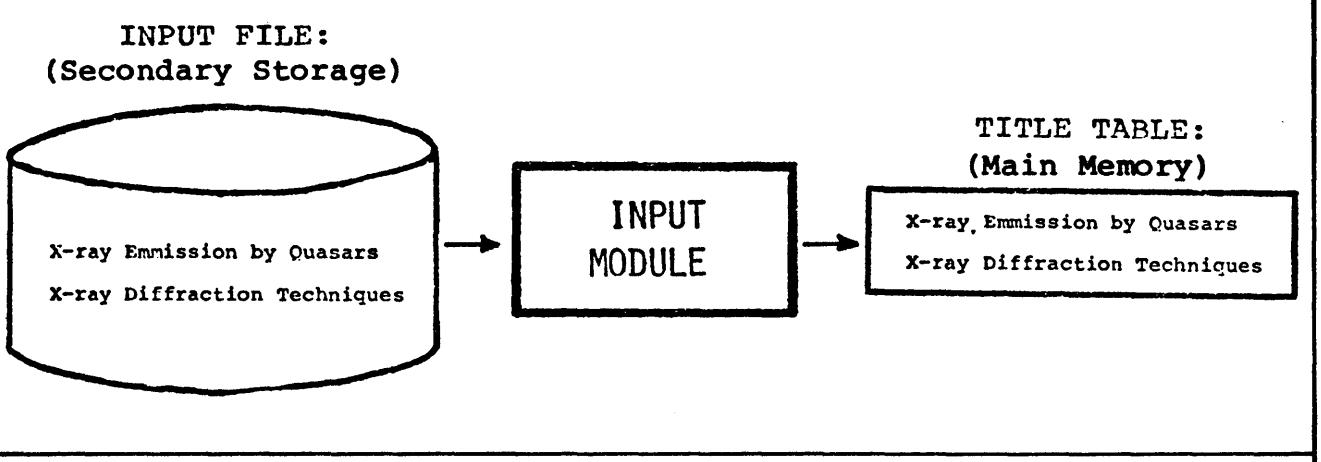


Figure 6.2 -- The conventional input module

Circular Shift Module. This module is called after the input module has completed its work. It takes each title that has been placed in the TITLE TABLE by the input module and computes all the circular shifts of that title. For example, the title:

X-ray Diffraction Techniques

becomes:

X-ray Diffraction Techniques.
Diffraction Techniques. X-ray
Techniques. X-ray Diffraction

One way to store this information would be to build an array containing all the circular shifts of the lines. But, this is expensive because it requires storing quite a few characters in memory. To conserve memory, the circular shifter builds a table with two columns. The first column gives the line number for the original title and the second column gives the character displacement within the line of the word that is first for that particular circular shift.

For example, the title "X-RAY DIFFRACTION TECHNIQUES" is the second title in the TITLE TABLE and its words start at the following displacements:

```
                  1     7           19
                  ↓     ↓           ↓
LINE #2:          X-ray Diffraction Techniques.
```

Thus, the table representing the circular shifts of this line is:

SHIFT TABLE		"TRANSLATION" OF SHIFT TABLE
LINE #	CHARACTER DISPLACEMENT	
2	1	X-ray Diffraction Techniques
2	7	Diffraction Techniques. X-ray
2	19	Techniques. X-ray Diffraction

Figure 6.3 summarizes the action of the circular shifter module. The titles in the TITLE TABLE are used as input to produce the SHIFT TABLE.

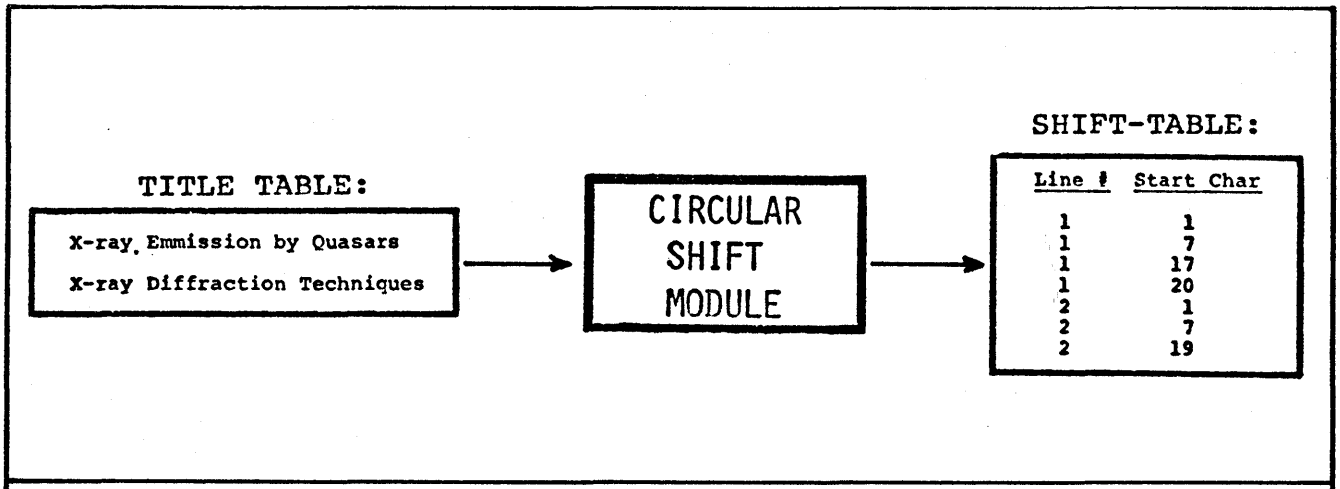


Figure 6.3 -- The conventional circular shift module

By using the SHIFT TABLE and the TITLE TABLE, all the possible circular shifts can now be listed:

X-ray Emmission By Quasars.
Emmission By Quasars. X-ray
By Quasars. X-ray Emmission
Quasars. X-ray Emmission By
X-ray Diffraction Techniques.
Diffraction Techniques. X-ray
Techniques. X-ray Diffraction

All that is left to do is to alphabetize the list of circular shifts and write them to the output file.

Alphabetizing Module. This module is called after the circular shifter completes its work. It takes as input the arrays produced by the input module and the circular shift module, i.e., the TITLE TABLE and the SHIFT TABLE. From this information, it produces an alphabetized list of all the circular shifts (a KWIC index). Once again, the module could store this information as an array of characters, but that would be inefficient. Instead, it produces an ALPHABETIZED SHIFT TABLE that is in the same format as the SHIFT TABLE produced by the circular shifter, but with the entries in alphabetical order. Figure 6.4 summarizes the action of this module.

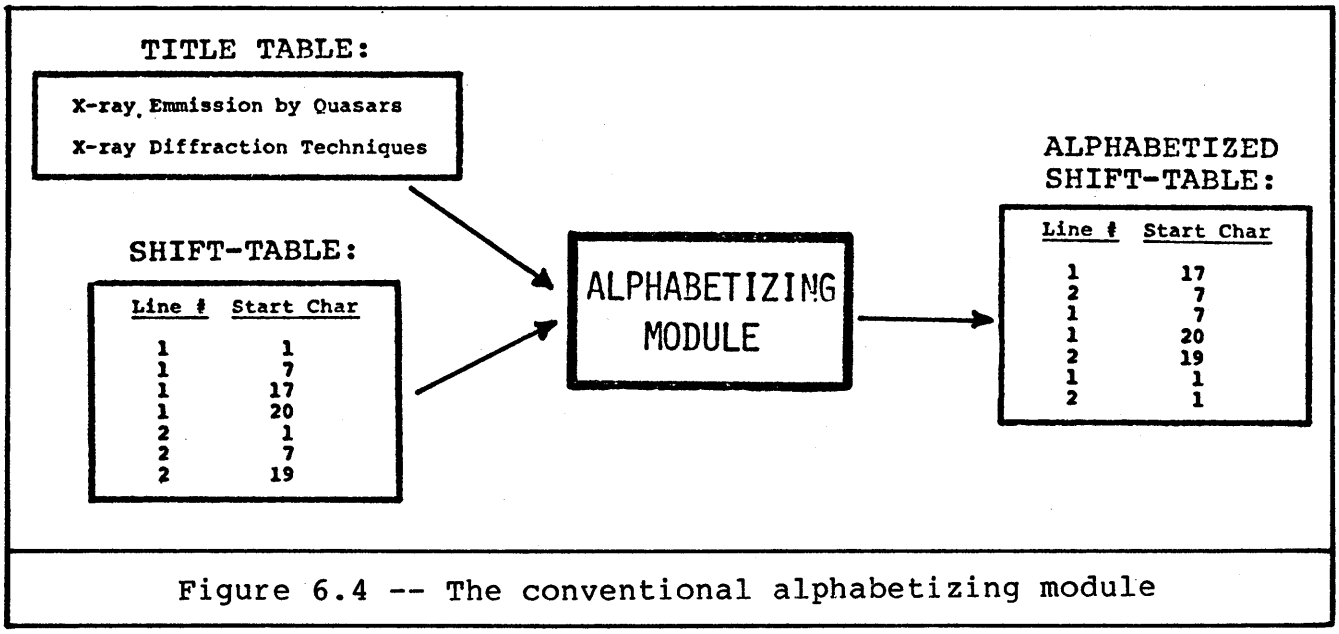


Figure 6.4 -- The conventional alphabetizing module

There is now an alphabetized list of circular shifts stored in an internal format as an ALPHABETIZED SHIFT TABLE and a TITLE TABLE. All that is needed to produce a KWIC index is to write this information to the output file in the desired format.

Output Module. This module is called when the alphabetizing module completes its work. It takes as input the ALPHABETIZED SHIFT TABLE and the TITLE TABLE and produces a nicely formulated output listing (KWIC index). Figure 6.5 summarizes the operation of this module.

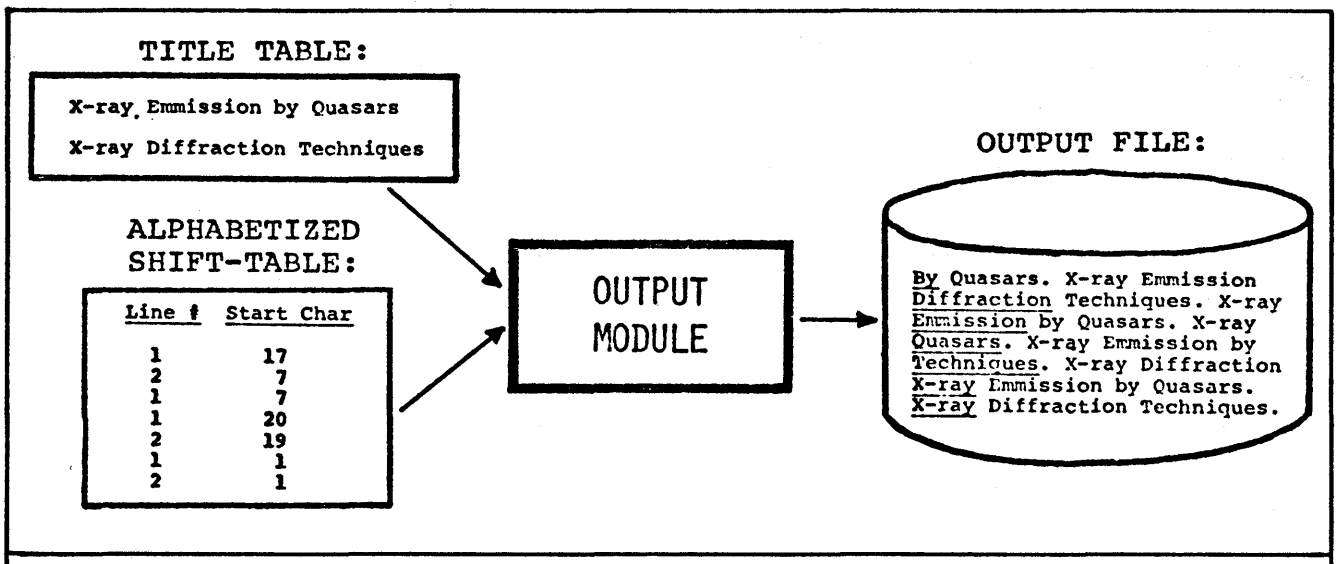


Figure 6.5 -- The conventional output module

All the modules needed to do actual manipulation of the data have been described, but one more module is needed to coordinate the first four.

Master Control Module. This module does little more than control the sequencing among the other four modules. It may also handle error messages, space allocation, etc.

Figure 6.6 summarizes the sequencing performed by the master control module. There are four steps:

1. The input module reads the input file and produces a TITLE TABLE.

2. The circular shift module produces a SHIFT TABLE from the TITLE TABLE.

3. The alphabetizing module produces an ALPHABETIZED SHIFT TABLE from the SHIFT TABLE and the TITLE TABLE.

4. The output module writes a KWIC index to the output file using the ALPHABETIZED SHIFT TABLE and the TITLE TABLE.

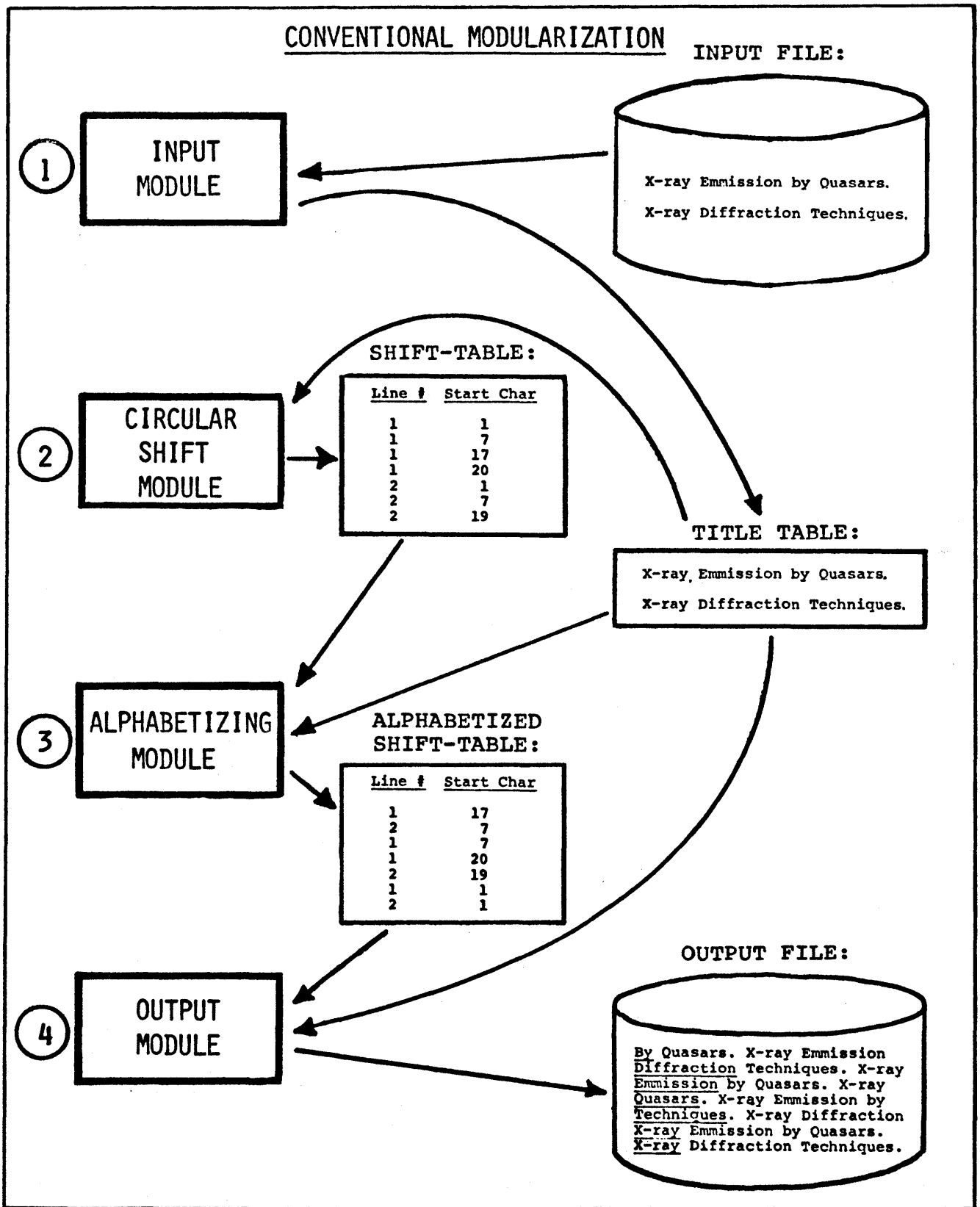


Figure 6.6 -- These modules directly manipulate shared data structures.

This concludes the description of the conventional modularization. Obviously, I have not presented all the details (e.g., core formats, pointer conventions) needed to begin programming, but there is enough information for comparison with the object-oriented approach.

This conventional approach is a modularization in the sense meant by all proponents of modular programming. The system is divided into a number of modules with well-defined interfaces; each one is small enough and simple enough to be thoroughly understood and well programmed. Experiments on a small scale indicate that this is approximately the decomposition which would be proposed by most programmers for the task specified.

Object-Oriented Modularization

The object-oriented modularization has several things in common with the conventional modularization. It uses the same data structures as the conventional modularization and it also uses the same algorithms. The difference between the two approaches is in how the program is broken up into work assignments, i.e., how it is modularized.

In the conventional decomposition the criterion used to modularize the program was to make each major processing step a module. One might say that to get the first decomposition, one makes a flowchart. This is the most common approach to decomposition or modularization. It is an outgrowth of programmer training which teaches us that we should begin with a rough flowchart and move from there to a detailed implementation. The flowchart was a useful abstraction for systems with on the order of 5,000-10,000 instructions, but as we move beyond that, it does not appear to be sufficient; something additional is needed.

The object-oriented decomposition uses the hiding of design decisions as the criterion for modularization. As you will see, each module in the object-oriented decomposition hides one design decision.

For example, in the conventional decomposition there are four modules (input, circular shift, alphabetizing, and output) that directly manipulate the title table. All of these modules are aware of the design decision to represent the title table by an array in main memory, i.e., their correct operation depends on the structure of the title table. In the object-oriented approach, there is only one module that directly manipulates the title table. This module hides the structure of the title table (i.e., the fact that it is an array in main memory and not on file on disk) from the other modules in the system. All the other modules are completely independent of the structure of the title table.

This approach continues the trend toward structured programming. The first major step by proponents of structured programming was to add structure to the control flow of a program. This effort involved replacing the "GO TO" of FORTRAN-like programming languages with more structured control statements such as the "WHILE DO" and CASE OF" statements in Pascal.

The second major step in structured programming is to structure the knowledge of how data is represented within a machine. This step is taken by the object-oriented approach to program decomposition. The knowledge of how data is represented (i.e., is it an array, or a linked list? Is it in main memory or on disk?) is confined to one module. All other modules are written to be independent of the way the data is structured.

In a moment you will see why the object-oriented approach is advantageous, but first the object-oriented modularization of the KWIC index program is given. The object-oriented modularization has six modules.

Title Storage Module. This module manages an object (a data structure) containing the original titles read in from secondary storage. This object is called the TITLE OBJECT. It is structured the same way as the title table in the conventional modularization, i.e., it is an array of characters with characters packed four to a word and an otherwise unused character used to indicate the end of a word. However, though it is structured like the title table, there are two major differences that makes it an object and not just a data structure.

1. It has a label that tells its type, i.e., "TITLE OBJECT".

2. It has a set of operations defined for it that are the only operations permitted on the object.

The title storage module provides a set of procedures available to other modules that use the TITLE OBJECT. These procedures are the only way that other modules can use the TITLE OBJECT. Other modules cannot directly access the TITLE OBJECT. As an example, consider the following procedures provided by the title storage module:

CHAR(t,w,c) - This procedure returns the cth character of the wth word of the tth title in the TITLE OBJECT.

SETCHAR (t,w,c,d) - This procedure sets the tth, wth, cth, character in the TITLE OBJECT to the value d.

WORDS (t) - This procedure returns the number of words in the tth title of the TITLE OBJECT.

DELINE (t) - This procedure deletes the tth title of the TITLE OBJECT.

DELWORD (t,w) - This procedure deletes the wth word in the tth title of the TITLE OBJECT.

The title storage module provides some other procedures, but I think you get the idea. Other modules that use the TITLE OBJECT do so by calling the procedures provided by the title storage module. They never manipulate the data structure directly, -- only through the procedures provided by the title storage module.

You can visualize the title storage module as a "black box" (see Figure 6.7) that hides the representation of the TITLE OBJECT (i.e., hides the fact that it is structured as a table in main memory instead of a linked list or a random access disk file). The buttons on the front of the box are the procedures provided by the title storage module for manipulating the TITLE OBJECT hidden inside the box. Other modules using the TITLE OBJECT call these procedures instead of manipulating the TITLE OBJECT themselves.

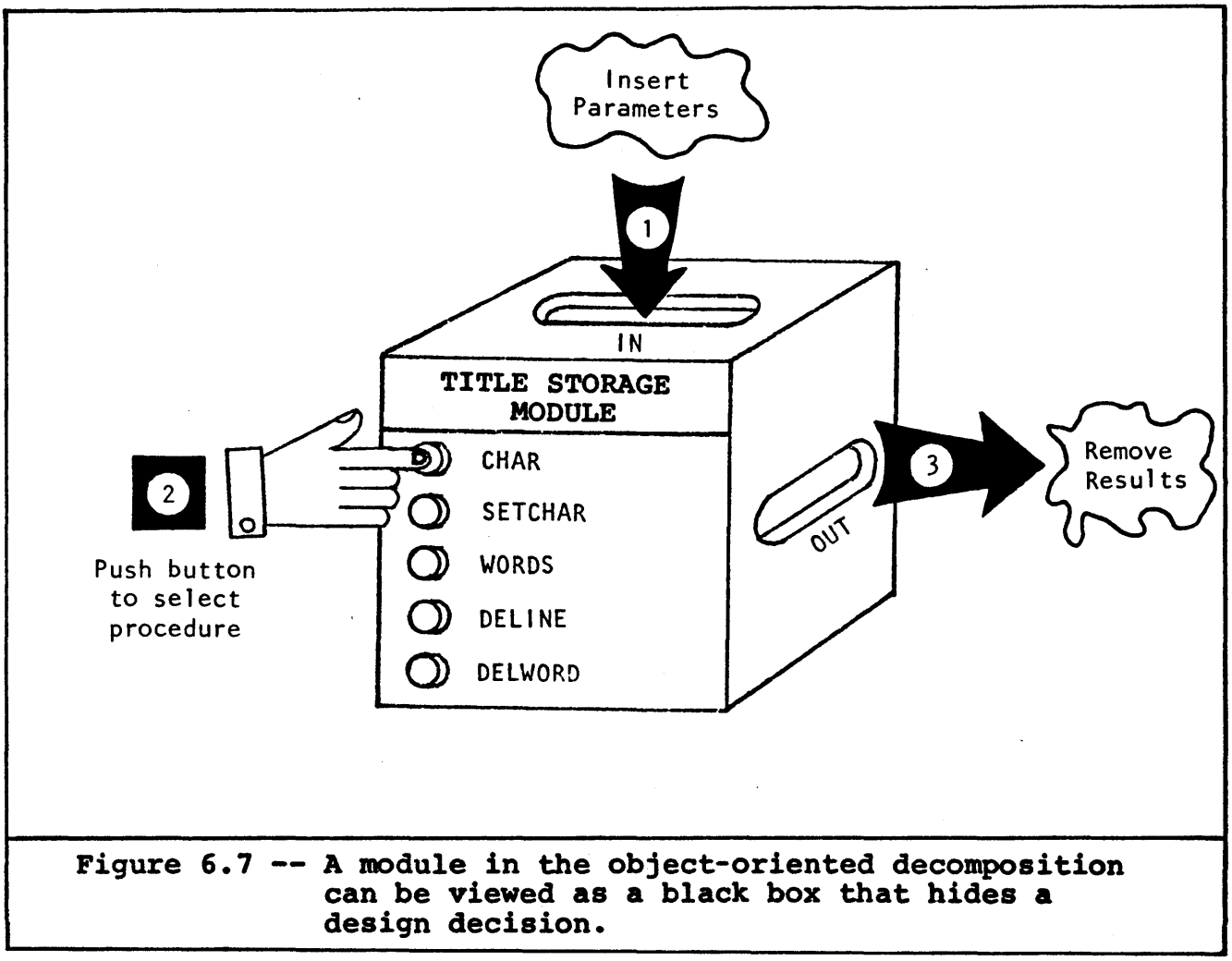


Figure 6.7 -- A module in the object-oriented decomposition can be viewed as a black box that hides a design decision.

Input Module. This module hides the input format. It reads the original lines from the input media and calls procedures provided by the title storage module to store the lines inside the title object.

Master Control Module. This module is similar to the master control module described for the conventional modularization. It controls the sequencing of the other modules.

As an example, consider the sequencing of the modules that have already been described. One of the first things the master control module does is call the input module. The input module then reads the input file and stores the titles in the TITLE OBJECT by calling the procedures (e.g., SETCHAR) provided by the title storage module. When the input module has finished, it returns to the master control module.

Now that the titles are stored in the title object, the next step is to produce all the circular shifts of the titles. To do that, a data structure is needed to contain the circular shifts of the titles. This means that another module is needed to hide the representation of this data structure, because the representation of each data structure is a design decision that needs to be confined within a module. This module is described before more of the sequence of execution is explained.

Circular Shifter Module. This module manages the SHIFTED TITLE OBJECT. The users of this module view the SHIFTED TITLE OBJECT as an array that contains all the circular shifts of the titles contained in the title object.

For example, if the TITLE OBJECT contains the titles:

X-ray Emmission By Quasars.
X-ray Diffraction Techniques.

then other modules using the SHIFTED TITLE OBJECT view it as containing all the circular shifts of these titles:

X-ray Emmission By Quasars.
Emmission By Quasars. X-ray
By Quasars. X-ray Emmission
Quasars. X-ray Emmission By
X-ray Diffraction Techniques.
Diffraction Techniques. X-ray
Techniques. X-ray Diffraction

The circular shifter module provides a set of procedures available to other modules that use the SHIFTED TITLE OBJECT. As with the title storage module, these procedures are the only way that other modules can use the SHIFTED TITLE OBJECT; they cannot manipulate it directly.

Here are a few examples of the procedures provided by the circular shifter module:

CSCHAR (t,w,c) - This procedure is very similar in function to the CHAR procedure in the title storage module. The only difference is that this procedure returns a character from the SHIFTED TITLE OBJECT while CHAR returns a character from the TITLE OBJECT.

CSSETCHAR(t,w,d) - The circular shifter module has a full complement of procedures for manipulating the SHIFTED TITLE OBJECT. They are all similar to the procedures provided by the title storage module except they manipulate the SHIFTED TITLE OBJECT instead of the TITLE OBJECT.
CSWORDS (t)
etc.

CSSETUP - Before the procedures for reading and manipulating the SHIFTED TITLE OBJECT can be used, some initialization needs to be done, i.e., all the circular shifts of the titles must be created. The CSSETUP procedure does this. It is called once, before any of the other procedures are used, and it initializes the object. I will explain this a bit more when I resume explaining the sequence of execution.

From the discussion so far, it would appear that the circular shifter maintains the SHIFTED TITLE OBJECT as an array of characters, because this is the view presented to other modules using the SHIFTED TITLE OBJECT. Yet, earlier, in the explanation of the conventional modularization, I explained that while it is possible to maintain this information as an array of characters, in practice it requires too much storage. The object-oriented modularization faces the same limit, and it uses the same solution -- storing the information as an array of line numbers and starting characters.

Internal to the circular shifter module, and hidden from all users of the module, is a shift table. It has the same structure as the shift table in the conventional modularization -- a column of line numbers and column of starting-character numbers.

Also internal to the circular shifter module is an object reference for the title storage module. This means that the circular shifter module can use the TITLE OBJECT by calling the procedures provided by the title storage module. The circular shifter module must be able to do this because it does not keep a copy of the actual titles itself -- it uses the TITLE OBJECT for this purpose. All the circular shifter module has is the shift table that describes all the possible shifts of the titles in the TITLE OBJECT.

Figure 6.8 illustrates the difference between the users abstract view of the object and the actual representation of the object. The user "sees" an array of characters because this is the view presented by the procedures in the circular shifter module. But, the information is actually represented by a shift table and a reference for the title storage module.

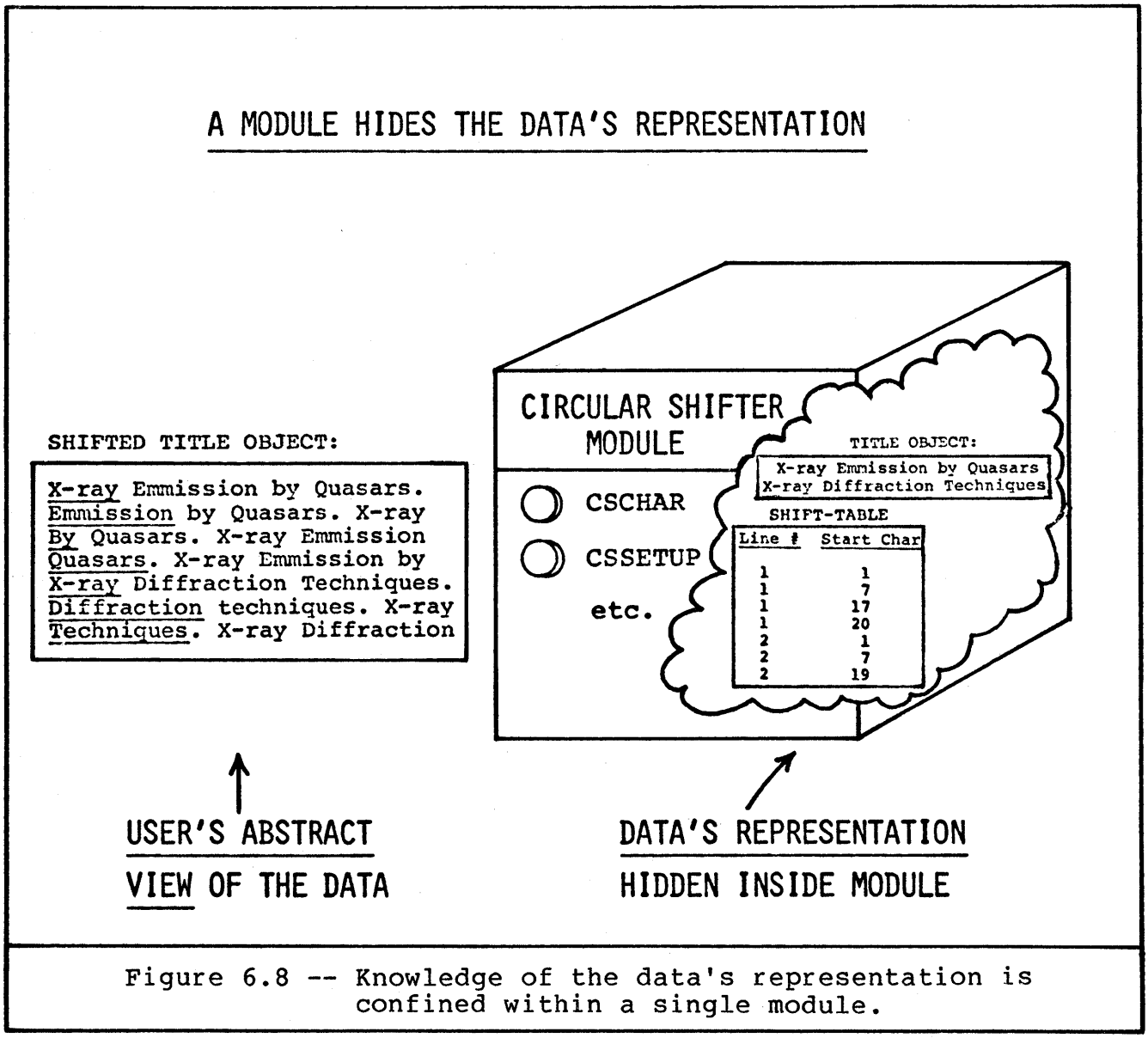


Figure 6.8 -- Knowledge of the data's representation is confined within a single module.

The circular shifter module effectively hides the representation of the SHIFTED TITLE OBJECT from other modules that use it. It does this by presenting the other modules with an abstract view of the object, i.e., to them it looks like an array of characters. But, internally, the SHIFTED TITLE OBJECT is represented by the shift table: a set of indices for the TITLE OBJECT.

Return for a moment to the sequence of program execution. When the sequence was left, the master control module had called the input module, and the input module had transferred the titles from the input file to the TITLE OBJECT. The input module never directly manipulated the TITLE OBJECT -- it only used the procedures provided by the title storage module.

Now that the input module has finished storing the titles and has returned to the master control module, the circular shifts can be created. To do this, the master control module calls the CSSETUP procedure in the circular shifter module. The CSSETUP procedure computes the circular shifts of the titles stored in the TITLE OBJECT (using only the procedures provided by the title storage module to manipulate the TITLE OBJECT) and stores the appropriate indices to the TITLE OBJECT in its shift table. When CSSETUP is finished initializing the shift table, it returns to the master control module.

Now that the shift table has been initialized, all the other procedures provided by the circular shifter can be used. These procedures create the view of working with an abstract data type called a SHIFTED TITLE OBJECT that looks like a character array of all the circular shifts of the titles. The user has this abstract view even though the SHIFTED TITLE OBJECT is actually represented by the shift table's indices and the TITLE OBJECT.

Now that all the circular shifts of the titles are available, all that is needed is to alphabetize them and write them to the output file. An alphabetized list of the circular shifts requires another data structure and thus another module to manage it.

Alphabetizer Module. This module manages the ALPHABETIZATION OBJECT, which provides the user with information about the alphabetical order of the circular shifts.

The alphabetizer module has two procedures:

- ALPH - This procedure is similar to the CSSETUP procedure in the circular shifter module. It is called to initialize the alphabetizer module. When it is called, it builds an alphabetized shift table which is used by the second procedure. This alphabetized shift table has the same structure as the alphabetized shift table in the conventional modularization, but it is constructed by using the circular shifter module to manipulate the SHIFTED TITLE OBJECT instead of directly manipulating the shift table like the conventional modularization.

- ITH(i) - This procedure allows the user to alphabetically access the circular shifts provided by the circular shifter module. It does this by returning the index t for the line in the SHIFTED TITLE OBJECT that is i^{th} in the alphabetical order. For example, if the call ITH(1) returns the index 3 it means that the 3^{rd} line in the SHIFTED TITLE OBJECT is the 1^{st} line in the alphabetical ordering of the SHIFTED TITLE OBJECT.

Figure 6.9 illustrates the "black box" view of the alphabetizer module.

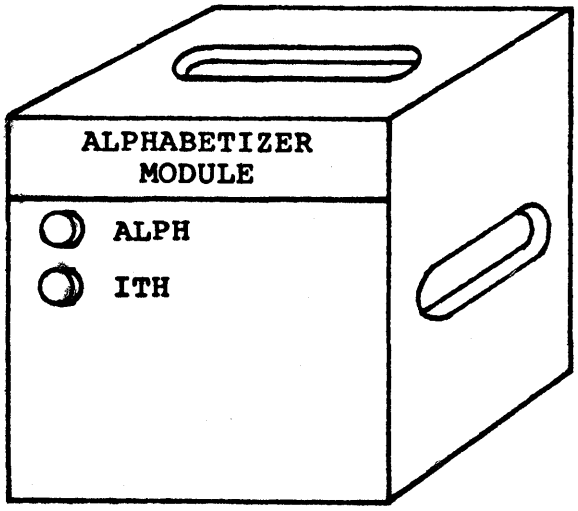


Figure 6.9 -- The alphabetizer module hides the alphabetization design decisions.

Output Module. This module creates the output file containing the KWIC index. It does this by using procedures provided by the alphabetizer module and the circular shifter module. It calls procedures in the circular shifter module to find out what the circular shifts are and it calls ITH(i) within the alphabetizer module to find out which circular shift comes first, which comes second, etc.

Note that this module hides the format of the output file from the other modules. If you want to change the format of the output, the changes necessary will be confined to the output module.

That completes the description of all of the modules in the object-oriented modularization of the KWIC index program. As a review, here is the sequence of program execution.

1) INPUT. The master control module calls the input module. The input module reads the input file and uses the procedures provided by the title storage module to store the titles in the TITLE OBJECT.

2) CIRCULAR SHIFT. The master control module calls the CSSETUP procedure in the circular shifter module which computes all the circular shifts of the titles. The circular shifter stores the description of the circular shifts in a compact internal format, but hides this internal representation from users of the SHIFTED TITLE OBJECT by providing a set of procedures that lets the users view the SHIFTED TITLE OBJECT as a character array. The CSSETUP procedure, as well as all of the other procedures in the circular shifter module, use the information stored in the TITLE OBJECT, but they do not access it directly. They use the procedures provided by the title storage module.

3) ALPHABETIZATION. After the circular shifts have been created, the master control module calls the ALPH procedure in the alphabetizer module which computes the alphabetical order of the circular shifts. During this computation, the ALPH procedure makes use of the information stored in the SHIFTED TITLE OBJECT by using the procedures provided by the circular shifter module -- it never manipulates the SHIFTED TITLE OBJECT directly. Once ALPH is finished, the procedure ITH can be used to determine the alphabetical order of the circular shifts.

4) OUTPUT. Now that the circular shifts are alphabetized, all that remains to be done is to print the KWIC index on the output file. The output module accomplishes this by calling procedures in the alphabetizer module and the circular shifter module. The output module calls the procedure ITH in the alphabetizer module to find out the order of the circular shifts, and then uses procedures in the circular shifter module to find out what the words are in that particular circular shift.

Figure 6.10 summarizes the object-oriented modularization. Each module is shown as a box (an object) with a set of object references for all the modules that it uses. The numbers shown (1-4) correspond to the sequence of calls by the master control module as described above.

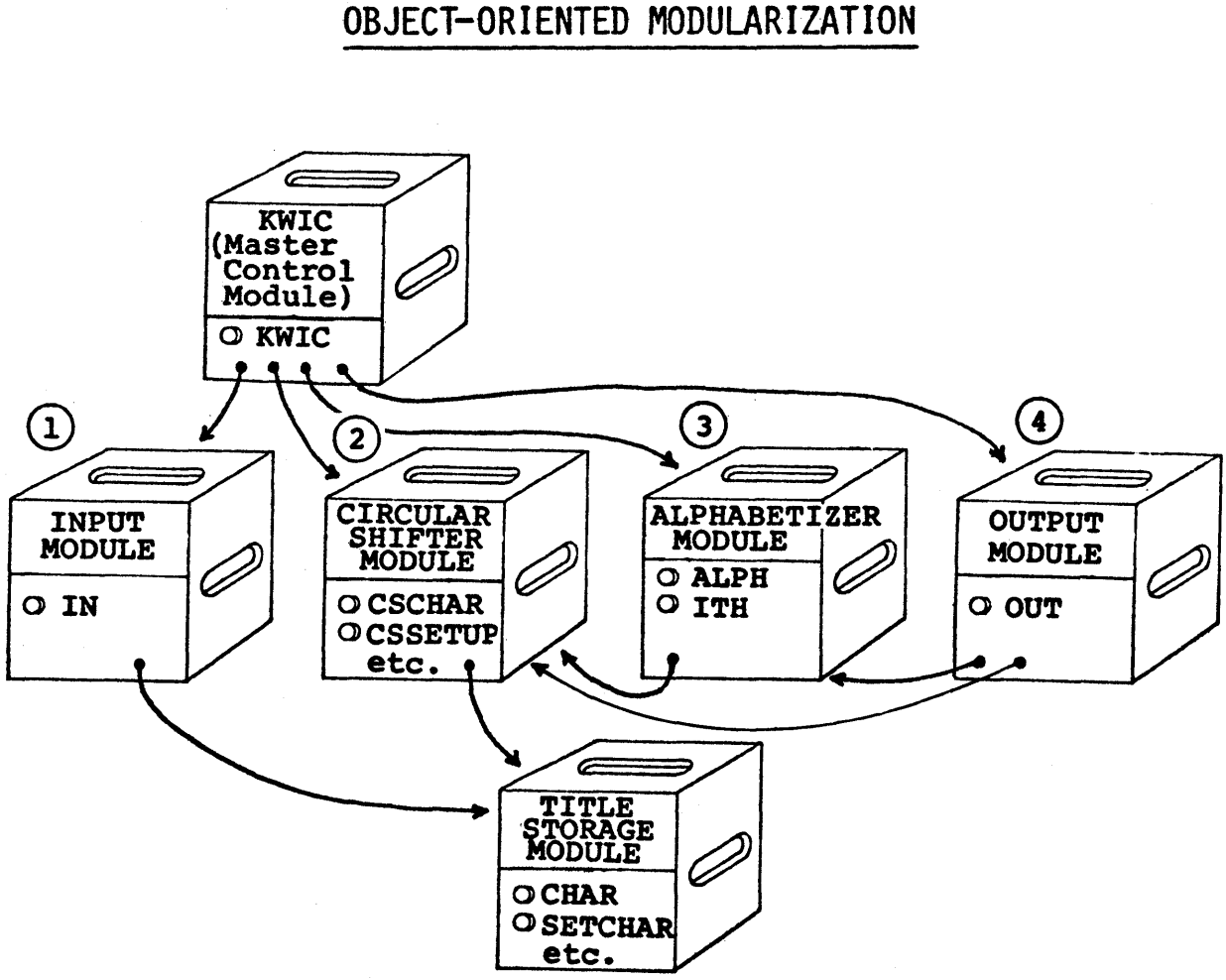


Figure 6.10 -- These modules share information by using services (procedures) provided by other modules instead of directly manipulating shared data structures.

As you may have guessed by now, the 432 has a hardware-recognized object type that represents a module. It's called a "domain object" and it is covered in the next section of this chapter.

For now, the important thing to notice is the difference between Figure 6.10 and Figure 6.6. In the conventional modularization (Figure 6.6), modules share information by directly manipulating shared data structures. In the object-oriented modularization, modules share information by using services (procedures) provided by other modules.

That concludes the description of the object-oriented modularization. Now it is compared with the conventional modularization.

Comparison of the Two Modularizations

General. The major difference between the two design methods is that the object-oriented approach makes a specific attempt to "hide" design decisions that are likely to change, while the conventional approach makes no such attempt.

One of the best metrics to measure the ability of a design to hide major design decisions is to count the number of modules that manipulate each data structure. Because data structures are likely to change when design decisions change, it is desirable to minimize the number of modules that directly manipulate each data structure. Minimizing the number of modules that need to "understand" how the data is structured minimizes the number of modules that need to be changed if the structure of the data changes.

Note the difference between the term manipulate and the term use. Manipulate means direct access to the physical data structure by a module. Manipulate means that a module is dependent upon the physical representation (e.g., a memory-resident table or a random access disk file) chosen. Use means indirect access to the data structure through procedures that hide its representation. Designing modules to use data structures instead of manipulate them means that the way the data is represented can change without changing the modules that use it.

Figures 6.11 and 6.12 illustrate how the two modularizations differ in the number of data structures directly manipulated by each module. Each arrow from a module points to a data structure that it manipulates.

In Figure 6.11, you can see that the conventional modularization distributes the knowledge of the data's representation among many modules. Most data structures are directly manipulated by several different modules.

In Figure 6.12, you can see that quite the opposite is true for the object-oriented decomposition. This method centralizes knowledge of the data's representation within a single module. Although each data structure may be used by many modules, only one module has the detailed knowledge of the data's structure required to manipulate it directly. This module provides the other modules with procedures for using the data structure. Because the other modules use the data by calling these procedures, changing the structure of the data only requires changing these procedures, not all the modules that use them.

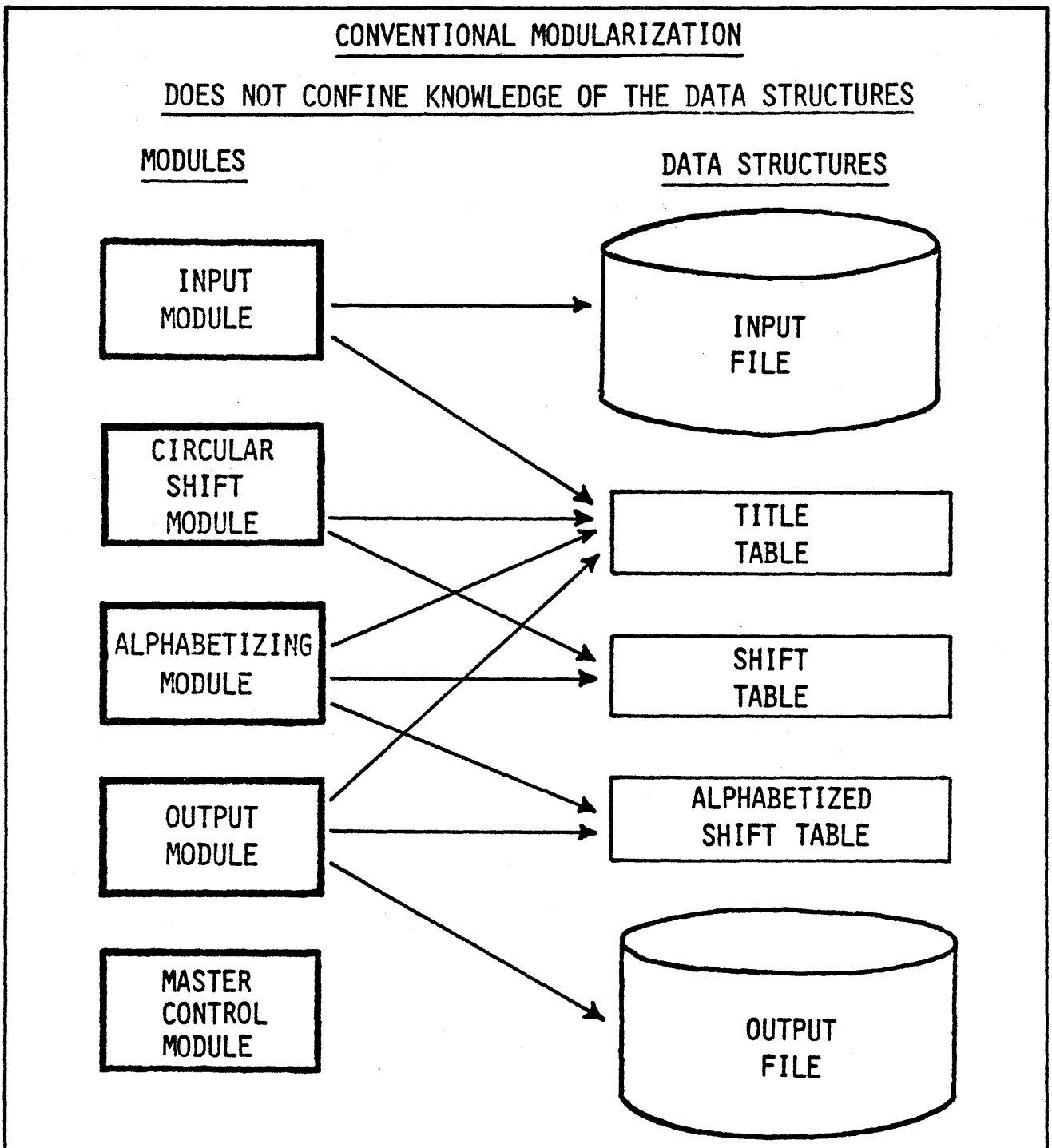


Figure 6.11 -- Conventional modularization distributes knowledge of the data structures throughout many modules.

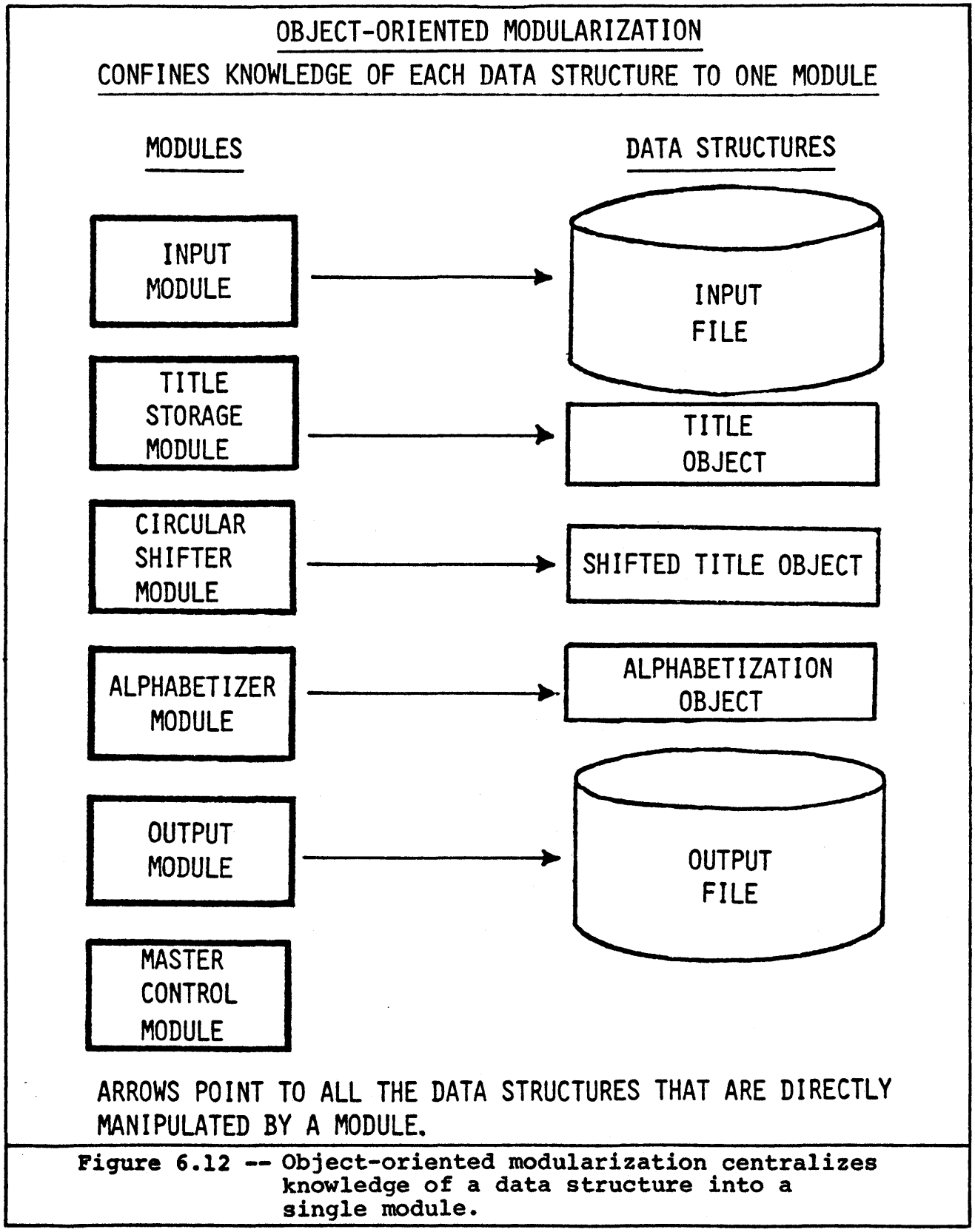


Figure 6.12 -- Object-oriented modularization centralizes knowledge of a data structure into a single module.

Now that you've seen the general differences between the two modularizations, specific differences are covered in the areas of:

- Changeability
- Independent development
- Comprehensibility

Changeability. There are a number of design decisions which are questionable and likely to change under many circumstances. Here is a partial list:

1. The decision to have all lines stored in core: For large jobs it may prove inconvenient or impractical to keep all of the lines in core at any one time.

2. The decision to pack the characters four to a word: In cases where we are working with small amounts of data, it may prove undesirable to pack the characters; time will be saved by a character per word layout. In other cases, we may pack, but in different formats.

3. The decision to make an index for the circular shifts rather than actually store them as such: Again, for a small index or a large core, writing them out may be the preferable approach. Alternatively, we may choose to prepare nothing during CSSETUP. All computation could be done during the calls on the other functions such as CSCHAR.

4. The decision to alphabetize the list once, rather than either (a) search for each item when needed, or (b) partially alphabetize: In a number of circumstances, it would be advantageous to distribute the computation involved in alphabetization over the time required to produce the index.

5. Input format.

By looking at these changes you can see the differences between the two modularizations. Each change is confined to a single module in the object-oriented modularization, but usually requires changes to several modules in the conventional modularization. Here are some specific examples:

Change 1 -- Title Storage

Changing the decision to keep all titles resident in memory effects every module except the master control module in the conventional modularization. All of them directly manipulate the title table and expect it to be resident at all times (see Figure 6.11).

In the object-oriented decomposition, this change is confined to the title storage module because it is the only module that directly manipulates the TITLE OBJECT (see Figure 6.12). All the other modules use the procedures provided by the title storage module to manipulate the TITLE OBJECT. They do not need to be changed as long as the revised title storage module continues to provide the same set of procedures.

Change 2 -- Character Packing

This change has an effect similar to the first change. Changing the number of characters packed to a word changes the representation of the title table. Therefore, all modules that depend upon the representation of the title table will need to be changed.

This involves changing every module in the conventional modularization, but only the title storage module needs to be changed in the object-oriented decomposition.

Change 3 -- Writing Out the Circular Shifts

Earlier, I explained that the decision had been made to represent the circular shifts as a set of indices to the original titles instead of storing the circular shifts as characters. If we decide to change this decision, and store the circular shifts as characters instead of indices to the original tables, we would have to change all the modules that directly manipulate the data structures used to represent the circular shifts. In the conventional modularization this means changing three modules: circular shift, alphabetizing, and output. In the object-oriented modularization, only the circular shifter needs to be changed.

Change 4 -- Delayed Alphabetization

In the conventional modularization, the alphabetizing module builds an alphabetized shift table that is also manipulated by the output module. The output module expects the alphabetized shift table to be complete when it is called, thus both the alphabetizing and the output module must be changed if we decide to delay alphabetization or to only partially alphabetize.

In the object-oriented modularization, the design decision of when to alphabetize the titles has been confined to the alphabetizer module. Not only has the alphabetizer module hidden the representation of the alphabetization object, it has also hidden the decision of when it is alphabetized. Thus, only the alphabetizer module needs to be changed in the object-oriented modularization.

Change 5 -- Input Format

This change is especially interesting because it is the only change that is confined to one module in both modularizations. If you return to Figures 6.11 and 6.12 for a moment you should be able to see why this is true. In both modularizations, the input file is directly manipulated by only one module. The difference between the two approaches is that the hiding happened by chance in the conventional modularization, but was planned in the object-oriented modularization.

This concludes the comparison of the two modularizations' support for changeability. This comparison is summarized in Figure 6.13.

CHANGE	MODULES CHANGED IN CONVENTIONAL MODULARIZATION	MODULES CHANGED IN OBJECT-ORIENTED MODULARIZATION
(1) TITLE STORAGE - Store only some of the titles in main memory. Swap to and from disk storage as needed.	• INPUT • CIRCULAR SHIFT • ALPHABETIZING • OUTPUT	• TITLE STORAGE
(2) CHARACTER PACKING - Store one character per word instead of packing four to a word.	• INPUT • CIRCULAR SHIFT • ALPHABETIZING • OUTPUT	• TITLE STORAGE
(3) WRITING OUT THE CIRCULAR SHIFTS - Store circular shifts as characters instead of making an index.	• CIRCULAR SHIFT • ALPHABETIZING • OUTPUT	• CIRCULAR SHIFTER
(4) DELAYED ALPHABETIZATION - Search for each item as needed, or partially alphabetize the list instead of alphabetizing the whole list at once.	• ALPHABETIZING • OUTPUT	• ALPHABETIZER
(5) INPUT FORMAT -	• INPUT	• INPUT
COMMENTS -	Most changes involved more than one module.	All changes were confined to one module.

Figure 6.13 -- The object-oriented approach confines changes to fewer modules.

Independent Development. In the conventional modularization, the interfaces between modules are the fairly complex formats and table organizations described above. These represent design decisions which cannot be taken lightly. The table structure and organization are essential to the efficiency of the various modules and must be designed carefully. The development of the formats is a major part of module development, and that part must be a joint effort among the several development groups.

In the object-oriented modularization, the interfaces are more abstract; they consist primarily in the function names and the numbers and types of the parameters. These are relatively simple decisions and the independent development of modules should begin much earlier.

Comprehensibility. To understand the output module in the conventional modularization, it is necessary to understand something of the alphabetizer, the circular shifter, and the input module. There are constraints on the tables used by the output module that only make sense because of the way that the other modules work. There will be constraints on the structure of the tables due to the algorithms used in the other modules. The system will only be comprehensible as a whole. This is not true in the second, object-oriented modularization.

SUMMARY

The important difference between the two methodologies is:

THE CRITERIA USED FOR DECOMPOSITION

- Conventional -- Make each major step in the processing a module.
- Object-Oriented -- Hide each design decision within a single module.

Hiding the representation increases the independence of each module. If the modules that use the title storage module do not know how it represents the TITLE OBJECT, then they cannot be written to depend upon the representation of the TITLE OBJECT.

It is this independence that gives modular programming its benefits of flexibility, comprehensibility, and shorter development time: flexibility, because the representation of an object can be changed by changing only one module -- the module that hides the representation; comprehensibility, because a programmer only needs to understand one module at a time; shorter development time, because each module is a responsibility assignment and can be developed totally independently of the other modules once the interface has been defined.

WHAT'S NEXT?

This section has compared two design methods and explained the advantages of object-oriented design, but has not related this to the architecture of the 432 micromainframe.

The next two sections of this chapter explain how the 432 architecture supports the object-oriented design method by providing two facilities: domains and type checking.

2. DOMAINS

This section describes domains -- one of the 432 architectural facilities that supports the object-oriented design method. The 3 subsections are:

- DOMAIN OBJECTS

  What a domain object is and how it fits into the basic program structure described in Chapter 3

- NETWORKS OF DOMAINS

  How domains are used to construct the static structure of a program

- CALLING A PROCEDURE IN A DIFFERENT DOMAIN

  An example explaining the change in access environments that occurs when a procedure calls another procedure that is in a different domain

DOMAIN OBJECTS

The 432 architecture supports the object-oriented design method by providing a hardware-recognized object type (called domain) that represents a module. These objects are called domains because the modules they represent confine knowledge of the object structure within a specified "domain". Figure 6.14 shows the two symbols used to represent domains in this book.

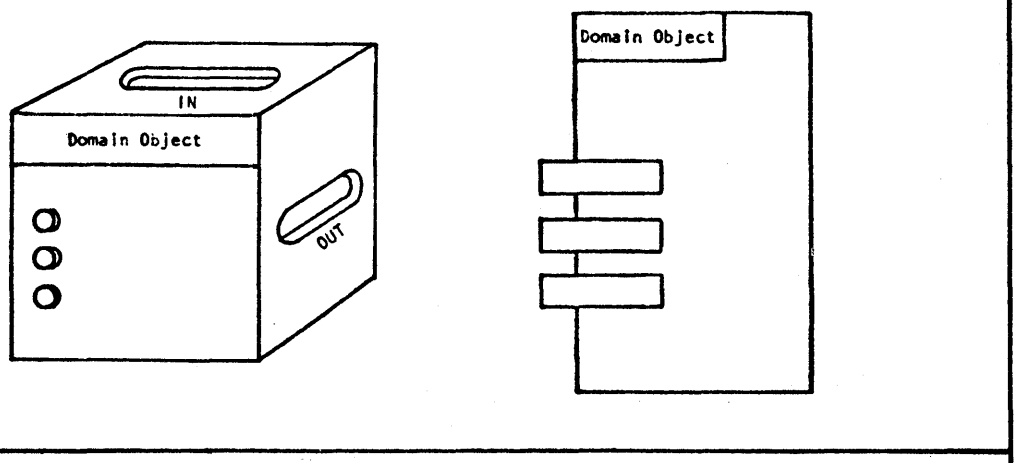


Figure 6.14 -- The 2 symbols for a domain object

Structurally, a domain is a very simple object. It is just a list of object references for all the static objects in the module. These static objects include:

- The instruction objects for the procedures in the domain
- The object being managed (e.g., the SHIFTED TITLE OBJECT in the CIRCULAR SHIFTER domain)
- Any other static objects containing information used by the procedures (e.g., constants, usage statistics)

By "static", I mean information that needs to persist between calls to a procedure. As an example, consider a sine routine that computes the sine of a number by interpolating a sine table. The sine table is static -- we do not want to throw it away when the procedure returns -- we want to save it to use the next time the procedure is called.

In contrast, some variables are dynamic. The storage used for them can be recycled (deallocated) when the procedure is exited. When the procedure is called again, storage for the dynamic variables is simply reallocated. Dynamic variables are things like loop counters that need to be reinitialized anyway.

A domain is a list of object references for all the static objects used by a module (not just one procedure -- the whole module). A context object (described in Chapter 3) is also a list of object references, but it is for all the dynamic objects used by a procedure (not the whole module -- just one procedure).

Figure 6.15 summarizes the basic program structure described in Chapter 3.

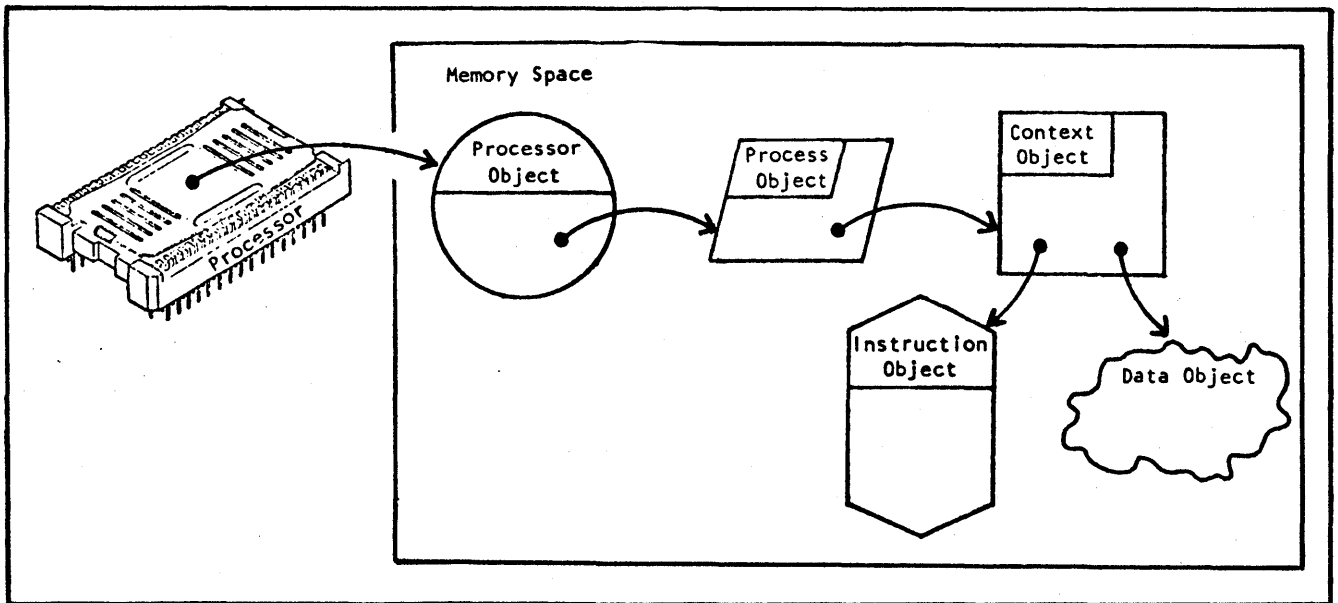


Figure 6.15 -- The basic structure of a program

Figure 6.16 shows the more detailed program structure. The domain object and the static objects it references are (usually) created at compile-time. Contexts and the dynamic objects they reference are (usually) created at run-time.

When a program calls a procedure, the 432 CALL CONTEXT instruction creates a context for the procedure, then calls the procedure. When the procedure returns, the context object and all the dynamic objects it references are deallocated and their storage reclaimed for further use. The domain object and all the static objects it references are retained.

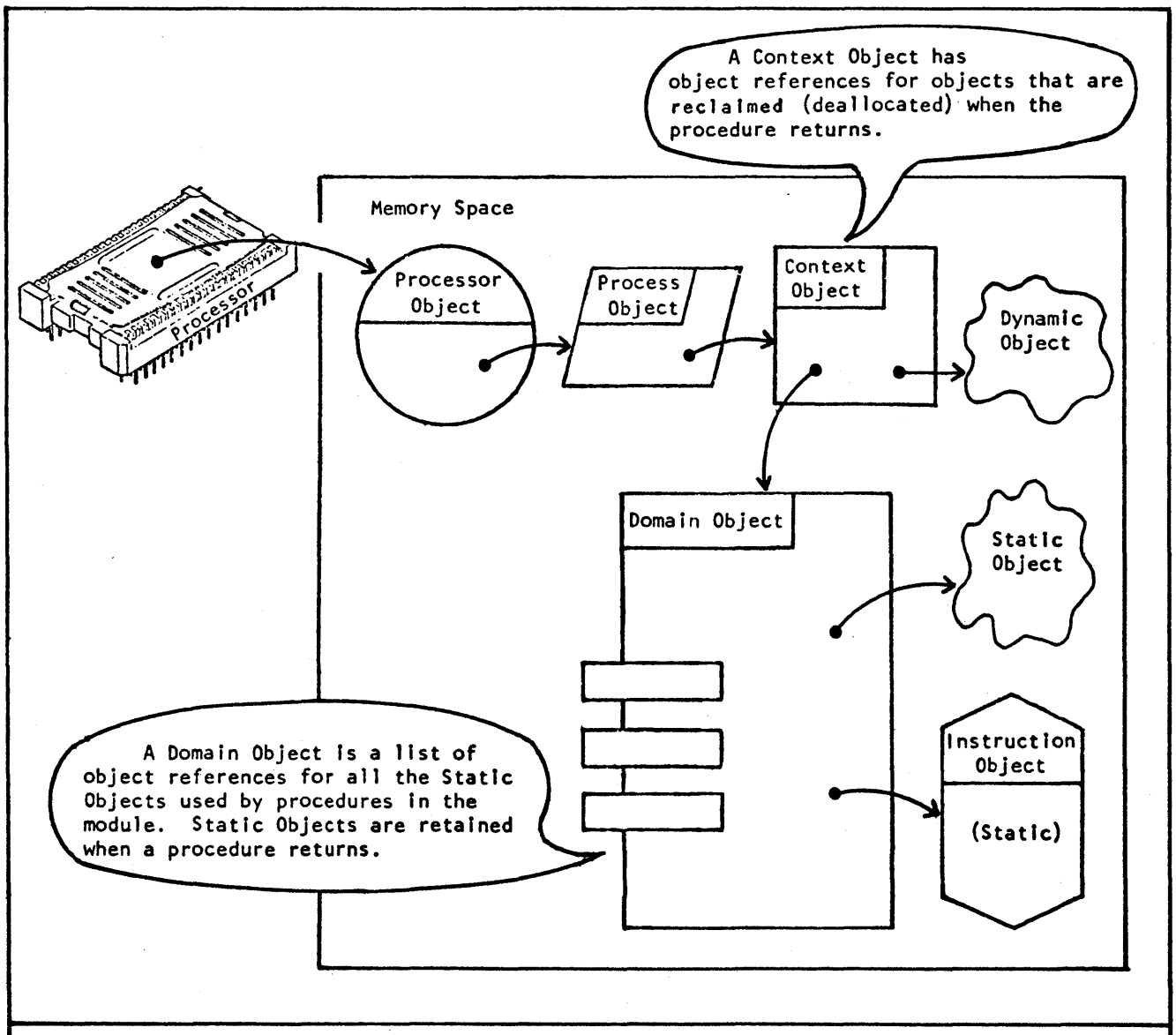


Figure 6.16 -- Domains are packages for the static objects used by a module.

NETWORKS OF DOMAINS

The static structure of a 432 program is represented by a network of domains -- a network of modules. The KWIC INDEX example is used to show how this works.

Each module in Figure 6.17 is represented by a domain, and each domain hides the representation of an object. For example, the TITLE STORAGE domain hides the representation of the TITLE OBJECT, and the CIRCULAR SHIFTER domain hides the representation of the SHIFTED TITLE OBJECT.

Some of the domains make use of objects in other domains to create a more complex object. The CIRCULAR SHIFTER, for instance, uses the TITLE STORAGE domain's TITLE OBJECT to create the more complex SHIFTED TITLE OBJECT.

If a module uses the procedures provided by a second module, then it needs to have a reference for the domain that represents the second module. Figure 6.17 shows the network of domains that forms the static structure of the KWIC index program. Each domain has object references for the other domains it needs to use. Once again, the 432's "need to know" addressing mechanism is used. A module can only use a procedure if it has a reference for the domain that contains the procedure.

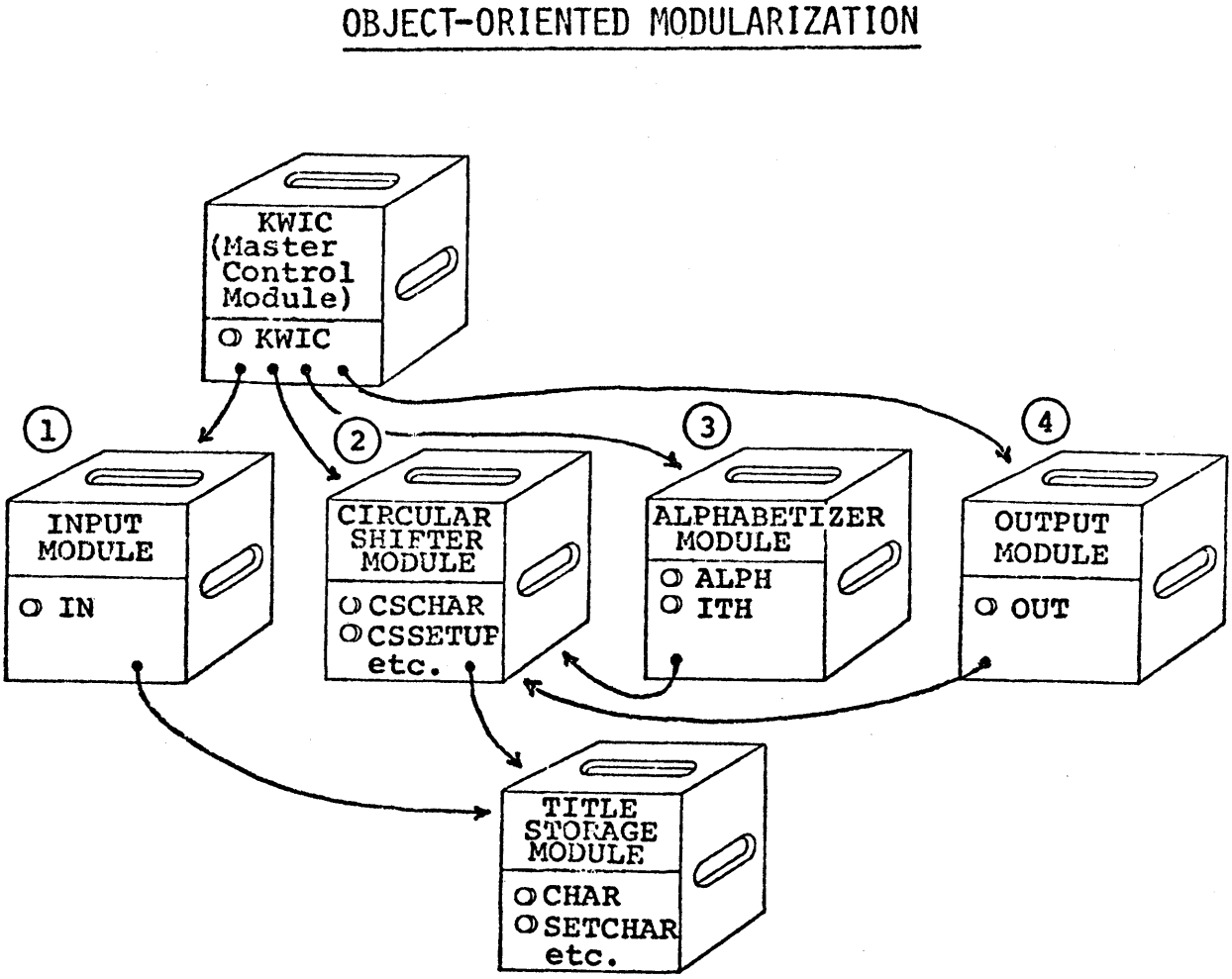


Figure 6.17 -- These modules share information by using services (procedures) provided by other modules instead of directly manipulating shared data structures.

For example, the INPUT domain has an object reference for the TITLE STORAGE domain because it uses the TITLE OBJECT contained by the TITLE STORAGE domain to store the lines it reads from the input device. The output module has a reference for both the ALPHABETIZER and the CIRCULAR SHIFTER module because it uses both. It calls the ALPHABETIZER to determine the order of the lines, and it calls the CIRCULAR SHIFTER to find out what each line contains.

Figure 6.17 is also a good illustration of how domains hide the representation of an object. Each domain is drawn as a "black-box" with buttons on the front to select the desired procedure. This is how domains are viewed by other domains that use them. The INPUT domain uses the TITLE STORAGE domain, it depends upon the TITLE STORAGE domain to provide the procedures CHAR, SETCHAR, etc., but it does not care how the TITLE OBJECT is represented or what algorithms are used to manipulate it.

The domains in Figure 6.17 are viewed from the "outside", i.e., from the perspective of the user. Domains can also be viewed from the "inside", i.e., from the perspective of a procedure executing inside of its own domain. Figure 6.18 shows a black box that has been slid open. This symbol is used when viewing a domain from the "inside".

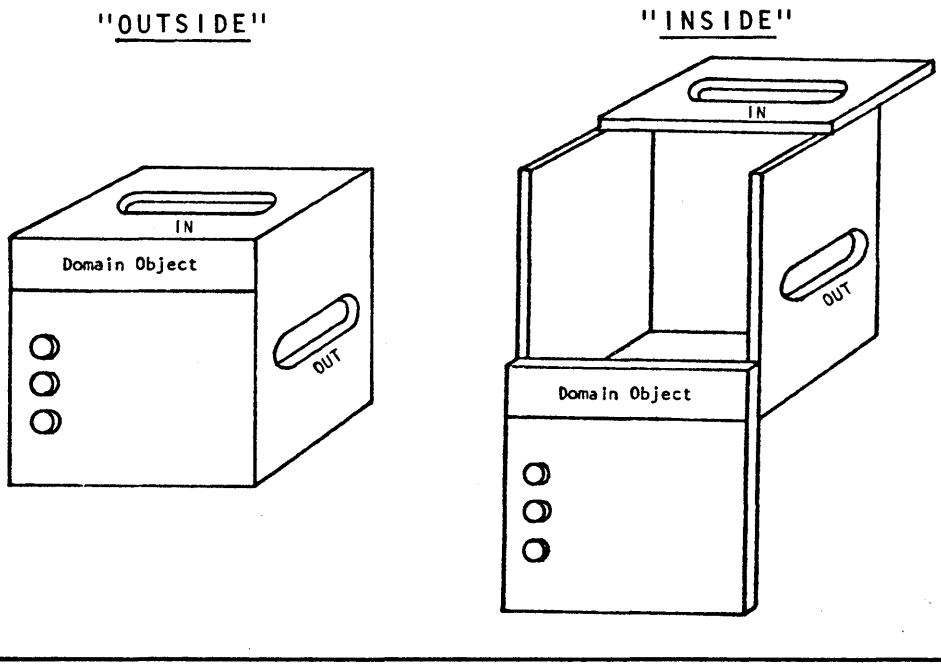


Figure 6.18 -- Two ways to view a domain

A procedure that is executing inside a domain needs to be able to access some objects that we do not want procedures outside of the domain to be able to access. For example, consider the TITLE OBJECT inside the TITLE STORAGE domain. Procedures that are inside the domain (e.g. CHAR, SETCHAR, etc.) must be able to access the TITLE OBJECT because these procedures directly manipulate the object. But, procedures outside of the domain (e.g., procedures like CSSETUP or ALPH that are inside other domains) should not be able to directly access the object, because these procedures may only manipulate the TITLE OBJECT by calling procedures in the TITLE STORAGE module.

On the other hand, there are some objects referenced by the domain that procedures outside of the domain must be able to access. These objects include the instruction objects for the procedures that are available to users of the domain (e.g., users of the line storage domain can call the procedures CHAR, SETCHAR, etc.).

To both provide access (to services) and prevent access (to internals), a domain's list of object references is divided into two parts: a public part and a private part (see Figure 6.19). Objects in the private part of the list can only be accessed by procedures inside the domain. Objects in the public part of the domain can be accessed by procedures inside the domain plus all procedures outside the domain that have a reference for the domain.

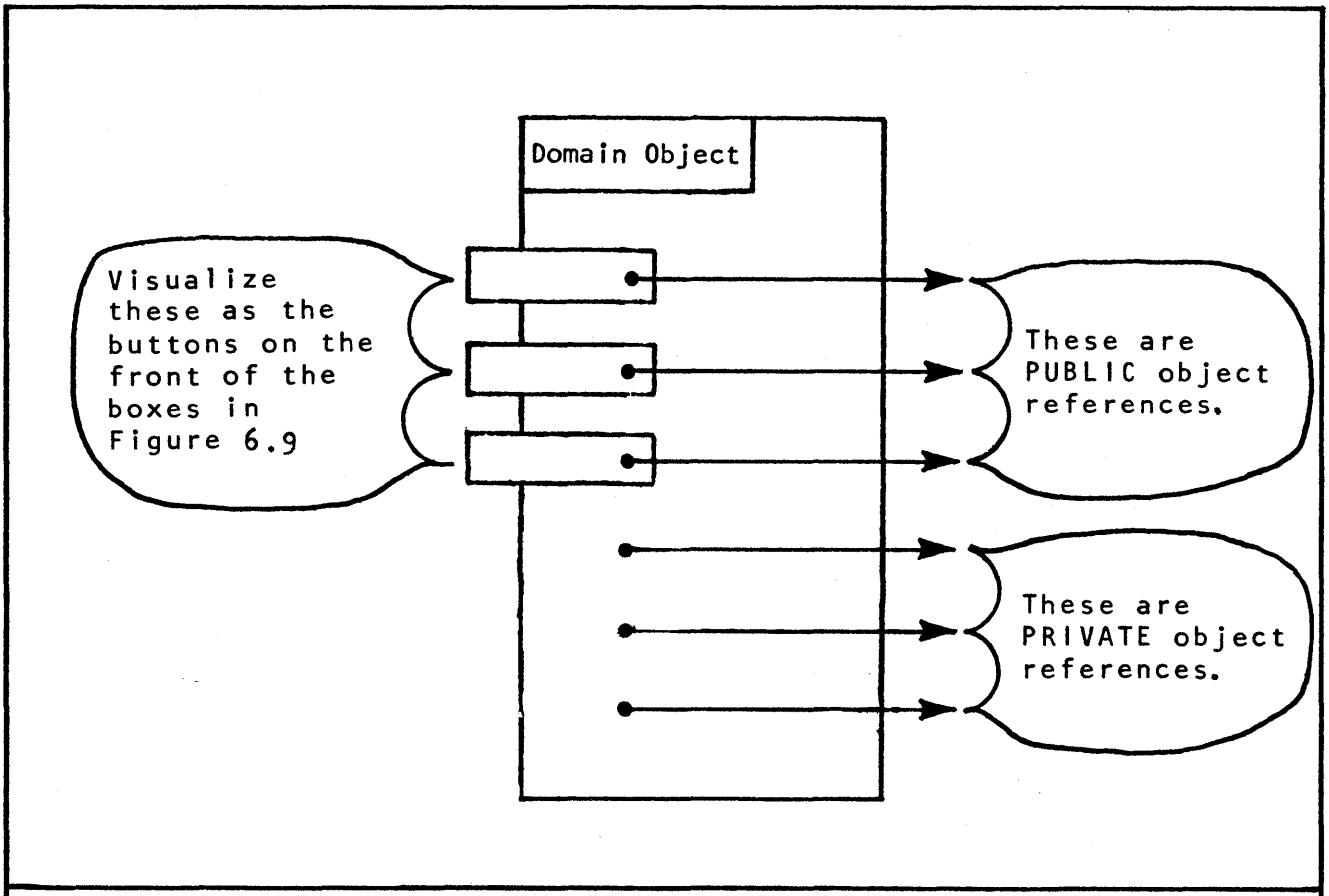
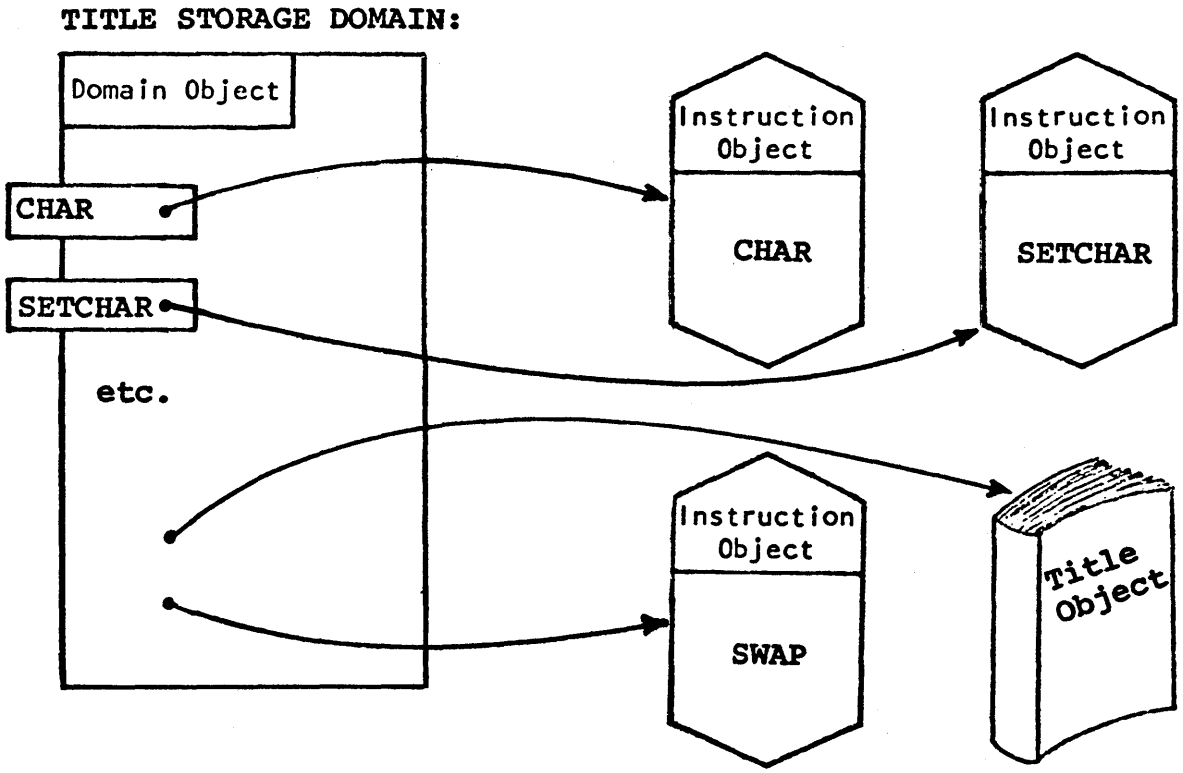


Figure 6.19 -- A domain is a list of object references for all objects in a module.

As an example, look at the domain object for the TITLE STORAGE module defined earlier. This domain (see Figure 6.20) has a public reference for each of the procedures available to other modules using the TITLE STORAGE MODULE, and a private object reference for the TITLE OBJECT. The procedures are available for use by the other modules, but the TITLE OBJECT is hidden.

In this example, the particular representation chosen for the TITLE OBJECT requires a SWAP procedure. SWAP is used by all of the public procedures to move parts of the object to and from secondary storage. This procedure is part of the module, but it should be hidden from the users of the module; therefore its reference is placed in the private part of the domain object.



Note: **Although the title storage module has several other public procedures (e.g., WORDS, DELINE), the diagrams in this section, for simplicity, only show two public references.**

Figure 6.20 -- The title storage domain

CALLING A PROCEDURE IN A DIFFERENT DOMAIN

Here is an example of what happens when a procedure in one domain calls another procedure in a different domain. Pay particular attention to the complete change in access environments that occurs when leaving one domain and entering another.

The example comes from the KWIC index program. At Time Period 1 in Figure 6.21.1A, the CSCHAR procedure within the CIRCULAR SHIFTER DOMAIN is executing. 1CSCHAR is the name I have given to the context object for this instance of the CSCHAR procedure. You can tell that this is the current context being executed by this process because the process object always has a reference for the current context being executed and that reference is pointing to 1CSCHAR.

In Figure 6.21.1A, the CIRCULAR SHIFTER domain has public object references for the CSCHAR and CSSETUP instruction objects. This makes sense because these procedures are publicly available to all programs that can reference the domain. The CIRCULAR SHIFTER DOMAIN also has private object references for the SHIFTED TITLE OBJECT and the TITLE STORAGE DOMAIN OBJECT. Both of these objects are hidden from procedures outside of the CIRCULAR SHIFTER DOMAIN, but are available for use by all the procedures in the domain (including CSCHAR).

Note that because the CIRCULAR SHIFTER DOMAIN has a reference for the TITLE STORAGE DOMAIN, all of the procedures in the CIRCULAR SHIFTER DOMAIN can access all the objects referenced by the public part of the TITLE STORAGE DOMAIN. These public objects include the procedures that the circular shifter will use to manipulate the TITLE OBJECT hidden in the private part of the TITLE STORAGE DOMAIN.

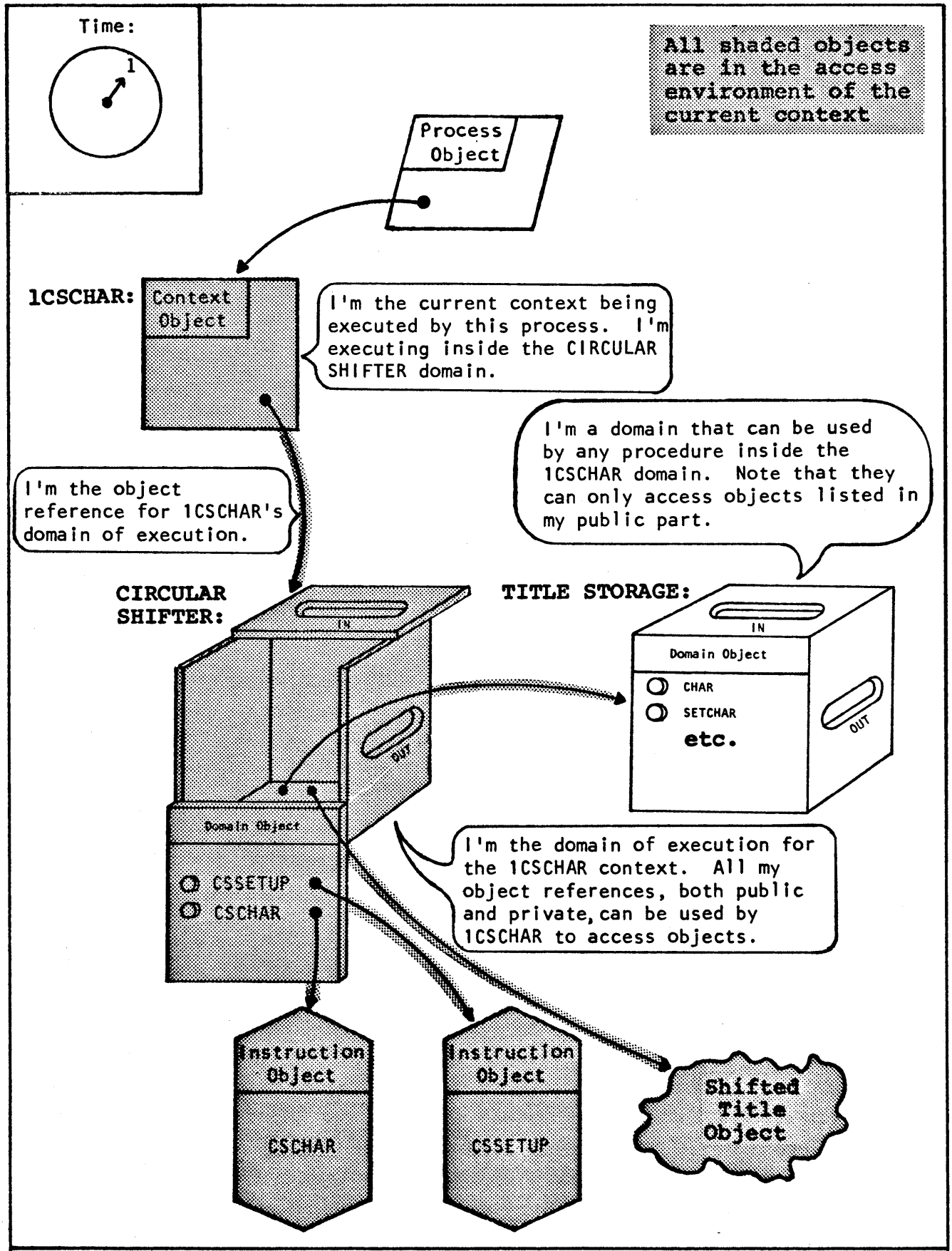


Figure 6.21.1A

Figure 6.21.1B shows the objects referenced by the TITLE STORAGE DOMAIN, both public and private. The references for the CHAR, SETCHAR, etc. procedures are in the public part, and can be accessed from inside the CIRCULAR SHIFTER DOMAIN. The references for the TITLE OBJECT and the SWAP procedure are in the private part, and can only be accessed from a procedure inside the TITLE STORAGE domain. How does the CSCHAR procedure call the CHAR procedure, which is in a different domain?

The first thing to do is create a context for the CSCHAR procedure. A context object is needed to hold the references for the dynamic objects, and to contain system information like the return link to the calling procedure. A context for a procedure is created by executing a CALL CONTEXT instruction. This instruction both creates and calls a context. It allocates the storage for the context object, initializes the system information it contains, and then transfers control to the newly-created context.

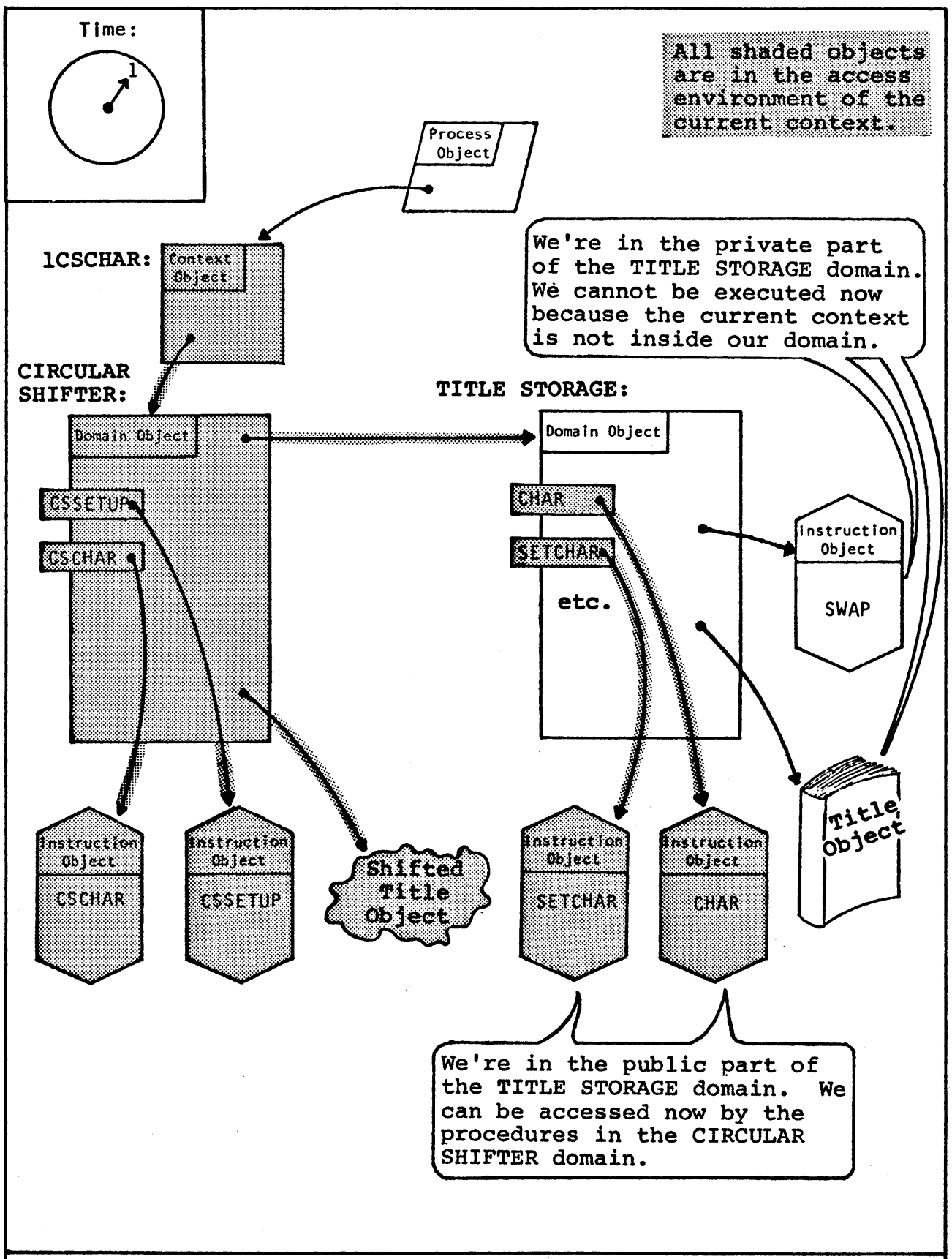


Figure 6.21.1B

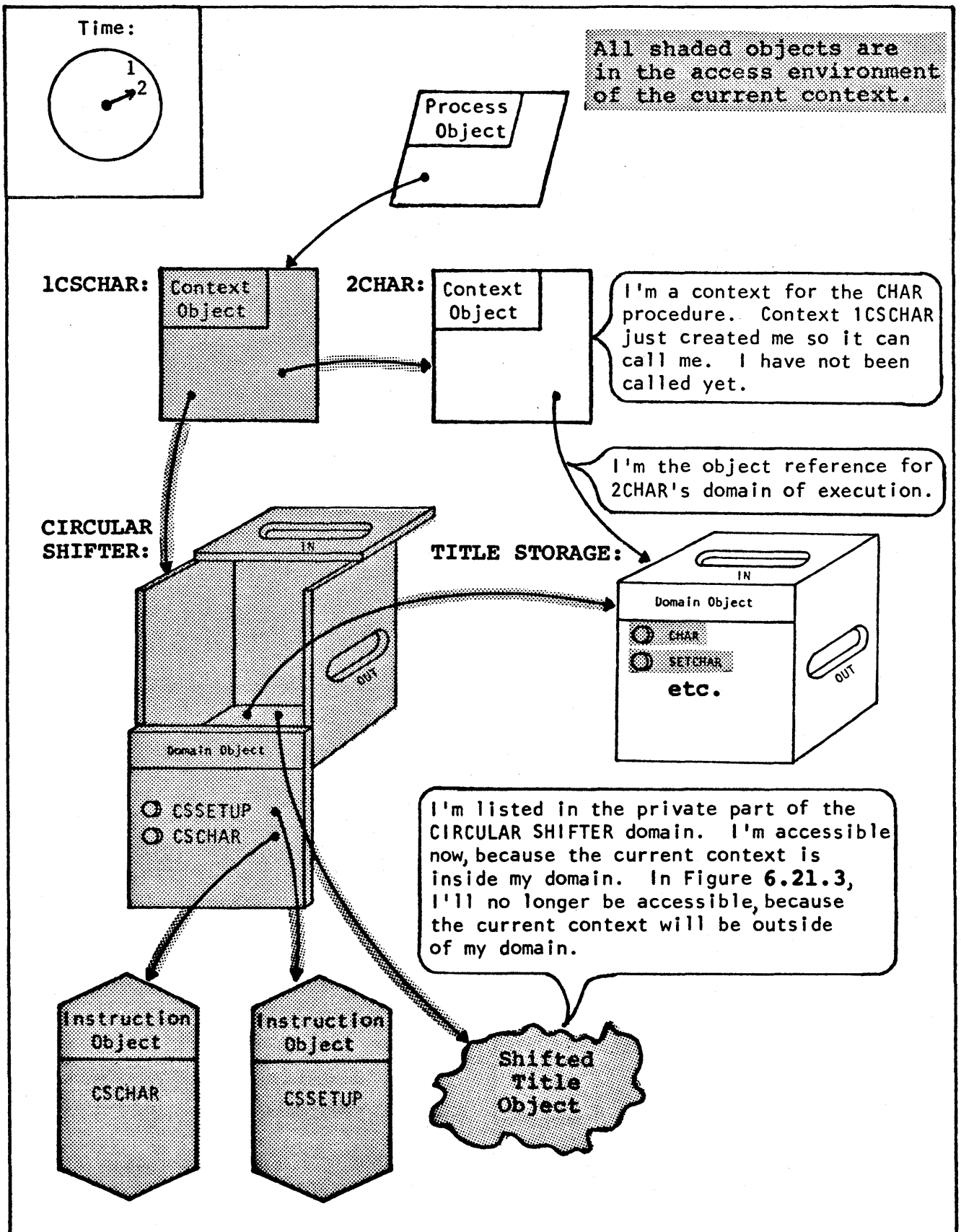

Time:
1
2
All shaded objects are in the access environment of the current context.
Process Object
1CSCHAR:
Context Object
2CHAR:
Context Object
I'm a context for the CHAR procedure. Context 1CSCHAR just created me so it can call me. I have not been called yet.
I'm the object reference for 2CHAR's domain of execution.
CIRCULAR SHIFTER:
IN
OUT
Domain Object
CSSETUP
CSCHAR
TITLE STORAGE:
IN
Domain Object
CHAR
SETCHAR
etc.
OUT
I'm listed in the private part of the CIRCULAR SHIFTER domain. I'm accessible now, because the current context is inside my domain. In Figure 6.21.3, I'll no longer be accessible, because the current context will be outside of my domain.
Instruction Object
CSCHAR
Instruction Object
CSSETUP
Shifted Title Object

Figure 6.21.2

In Figure 6.21.2, the CALL CONTEXT instruction has been partially executed. The context object 2CHAR has been created for the CHAR procedure, but it has not been called yet.

Once the context has been created, the CALL CONTEXT instruction finishes its execution by transferring control to the newly-created context (see Figure 6.21.3). Note that the CALL CONTEXT instruction creates a return link from the called context (2CHAR) to the calling context (1CSCHAR), and also updates the object reference in the process object to point to 2CHAR since it is now the current context.

The CALL CONTEXT instruction changes the access environment. Note the substantial difference between the objects accessible to the CSCHAR procedure in Figure 6.21.2 before the call and the objects accessible to the 2CHAR procedure in Figure 6.21.3 after the call.

Before the call, all the objects listed in the public and private parts of the CIRCULAR SHIFTER DOMAIN could be accessed, because a procedure inside the CIRCULAR SHIFTER DOMAIN was executing. Also, all the objects in the public (but not the private) part of the TITLE STORAGE DOMAIN could be accessed because the CIRCULAR SHIFTER DOMAIN had an object reference for it.

After the call, the new context is outside of the CIRCULAR SHIFTER DOMAIN. It cannot access any of the objects listed in the CIRCULAR SHIFTER DOMAIN, not even the objects in the public part of the domain. There is not even an object reference for the CIRCULAR SHIFTER DOMAIN, so the new context cannot access <u>any</u> of the objects listed in it.

After the call, the new context is not only outside of the CIRCULAR SHIFTER DOMAIN, it is inside the TITLE STORAGE domain. This means that it can now access all the objects listed in both the public and private parts of the TITLE STORAGE DOMAIN.

This ends the example. This section ends with a summary of what has been said about domains.

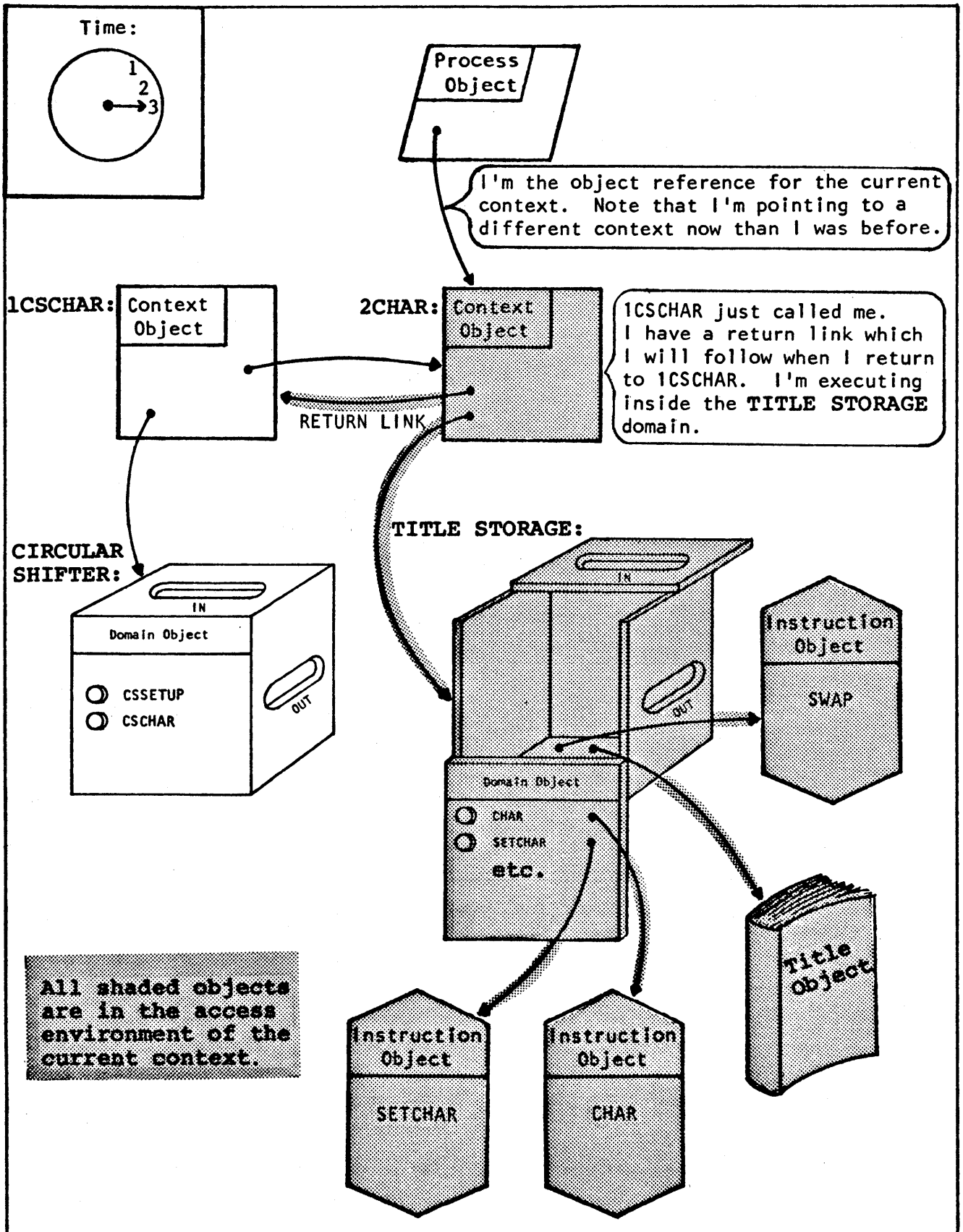

Time:
1
2
3
Process Object
I'm the object reference for the current context. Note that I'm pointing to a different context now than I was before.
1CSCHAR:
Context Object
2CHAR:
Context Object
1CSCHAR just called me. I have a return link which I will follow when I return to 1CSCHAR. I'm executing inside the TITLE STORAGE domain.
RETURN LINK
TITLE STORAGE:
CIRCULAR SHIFTER:
IN
Domain Object
CSSETUP
CSCHAR
OUT
IN
OUT
Domain Object
CHAR
SETCHAR
etc.
Instruction Object
SWAP
Title Object
Instruction Object
SETCHAR
Instruction Object
CHAR
All shaded objects are in the access environment of the current context.

Figure 6.21.3

DOMAINS SUMMARY

- A DOMAIN is a 432 hardware-recognized object that represents an instance of a module.
- NETWORKS OF DOMAINS form the static structure of a program.
- DOMAINS HIDE. Only procedures inside the domain can access objects listed in the private part of the domain.
- ACCESS ENVIRONMENTS change when a procedure is called.

Figure 6.22

3. TYPE CHECKING

This section describes type checking -- the second 432 architectural facility that supports the object-oriented design method.

The four subsections in this section are:

- SELF-MANAGED OBJECTS VS. TYPE MANAGERS
  the need for type checking
- TYPE DEFINITION OBJECTS
  a way to do type checking
- HARDWARE SUPPORT FOR TYPES
  how the architecture supports type checking
- PRIVATE TYPES
  how type definition objects can improve control of access to objects

SELF-MANAGED OBJECTS VS. TYPE MANAGERS

Previous sections have discussed modules that contain the objects they manipulate. This hides the representation of the object because it never leaves the "inside" of the module. This kind of module is called a self-managed object, because the object cannot be separated from the module that manages it.

There is a second kind of module called a type manager module. A type manager module is different from a self-managed object because it does not always contain the object it manipulates. A type manager allows other modules using the object to possess it until it needs to be manipulated. At that time, the module using the object passes it to the type manager, which performs the manipulation and then returns the object to the module using it.

Figure 6.23 contrasts these two kinds of modules. The only difference between the two is where the object is located before and after the operation is performed. The self-managed object keeps the object inside the module at all times; the type manager lets the module using the object possess it when its not being manipulated by the type manager.

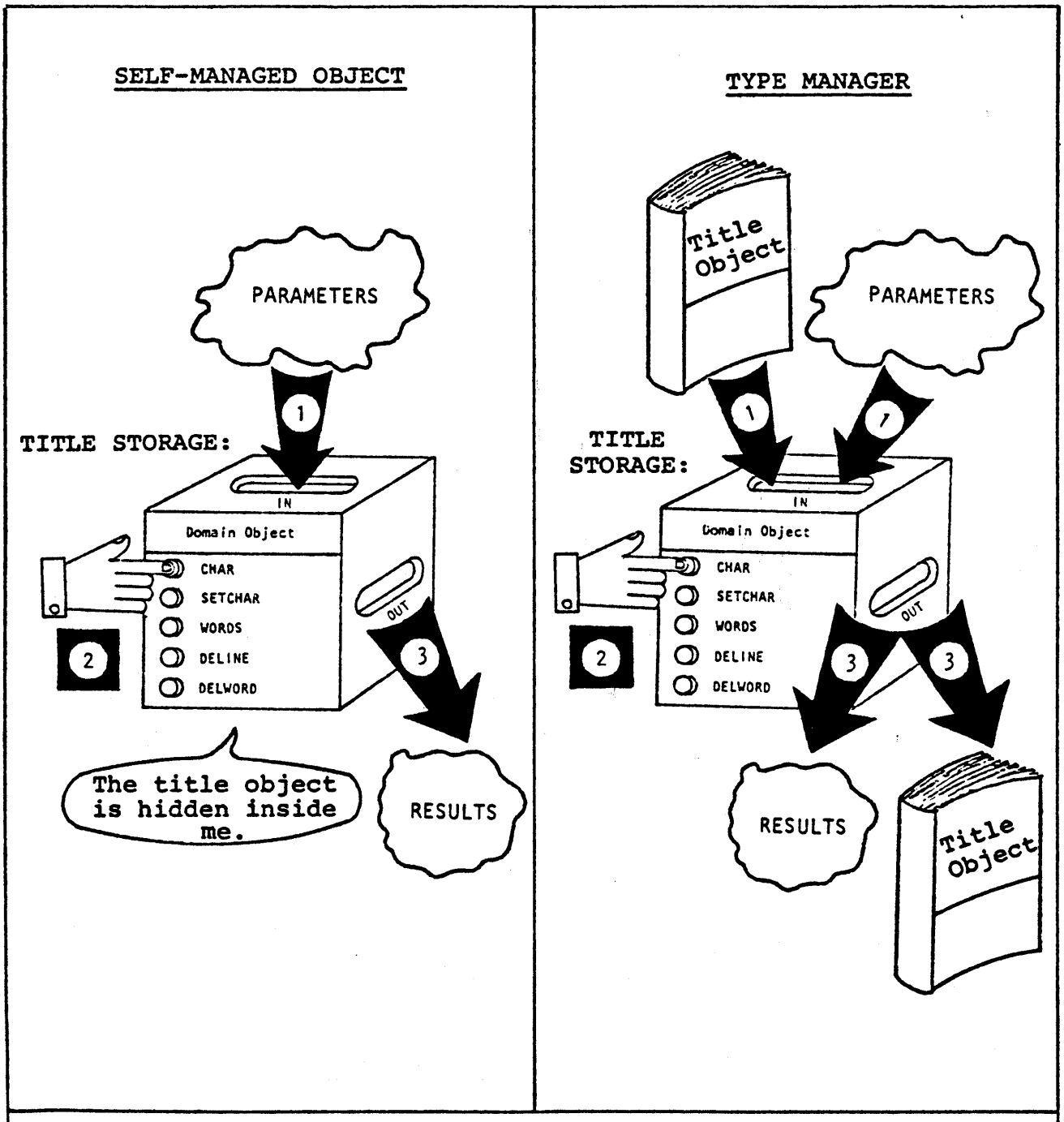


Figure 6.23 -- Two ways to build a module

The major advantage of type managers is that a new instance of the module is not needed for each new instance of the object. One module can be used as the type manager for many instances of the object.

As an example, consider a module that manages a user's disk file directory. If there are three users: ANN, BOB, and CAROL, using self-managed objects, there must be three instances of the FILE DIRECTORY MANAGER -- each one containing a user's FILE DIRECTORY object (see Figure 6.24). Using a type manager, there are still three FILE DIRECTORY objects, but only one instance of the FILE DIRECTORY MANAGER.

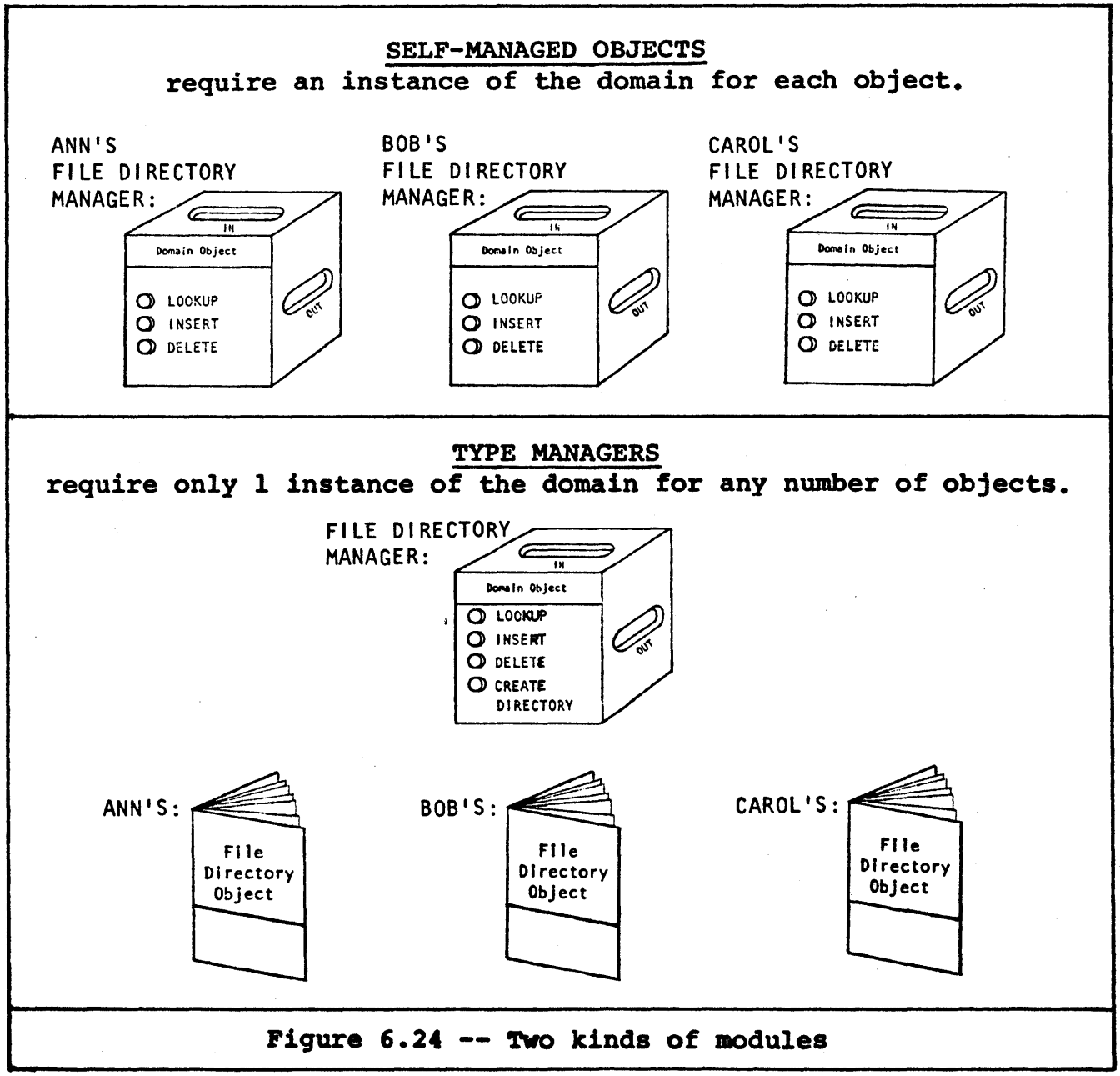


Figure 6.24 -- Two kinds of modules

Note that with a type manager, the CREATE DIRECTORY procedure can be in the file directory manager along with the other procedures. This procedure is used to create a new directory object whenever a new user is added.

Type managers present some problems, though. For example, how does the type manager check to insure that the object it is given to manipulate is indeed an object of the type it manipulates? For example, how does a type manager for file directories know that it is given a file directory object to manipulate and not something else, like a process object or a context object?

A second problem is controlling access to the object. This is no problem with the self-managed object, because other modules never possess the object. But, with type managers, other modules do possess the object. How can the type manager insure that other modules cannot manipulate the object?

The following sections explain how the 432 solves these problems.

LABELS

Chapter 2 stressed that every object has a label that tells its type. Note that each of the symbols used to represent an object has a label (see Figure 6.25).

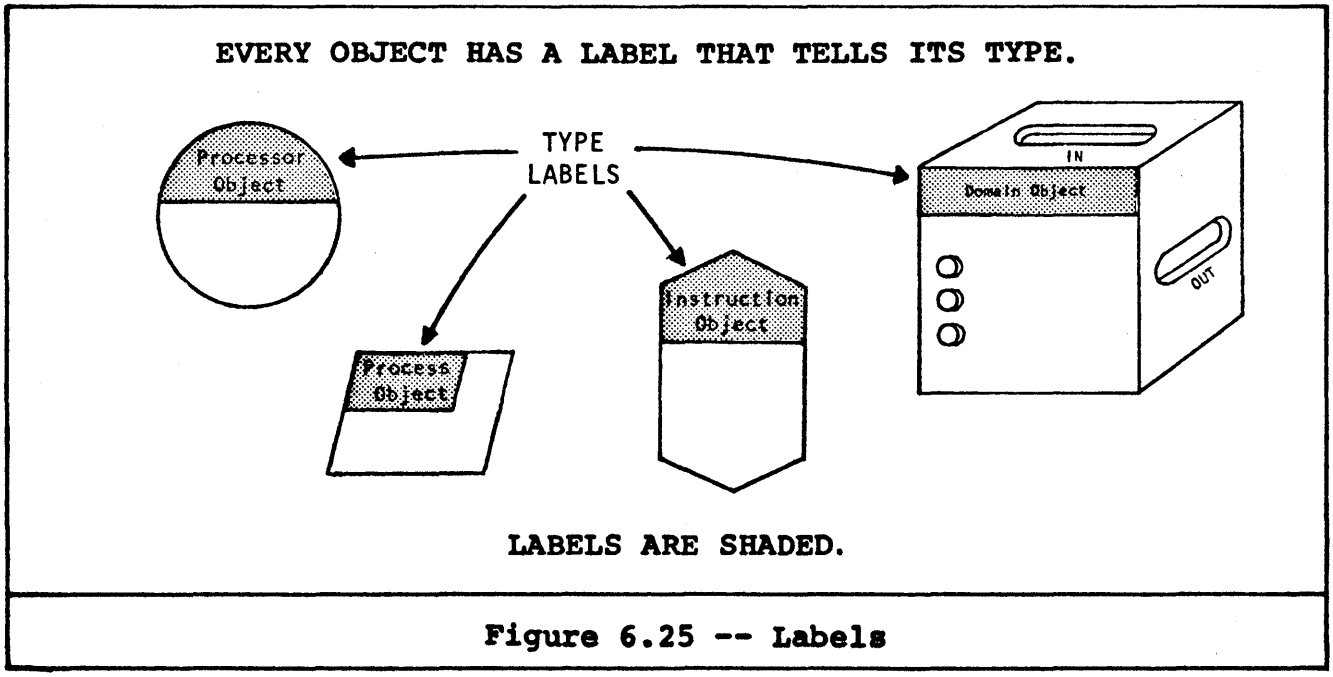


Figure 6.25 -- Labels

These labels are used to solve the first problem discussed in the last section -- insuring that the object to be manipulated is of the proper type.

For example, when a processor executes any hardware instruction that manipulates objects, it checks these labels before it executes the instruction. This is called type checking. Consider the SEND instruction explained in Chapter 4. This instruction is supposed to contain a reference for a communication port object as one of its parameters. When the processor executes the instruction, it checks the type label on this object to make sure it is indeed a communication port. If it is not, the instruction faults[1] and does not execute.

Software can also check the labels on these hardware-defined objects to make sure the object it manipulates is of the proper type. In fact, software can even create and check labels for software-defined object types. This means that software-defined object types, like title objects, can be given a type label that can be checked by the type manager before it manipulates the object.

Figure 6.26 compares a "normal" operation with an object of the correct type with an "illegal" attempt to perform the operation on the wrong object type. When the wrong type is used, the error is detected before the operation is performed, and the "illegal" object is left in its original state.

---

[1]The 432 has an extensive fault-handling mechanism. For details, see the Architecture Reference Manuals.

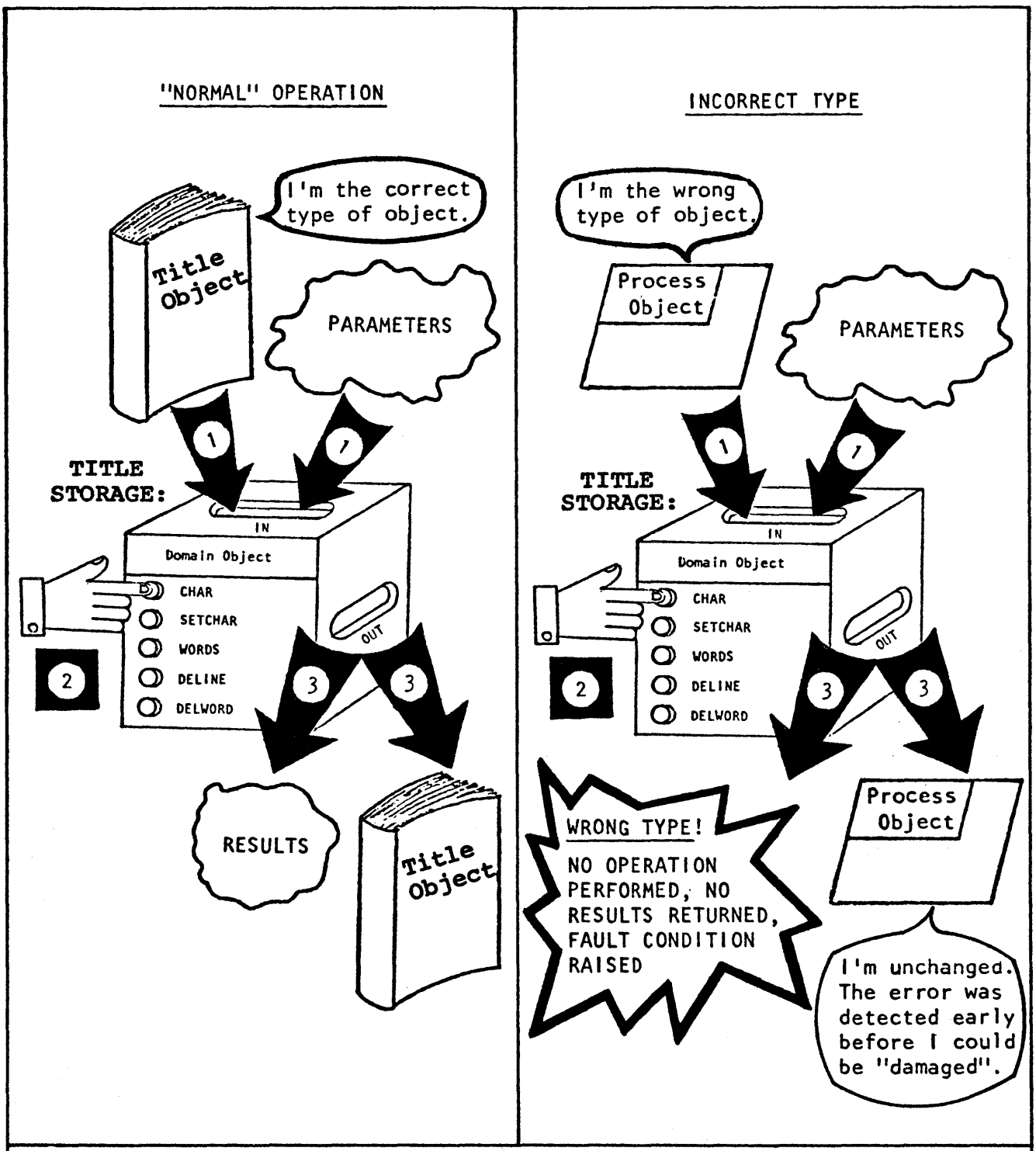


Figure 6.26 -- Type checking helps detect errors.

Clearly, labels solve the first problem. By using labels, the type of the object given to the type manager can be checked. But, what about the second problem: how to make sure that only the type manager can manipulate an object when other modules are permitted to possess it? Answer: by using labels. This is explained in the "Private Types" subsection. But first, I will cover how labels are implemented on the 432.

HARDWARE SUPPORT FOR LABELS

The 432 uses two mechanisms to implement labels. The first is used to label hardware-defined object types and is highly optimized for speedy execution. The second mechanism extends the first mechanism to provide a general and flexible facility for software-defined object types.

The labeling mechanism for hardware-defined object types is built-in to the object-oriented addressing mechanism of the 432. Every object has a label that tells its hardware type. This type can be very specific, such as "context object", "processor object", or "domain object", or the type can be rather general, such as "data object". The important point is that every object has this kind of label even if it is not very specific.

In addition to the label giving the hardware type, an object can have an optional label that describes its software-defined type. This is the second labeling mechanism provided by the 432. It extends the hardware-defined type mechanism by enabling software to give any hardware-defined type (such as "data object"), a more specific software-defined type (such as "document object").

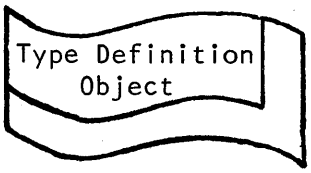


Figure 6.27 -- A type definition object

A software-defined type is represented by a type definition object (Figure 6.27 shows the symbol for a type definition object). The implementation view in Figure 6.28 shows a data object with the type definition object that says it is a document object. Note that the type definition object is actually a separate object. The object-oriented addressing mechanism has a special facility for keeping track of which type definition object types which object. For the purposes of this book, it suffices to draw a typed object as one object, as is shown in the conceptual view of Figure 6.28.

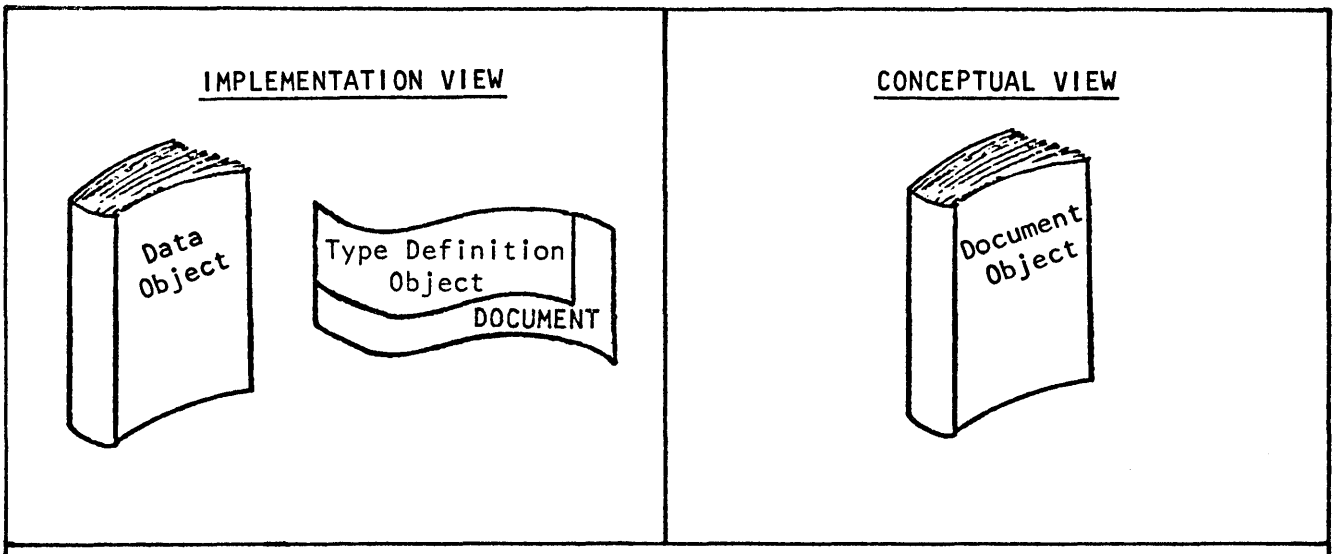


Figure 6.28 -- Two ways to draw an object with a software-defined label

The 432 architecture includes hardware support for creating and checking both hardware- and software-defined types. This insures that types are unforgeable. A type manager that creates DOCUMENT OBJECTS, types them, and gives them away, can be 100% confident that no other module has the ability to type an object as a DOCUMENT OBJECT. Other modules may be able to read the type, and tell that the object is a DOCUMENT OBJECT, but they cannot type any object as a DOCUMENT OBJECT.

I won't go into the details of how the 432 makes types unforgeable, but conceptually it is similar to our ability to possess a piece of paper that is typed as a one-dollar bill. We can read the type on the paper and tell it is a one-dollar bill, yet we cannot type other pieces of paper as one-dollar bills. The reason we cannot type other pieces of paper is that we do not possess the object needed to do that -- a printing press. Without the printing press, we cannot type paper as one dollar bills.

The 432 makes types unforgeable in a similar manner. To make a type definition object, you need an object called a descriptor control object. Without the descriptor control object, the hardware will not let you make type definition objects. The ability to make type definition objects is controlled by carefully controlling the distribution of descriptor control objects.

PRIVATE TYPES

Because types are unforgeable, they can also be used to control access to typed objects. Types let a type manager give an object away to other modules without giving away the ability to manipulate the object.

There are two different kinds of types because types are used for two different reasons. The first reason for using types has been covered -- type checking -- making sure the object that is going to be manipulated is of the proper type. These types are called "public types" because they allow a type manager to stamp its unforgeable type on the objects it manipulates.

The second kind of label is called a "private type". A private type is like a public type in many ways. A private type is unforgeable and identifies an object's type. Any procedure that has a reference for a public-type or private-type object can read the type of the object.

A private type has one additional property that a public type does not. An object of a private type cannot be accessed unless the procedure accessing the object possesses a matching type definition.

The object's type definition acts like a lock. A procedure that has an object reference for a private object cannot access the object until it unlocks it. The object can only be unlocked if the procedure has an object reference for a type definition that matches the type definition that is locking the object.

Figure 6.29 illustrates how a procedure locks an object. There is an instruction (RETRIEVE TYPE REPRESENTATION) to lock an object so that it can be accessed in a normal manner. This instruction requires as an operand a reference for a type definition object to be used as a key to unlock the object. Figure 6.30 shows what happens when a procedure tries to use the wrong "key" -- the object stays locked.

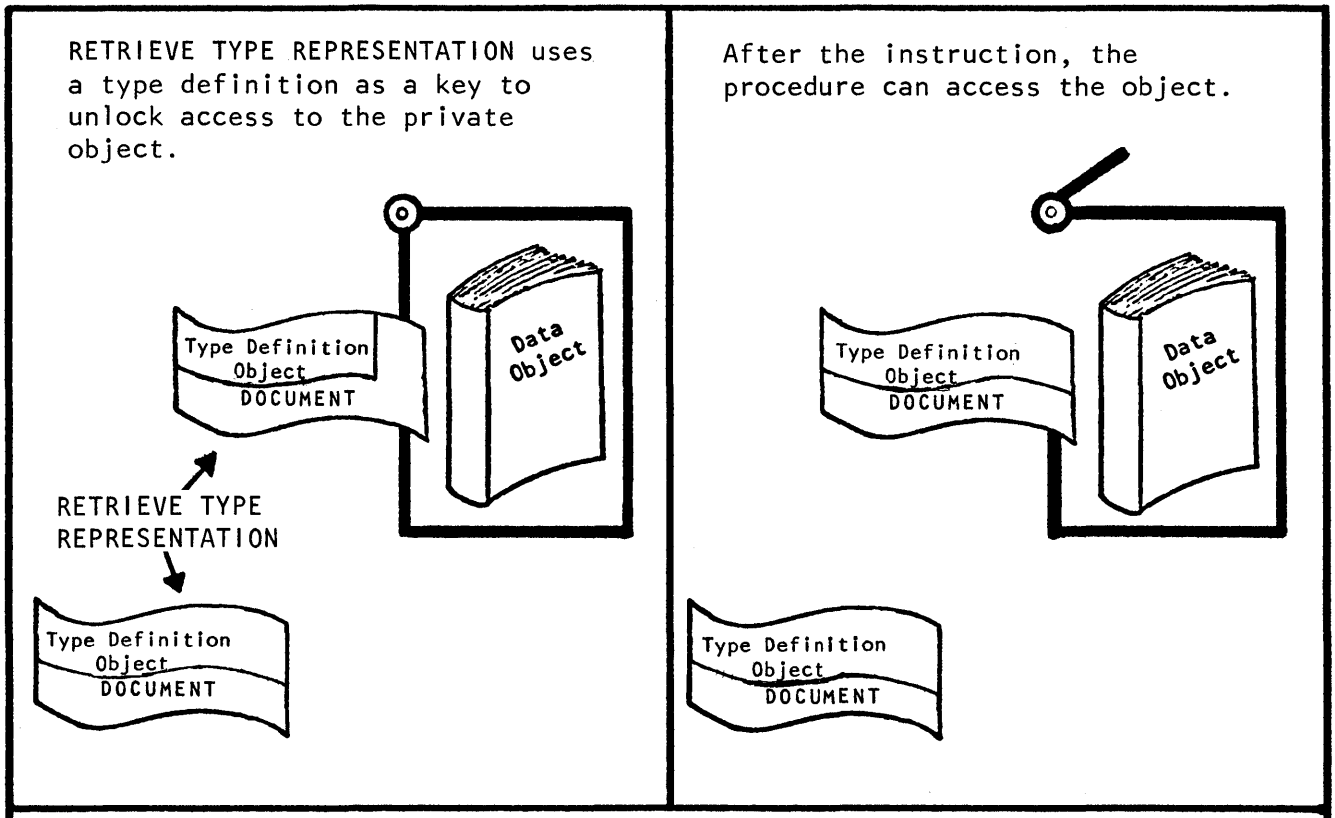


Figure 6.29 -- Successful RETRIEVE TYPE REPRESENTATION

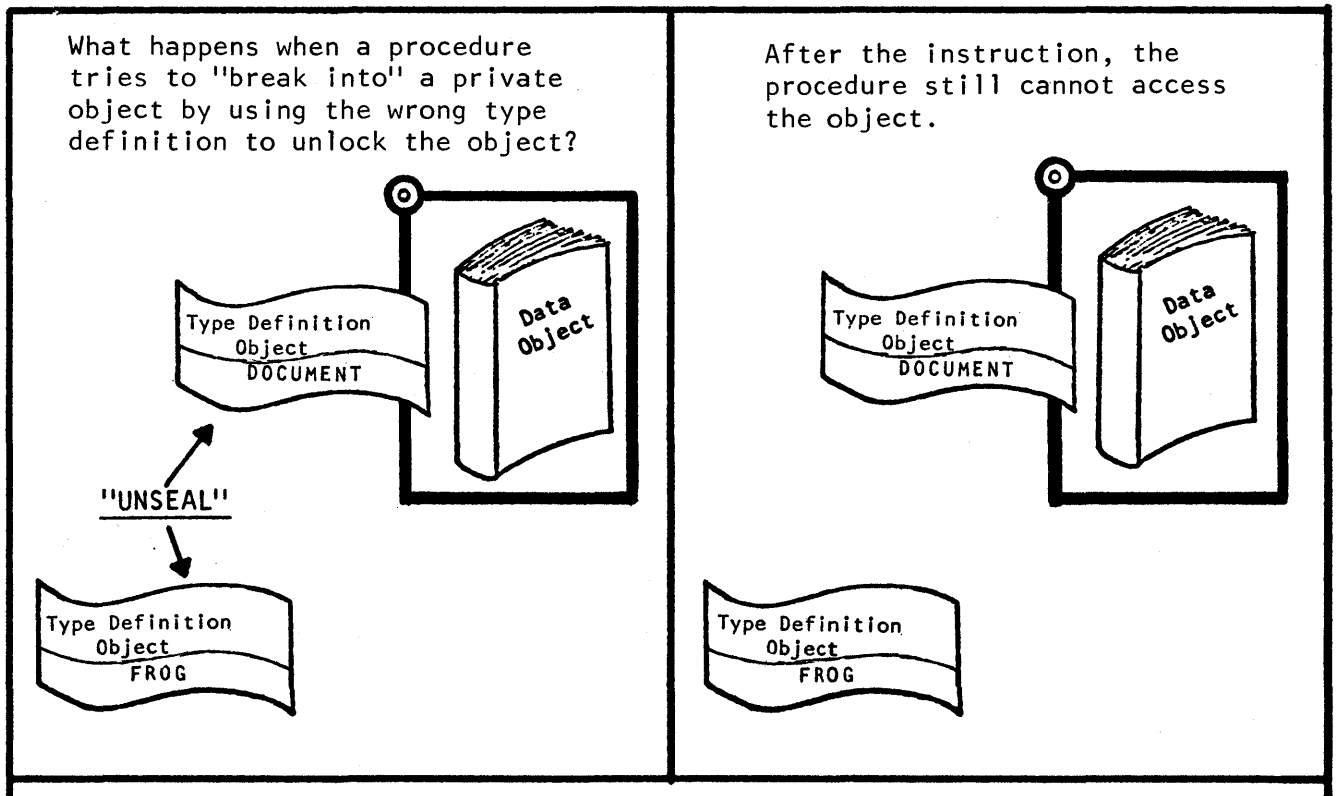


Figure 6.30 -- Unsuccessful RETRIEVE TYPE REPRESENTATION

Here are five points to remember about type checking.

- TYPE MANAGERS are used to keep objects outside of their modules when they are not being manipulated.
- HARDWARE TYPE CHECKS all hardware-defined objects before using them in an operation.
- HARDWARE SUPPORTS SOFTWARE-DEFINED TYPES with a mechanism that makes software-defined types unforgeable.
- PUBLIC TYPES are software-defined types that allow hardware to:
  - TYPE-CHECK a software-defined object type.
- PRIVATE TYPES are software-defined types that allow hardware to:
  - TYPE-CHECK a software-defined object type.
  - CONTROL ACCESS. A module can give away an object without giving away the ability to access it.

Figure 6.31

4. DESIGNING SOFTWARE SYSTEMS SUMMARY

DESIGN METHOD

- The OBJECT-ORIENTED DESIGN METHOD uses information hiding as the criteria for modularization. This increases the independence of modules, making them more flexible, more comprehensible, and quicker to program.
- TYPE MANAGERS are modules that package all knowledge of an object type. They contain all the procedures that directly manipulate objects of the type managed, plus all the static information shared by the type manager's procedures.

DOMAINS

- A DOMAIN is a 432 hardware-recognized object that represents an instance of a module.
- NETWORKS OF DOMAINS form the static structure of a program.
- DOMAINS HIDE. Only procedures inside the domain can access objects listed in the private part of the domain.
- ACCESS ENVIRONMENTS can change drastically when a procedure inside a different domain is called.

TYPE CHECKING SUMMARY

- TYPE MANAGERS are used to keep objects outside of their modules when they are not being manipulated.
- The HARDWARE TYPE-CHECKS all hardware-defined objects before using them in an operation.
- The HARDWARE SUPPORTS SOFTWARE-DEFINED TYPES with a mechanism that makes software-defined types unforgeable.
- PUBLIC TYPES are software-defined types that allow hardware to:
  - TYPE-CHECK a software-defined object type.
- PRIVATE TYPES are software-defined types that allow hardware to:
  - TYPE-CHECK a software-defined object type.
  - CONTROL ACCESS. A module can give away an object without giving away the ability to access it.

Figure 6.32

DESIGNING SOFTWARE SYSTEMS QUIZ

1. What is the difference between conventional and object-oriented modularizations?

2. What is a domain?

3. What is the difference between a procedure that is executing within a domain and a procedure that has a reference for a domain?

4. What is the difference between a self-managed object and a type manager?

5. How can a module possess an object without being able to manipulate it?

KEY TO DESIGNING SOFTWARE SYSTEMS QUIZ

1. What is the difference between conventional and object-oriented modularizations?

   A conventional modularization is decomposition by flowchart, i.e., each major step in the processing is a module. The object-oriented modularization isolates each design decision that is likely to change within a module. An object-oriented modularization is easier to develop, easier to understand, and easier to change.

2. What is a domain?

   A domain is a hardware-recognized object used to represent a module. It is a list of all the static objects used by the module, e.g., procedures and static data.

3. What is the difference between a procedure that is executing within a domain and a procedure that has a reference for a domain?

   A procedure executing within a domain has access to the objects listed in both the public and private parts of the domain. A procedure that is not executing within a domain, but does have an object reference for it, can only access the objects listed in the public part of the domain.

4. What is the difference between a self-managed object and a type manager?

   Self-managing objects and type managers are both modules. The difference is that a self-managing object module always contains the object it is managing. A type manager, on the other hand, only contains the object while it is being manipulated. One type manager can be used to manage several objects because the selected object is given to the type manager when manipulation is required; a single self-managing object, however, can only manage one object, itself.

5. How can a module possess an object without being able to manipulate it?

   A module can possess a private type object without being able to manipulate it. To manipulate a private type object, a module must be able to unlock it. A private object can only be unlocked if the module possesses a matching type definition that can be used as a key to unlock the object.

Chapter 7

THE I/O SUBSYSTEM

This chapter has four sections that describe the I/O subsystem of the 432 micromainframe:

- 432 SYSTEM ORGANIZATION

  This section explains how the I/O subsystem fits into the physical organization of a 432 system.

- I/O SUBSYSTEM ORGANIZATION

  This section describes the physical organization of the I/O subsystem.

- THE INTERFACE PROCESSOR

  Interface processors connect I/O subsystems to 432 systems. This section describes the functions provided by interface processors.

- AN I/O EXAMPLE

  This section contains a simple example that illustrates how a GDP process can communicate with an I/O subsystem to perform I/O.

432 SYSTEM ORGANIZATION

This section gives you an overview of how a 432 system is physically organized, i.e., how it is partitioned into functional blocks such as processors, memory, I/O systems, and buses.

Figure 7.1 illustrates a minimal 432 system. It is divided into three major parts:

- A data processing system that handles all the data processing operations, i.e., everything but I/O
- An I/O subsystem that handles all I/O processing operations and is fully compatible with Intel's broad family of peripherals

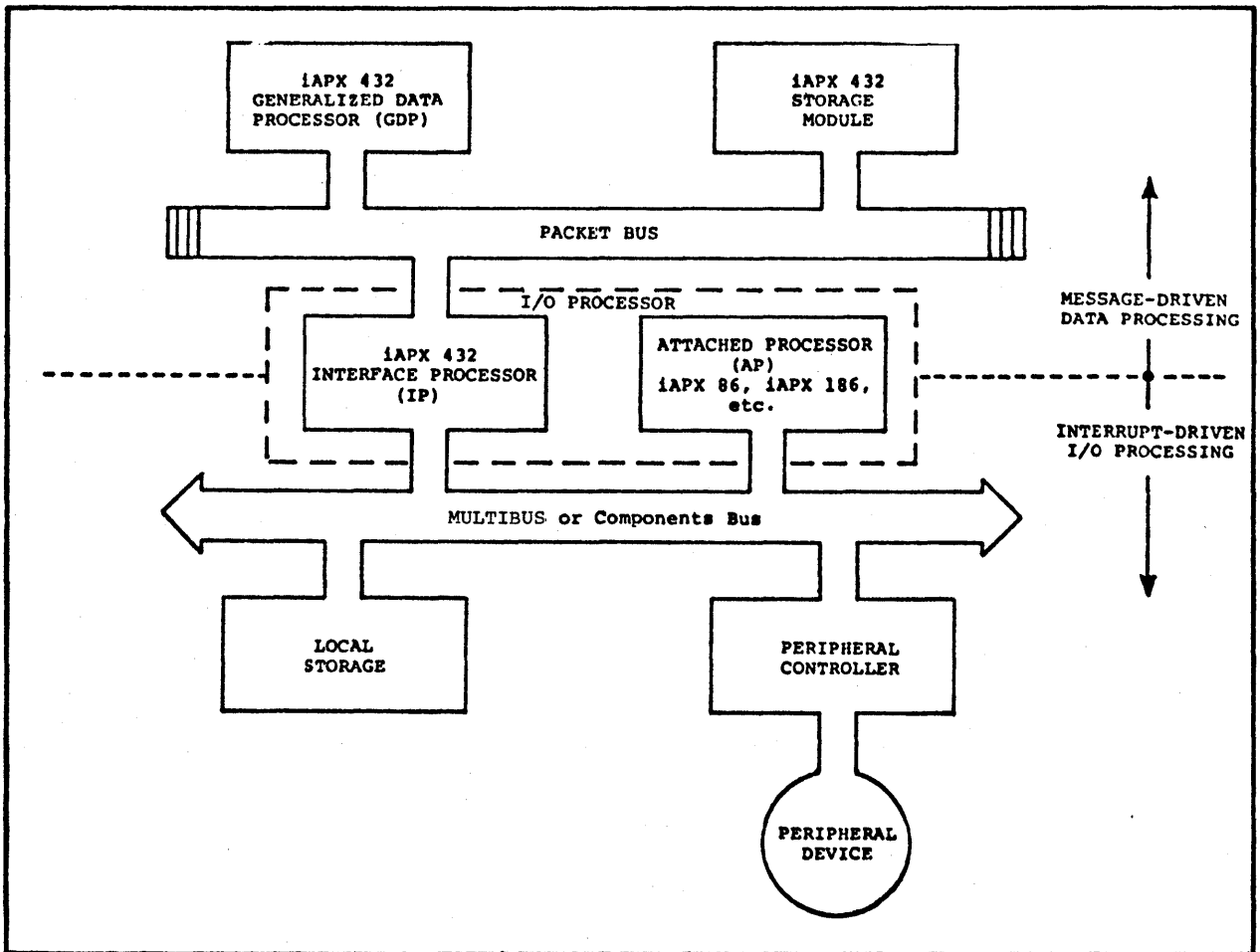


Figure 7.1 -- System organization

This system organization has three important advantages:

- Interrupts are isolated within the I/O subsystem.
- Both I/O and data processing performance are extensible.
- Attached processing by Intel's existing line of microprocessors and peripherals is supported.

Interrupts are isolated within the I/O subsystem. Interrupts are interesting because of their "two-faced" nature. They are undesirable, but essential. Historically interrupts have been a source of system reliability problems. Because of their random nature, interrupts can cause very complex and unpredictable interactions to occur. This problem is manageable in smaller systems, but becomes much more difficult to deal with in large complex systems. On the other hand, interrupts are necessary for interfacing to today's peripherals and real-time environments.

The system organization of the 432 micromainframe accommodates the two-faced nature of interrupts. By supporting interrupts, the 432 retains the ability to interface to today's peripherals and real-time systems. By confining interrupts within an interrupt-driven I/O subsystem and by using message-driven communication in the data processing system, the 432 limits the size of the system that is affected by the interrupts (see Figure 7.1). This improves the reliability of the overall system and makes it easier to debug.

Independent extensibility of I/O and data processing performance. The 432 provides a range of both I/O and data processing performance because additional I/O subsystems and data processors can be added as necessary.

Chapter 5 explained how the 432 schedules and dispatches processes on multiple GDPs in a way that allows the same software to run on systems with 1, 2, or many processors, called transparent multiprocessing. Figure 7.2 illustrates how multiple processors are configured in a system. Additional data processors, storage modules, and I/O subsystems can be added as needed to achieve the desired performance.

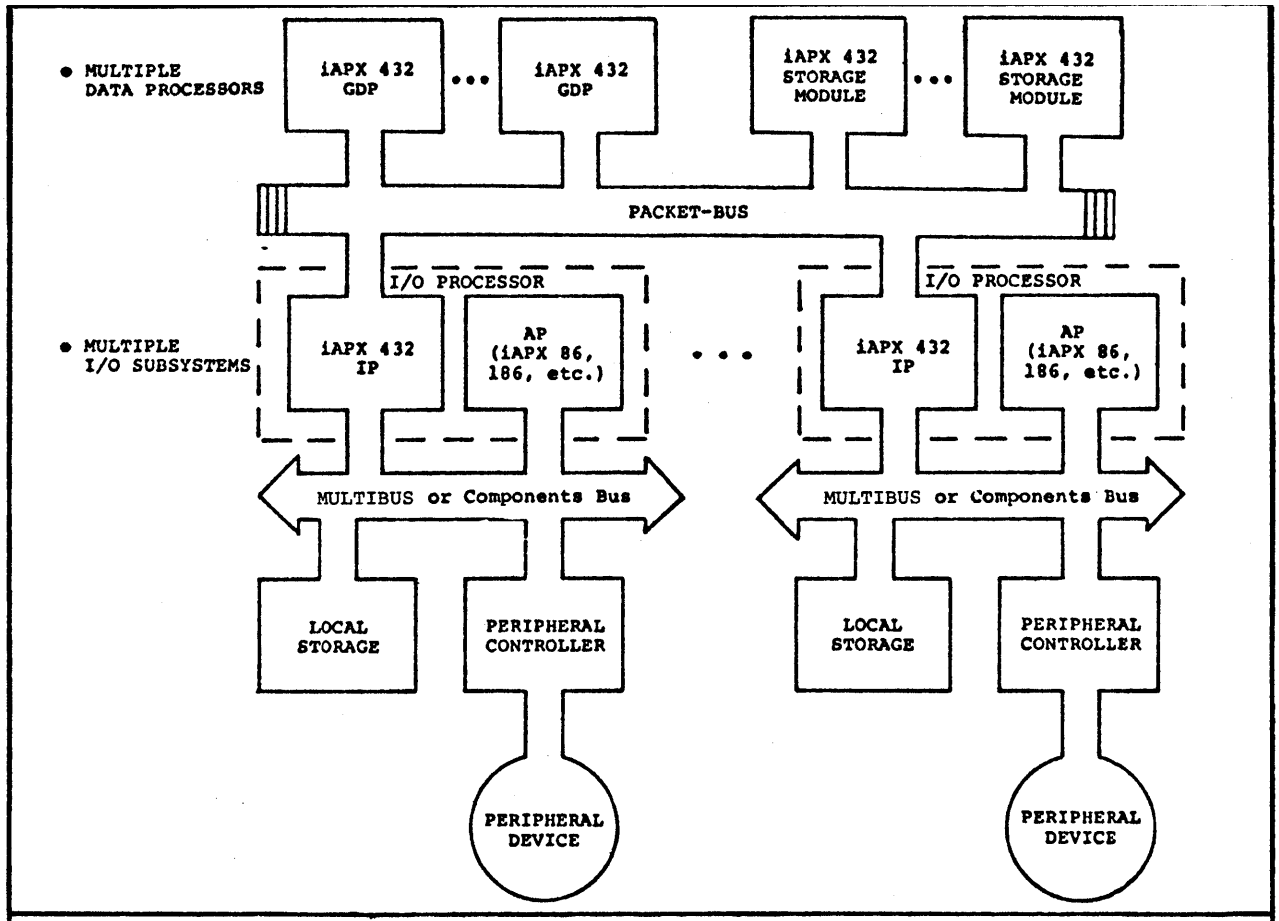


Figure 7.2 -- Performance extensibility

Transparent multiprocessing for generalized data processors not only provides "instant" breadth of product line by simplifying the construction of systems with different levels of performance, it also allows performance to be a design variable. The configuration of a system can be tuned to meet its data processing needs by adding data processors as needed, and can be independently tuned to meet its I/O requirements by adding additional I/O subsystems.

Fully independent and decentralized I/O subsystems. These subsystems completely off-load I/O processing functions (e.g., setting up DMA transfers, polling devices) from the data processing system, allowing generalized data processors to spend more of their time processing data instead of managing complex I/O details.

The level of function handled by the I/O subsystem can be varied by the system designer. For example, if the application requires an inexpensive I/O subsystem, the designer can elect to use an 8085 as the attached processor (AP) and limit the I/O subsystem functions to a fairly low level, such as setting up DMA transfers and polling devices. Higher-level functions such as file management can be performed by the data processing system.

On the other hand, if the application requires a complex I/O subsystem, the system designer can use an iAPX 86 or 186 as the attached processor. With this increased processing power, the I/O subsystem can off-load more functions from the data processing system, e.g., part of the file management function could be performed by the I/O subsystem instead of the data processing system.

The ability to support multiple I/O subsystems allows each subsystem's hardware and software configuration to be optimized for specific types of I/O. Figure 7.3 shows one example. I/O subsystem #1 is configured to handle high speed disk drives that make DMA transfers, while I/O subsystem #2 is configured to manage 15 terminals using a polling protocol. Note that this independent and decentralized I/O structure is very similar to the channel structure found on many of today's mainframe computers.

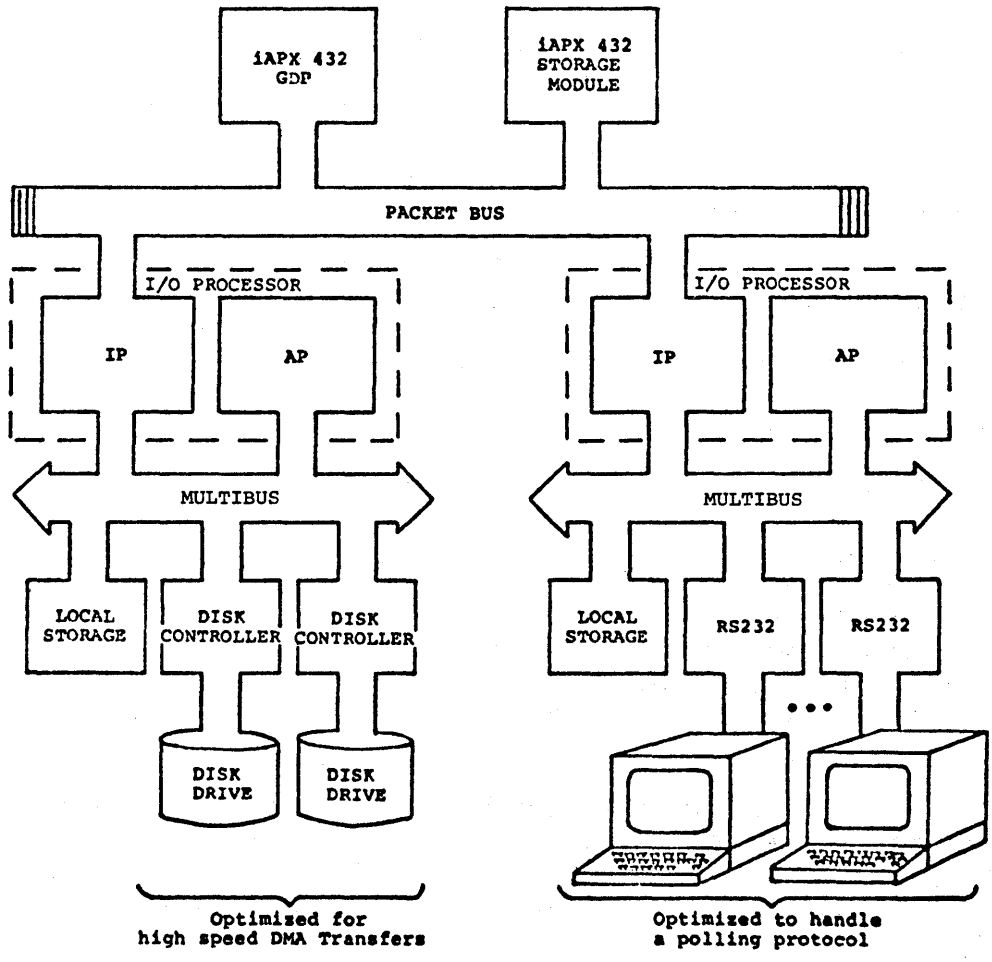


Figure 7.3 -- Optimized I/O subsystems

Attached processing. The design of the 432 allows Intel's existing line of microprocessors (e.g., 8085) to be attached as subsystems to the 432 micromainframe. Thus, programs written for other Intel processors can be run on an attached processor in a 432 system, preserving the software investment.

In summary, the four major advantages of the 432's system organization are:

- Interrupts are isolated within the I/O subsystem.
- Both I/O and data processing performance are extensible.
- I/O subsystems are fully independent and decentralized.
- Attached processing by Intel's existing line of microprocessors and peripherals is supported.

I/O SUBSYSTEM ORGANIZATION

Now that you have had an overview of the 432 system organization, the remainder of the chapter focuses on the I/O subsystem and how it interfaces with the data processing system. This section covers how the I/O subsystem is organized, the next section explains the functions provided by the IP, and the final section gives an example of an I/O transfer.

A 432 I/O subsystem has 4 parts:

- An attached processor (AP), e.g., an iAPX 86, 186, 8085, 8080, etc.
- 432 interface processor (IP)
- Local memory
- Peripheral controllers, e.g., 8271 floppy disk controller

An attached processor is half of the peripheral subsystem's I/O processor (the IP is the other half). The AP is the "intelligence" that manages an I/O subsystem. An AP does things like polling devices, responding to interrupts, and setting up and monitoring DMA transfers -- in general, all the busy work associated with controlling a group of peripheral controllers.

APs may also handle additional functions such as file debuffering, packing of logical records into physical records (blocks), and unpacking of blocks into logical records. As mentioned before, it is up to the system designer to decide where to distribute the functionality.

An interface processor (IP) is the second half of an I/O subsystem's I/O processor. An IP is a bridge between the "I/O world" and the "data processing world". IPs can be used to connect the 432 packet bus to either the 8086 component bus or to the Multibus. The next section is a closer look at how the IP works.

Local memory is used by the AP for two things. First, it is used to contain the programs that run on the AP. Second, it can be used as a buffer for data being moved from a peripheral to the data processing system's main memory or vice-versa.

Peripheral controllers are devices like 8271 floppy disk controllers or 8275 CRT controllers that handle the control requirements of specific I/O devices.

THE INTERFACE PROCESSOR

This section covers the facilities provided by the interface processor. It starts off by explaining:

- The structure of an IP program

and then describes the three major facilities provided by the IP:

- Mapping memory locations
- Extending the AP instruction set
- Initialization and diagnostic support

The Structure of an IP "Program"

Chapter 3 described the structure of a program for a 432 GDP. A 432 IP program has a similar structure.

Figure 7.4 illustrates the structure of an IP program. Note that an IP has a processor object, a process object, and a context object just like a GDP does.

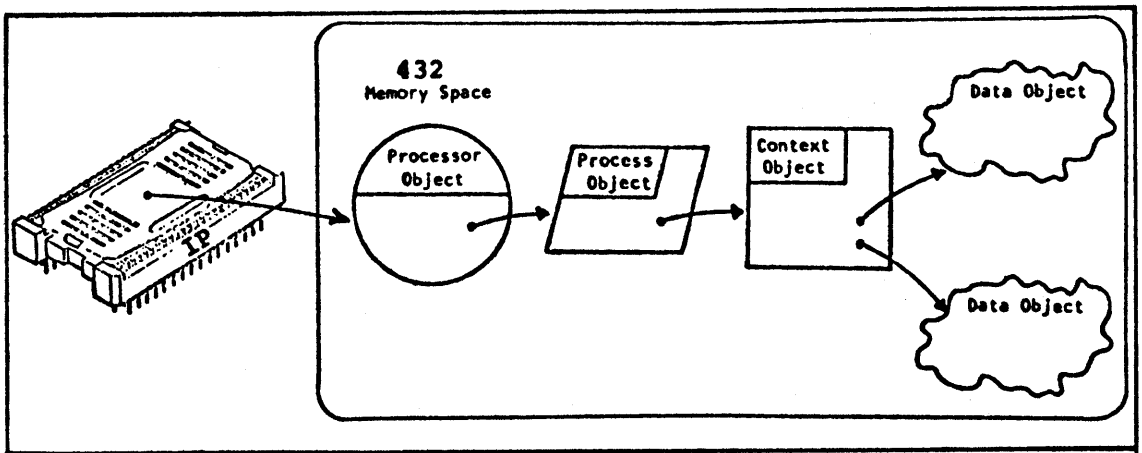


Figure 7.4 -- The structure of an IP "program"

IP processor, process, and context objects are similar to their GDP counterparts in many ways. Much of the information they contain is the same, because the objects fulfill much the same function. But, they are not identical; some of the information they contain is specific to the particular type of processor (IP).

One of the similarities shown Figure 7.4 is that an IP context object has a list of references for data objects that can be accessed by the context. The IP uses the same protection mechanisms as a GDP. The IP can only access objects for which it has an object reference. Thus, an AP using an IP's mapping facilities to access 432 memory can only access objects that are referenced by the IP program.

One of the differences shown in Figure 7.4 is that an IP context object does not have a reference for an IP instruction object. This is because IPs are slave processors; they do not fetch their own instructions. They accept and execute instructions one at a time from the attached processor.

The fact that IPs do not fetch and execute their own instructions has several other implications. In general, it means that IP program structure is much more static, i.e., less likely to change as the program executes. Here are two examples:

First, there are no CALL CONTEXT instructions to change the current context within the current process. Each IP process has only a single, fixed context.

The reason is straightforward. Contexts are switched whenever the flow of control changes from one procedure to another. But the IP does not have any flow of control because it does not fetch its own instructions. It accepts them one at a time as requests from the AP. Thus, the IP never really needs to change contexts, so it does not.

However, the IP may change processes, and when it does so, the new process has a new process object and context object associated with it. This implicitly switches the current context.

A second example of the IP's more static program structure is process switching. Process switching is not done automatically, as on the GDP, but is controlled by the AP software. For example, when the AP software detects that a SEND or RECEIVE involving one process has blocked, then the AP can switch to another process by changing the IP's current process reference.

When a process running on a GDP needs to communicate with the I/O subsystem, it does so by sending a message to a communication port that is used by the I/O subsystem as an "in-basket" for requests from the GDPs. When the GDP executes a SEND instruction to send a message to the communication port, one of two things happen, depending on the state of the communication port.

If the IP process is not waiting at the port for a message, then the GDP simply queues the message at the port. The execution of this instruction is exactly the same as if the GDP had sent a message to another GDP process.

But, what happens if the IP process is ready to receive a message and is waiting at the communication port? If the message was being sent from one GDP process to another GDP process, the message would be given to the waiting process which would then be sent to its dispatching port. It works the same way when the IP process is waiting to receive a message. The message is given to the IP process which is then sent to the IP dispatching port.

But why does the IP need separate process and context objects? After all, there is a one-to-one correspondence between them. When a message is sent from a GDP process to an IP process, the GDP uses the same SEND instruction it uses to send messages to other GDP processes. This means the GDP expects the IP process to be structured in much the same way a GDP process is structured, i.e., IP processes need to be structured to be compatible with the GDP mechanisms that interface with the IP. One of the implications of these compatibility constraints is that the IP needs separate process and context objects.

Structuring an IP program similar to a GDP program has two advantages. First, it simplifies the interface between processors by allowing a GDP to interface with IPs by using the same mechanisms it uses to interface with other GDPs. In other words, a GDP does not need to "know" a different program structure for IPs. The second advantage is that the structure of an IP program is easy for people to learn, because it is very similar to the structure of a GDP program.

In summary, this subsection has described the similarities and differences between the IP's program structure and a GDP's program structure. In general, the two program structures are very similar. This makes it easier to interface GDP and IP processors and also makes it easier to understand the IP program structure. But, there are some differences between the two program structures. First, the IP program structure tends to be much more static, because IPs do not fetch and execute their own instructions. Second, some of the information recorded in processor objects, process objects, etc., is processor-specific and thus is slightly different for GDPs and IPs.

This concludes the discussion of IP program structure. The next three subsections explain the three major facilities provided by an IP: mapping memory locations, extending the AP instruction set, and initialization and diagnostic support.

Mapping Memory Locations (Windows)

A primary function is to provide protected windows into the 432 address space. Figure 7.4 shows how this windowing works. Locations within the AP address space are mapped by the IP into the 432 address space. Whenever an AP (or any other device in the I/O subsystem) reads or writes to these mapped locations, the IP acts as a memory controller, grabs the bus, prevents access to any RAM or ROM in the AP subsystem that has the same address, and instead performs the read or write using the mapped address in the 432 address space. This mapping is called a "window" because it allows an attached processor a protected view of 432 memory.

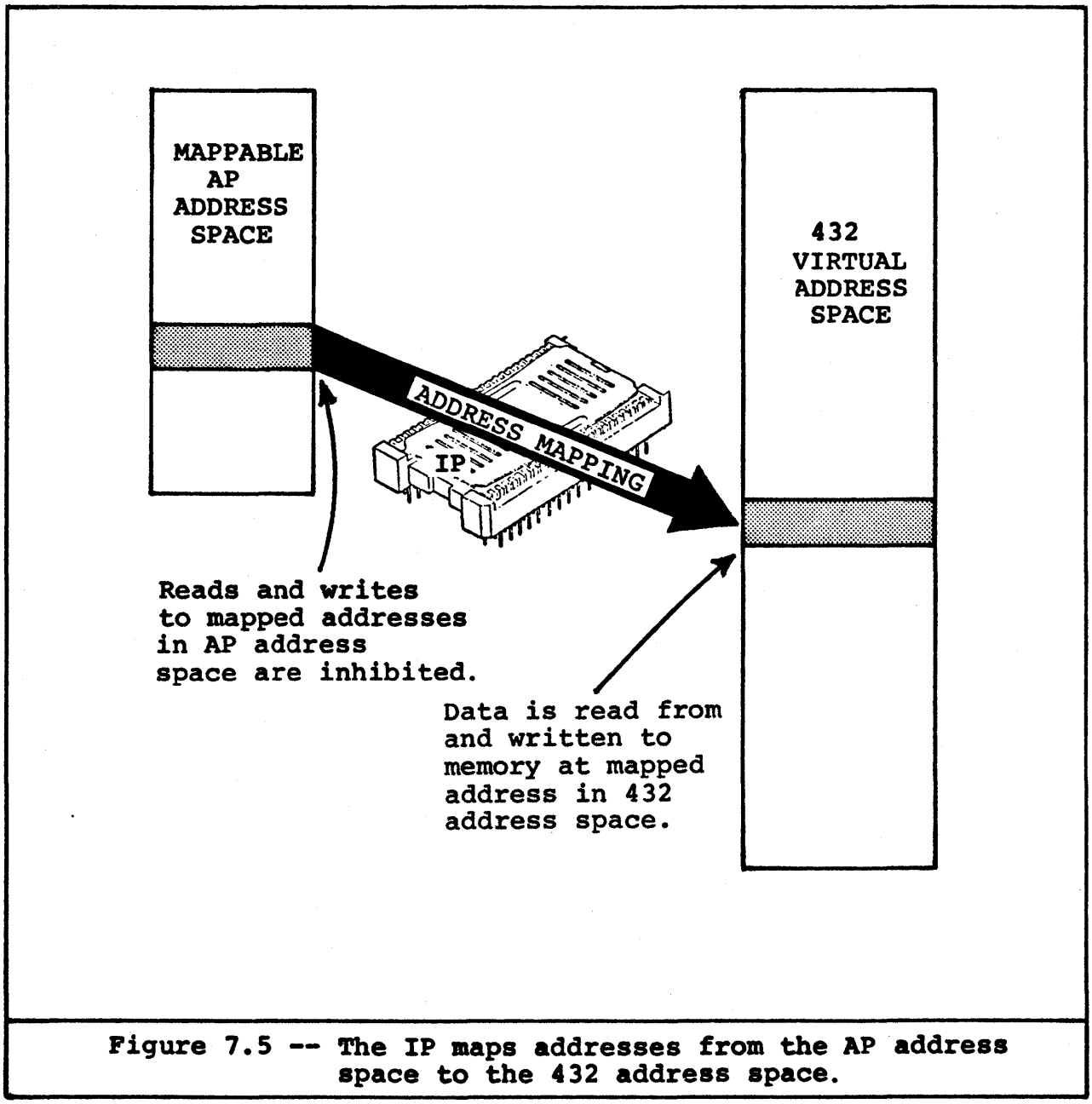


Figure 7.5 -- The IP maps addresses from the AP address space to the 432 address space.

Here is a simple example with one window. Locations 512 through 767 (decimal) in the AP address space are mapped to a 256 byte segment in the 432 address space (i.e., byte 512 in the AP address space maps to byte 0 in the 432 segment, byte 513 maps to byte 1, etc. -- see Figure 7.6).

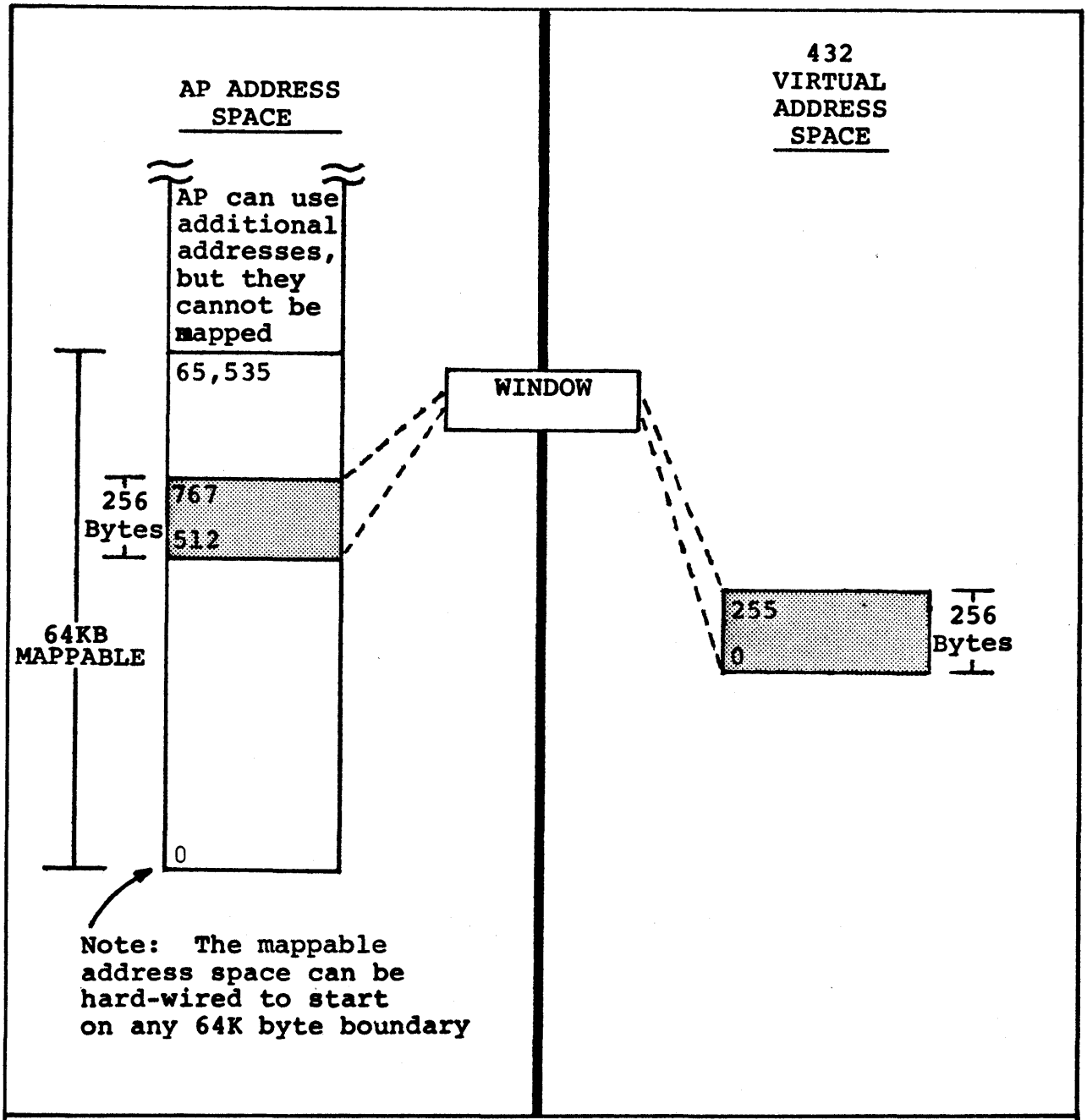


Figure 7.6 -- A window maps addresses from the AP address space to the 432 address space.

If the AP executes MOVB AX,512, it normally moves a byte from the AX register to location 512 in the AP address space. However, with the window set up, the byte is moved to byte 0 in the 432 segment. The IP monitors the I/O subsystem bus and when it sees an address that it is mapping, it performs the read or write using the mapped location in 432 memory. To the AP, however, it seems as if part of 432 storage has been moved into its address space.

The example can be extended to consider what happens when a peripheral device makes a transfer to the 432. One way to do this is to buffer the transfer in the local storage of the I/O subsystem as illustrated in Figure 7.7. After the peripheral device has completed the transfer, the attached processor can set up a mapping and transfer the data to 432 storage.

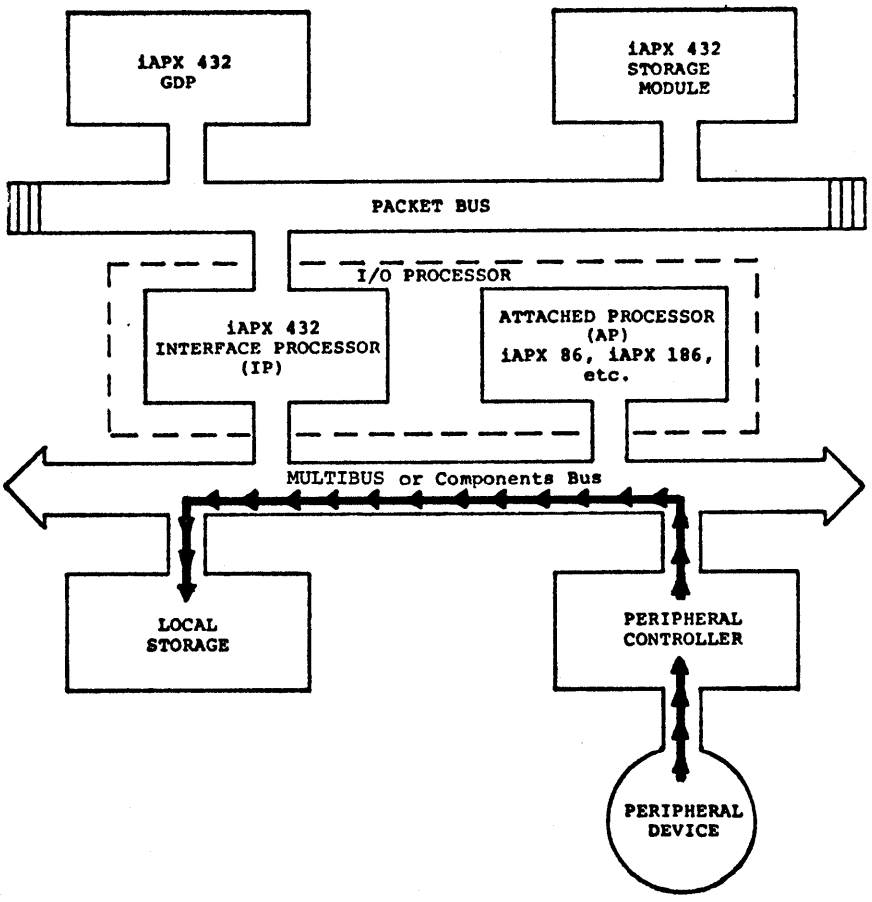


Figure 7.7 -- Unmapped data transfer

Memory mapping is performed in a way that preserves the integrity of the 432's object-oriented protection mechanism. As described in Chapter 3, a GDP context object contains a list of object references for all the objects that the context can access. The same is true for the IP context object. Only objects with object references in the context object may be accessed. To set up a map from the AP address space to the 432 address space, there must be an object reference for the locations selected in the 432 address space. Locations that lie outside of the objects referenced by the context object cannot be addressed because a map cannot be set up.

So far, the discussion of the IP has been limited to examples with a single window. In a running system it is often desirable to have several windows, thus a single IP can map to four different windows for data transfer at a time (see Figure 7.9). Each window can map a segment from 1 to 32K bytes long. These maps are set up and changed programmatically using one of the IP's instructions. A fifth window is defined for control of the IP by the AP, but this window cannot be changed by IP instructions. Each window requires 2 to the n bytes (n = 0 to 15) in the AP address space, and windows may not overlap in the AP address space.

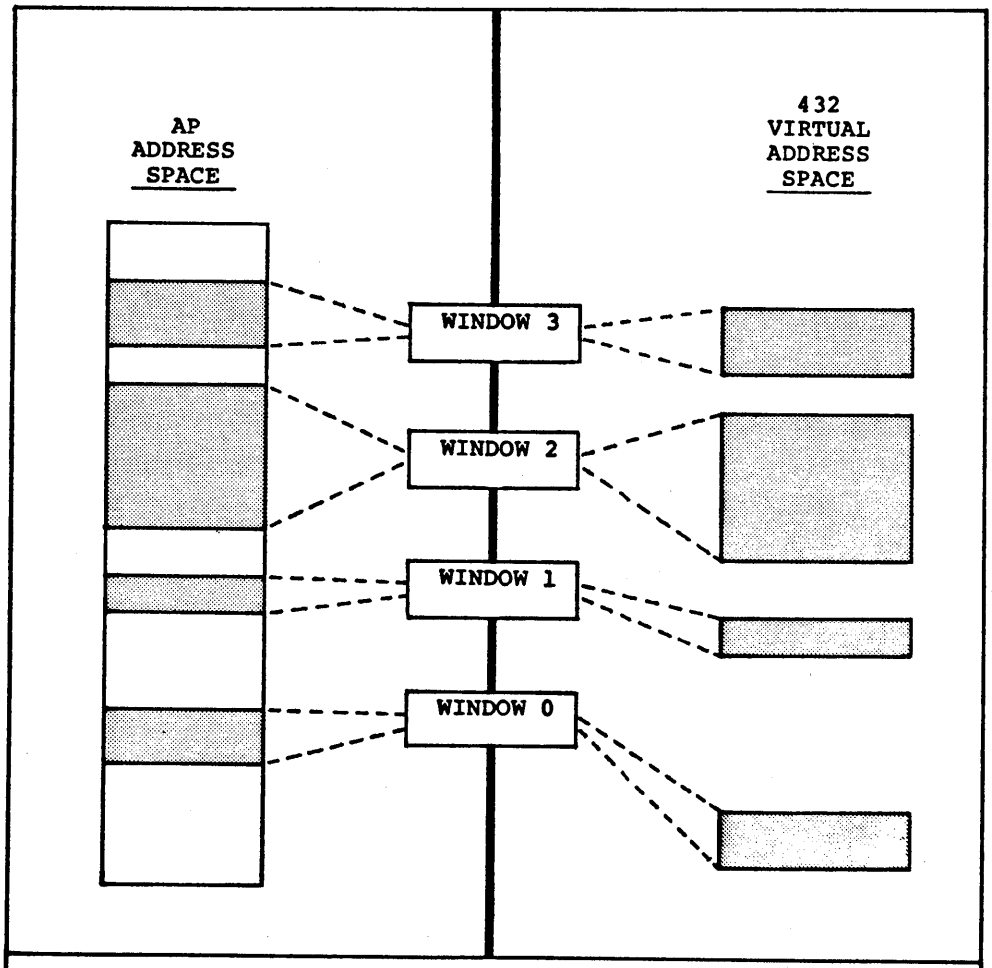


Figure 7.9 -- A single IP can map 4 data-transfer windows at a time.

Extending the Attached Processor's Instruction Set

To effectively interface with GDPs, an I/O subsystem must be able to execute the 432's object-oriented instructions such as: SEND, RECEIVE, RETRIEVE TYPE DEFINITION, etc. The interface processor provides the I/O subsystem with these operations by extending the instruction set of the attached processor. The AP and IP work together as a team -- the AP executes the arithmetic and logical operations and the IP executes the 432's high-level object-oriented instructions.

The IP executes 432 instructions as a slave processor for the AP. The IP itself does not fetch and execute a stream of instructions -- instead it executes instructions one at a time when commanded to by the AP. When it finishes executing an instruction, it interrupts the AP to signal that it is finished and waits until it is commmanded to execute another instruction.

The AP commands the IP to execute an instruction by writing the operands and opcode for the desired instruction into a function request area located in the IP's context object. The IP starts execution of the requested instruction as soon as the opcode is written into the function request area. Once it starts executing an instruction, no other instruction should be written until it finishes the current instruction. When it finishes executing the instruction, it records status information (e.g., operation successful, operation faulted) in its context object and interrupts the AP to let it know it is finished.

To give the IP a command, the AP must be able to write into the function request area in the IP's context object. It does this by using the memory mapping facilities described in the last sub-section. One of the five memory windows (WINDOW 4) is always used to map part of the IP's context object into the AP address space. This window allows the AP to give the IP a command and to read status information. Thus, to give the IP a command all the AP has to do is write the operands and the opcode to the addresses that are mapped to the function request area in the IP's context object (see Figure 7.10).

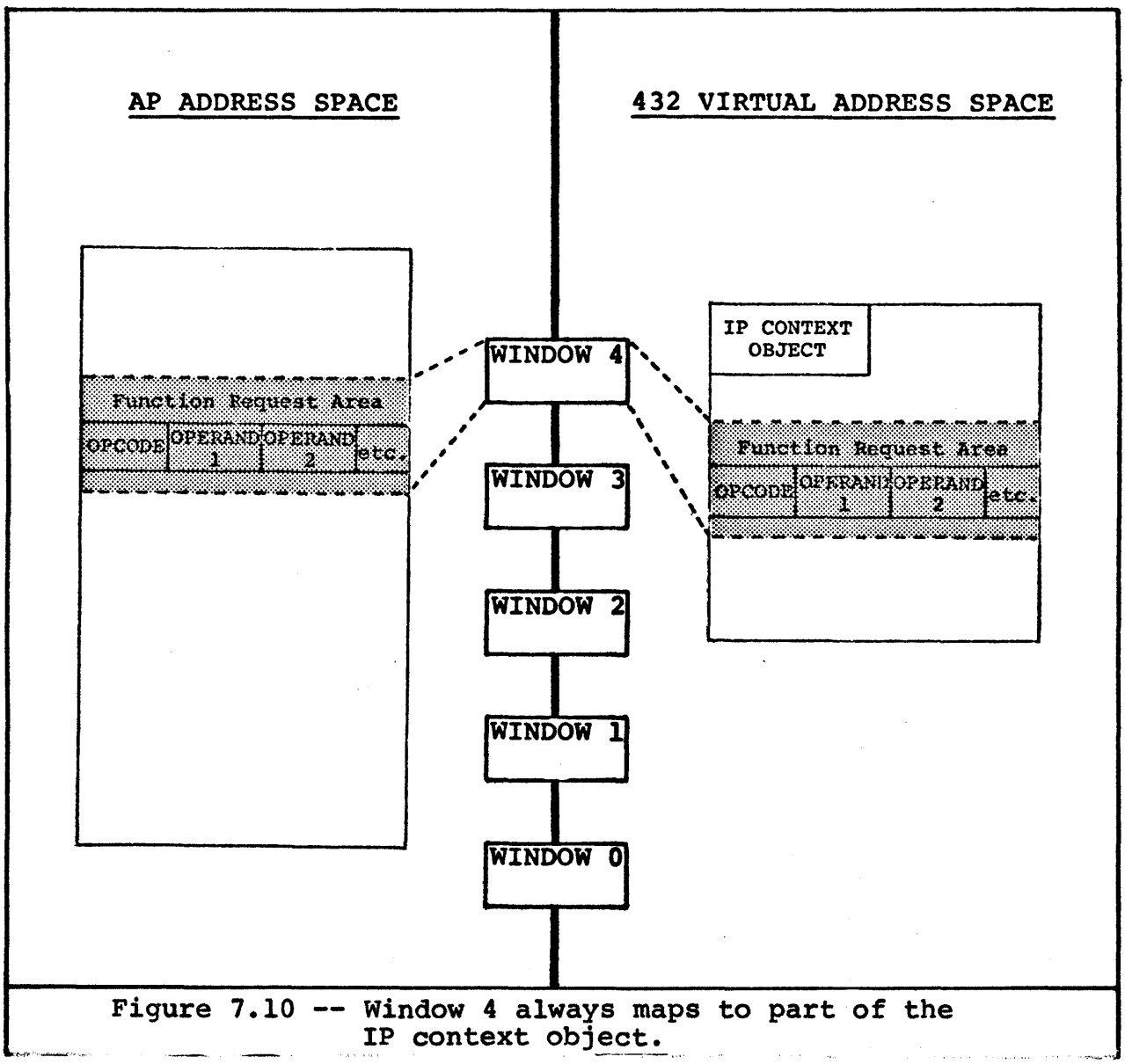


Figure 7.10 -- Window 4 always maps to part of the IP context object.

As an example, consider what happens when the I/O subsystem wants to receive a message:

1. The AP causes the IP to execute a RECEIVE instruction. It does this by writing the operands for the instruction and the opcode for the instruction into the function request area of the IP's context object. The context object resides in 432 physical memory, but the AP can write to it because Map 4 is always set up to map an address subrange from the AP address space to the context object.

2. Once the AP has written the command into the IP's function request area, the AP is free to go off and perform other duties. The IP will interrupt it when the RECEIVE is completed.

3. As soon as the AP has finished writing the operands and opcode into the function request area, the IP begins to execute the instruction. Note that the IP can only perform one function at a time. The AP may not request another function until this one completes.

4. In a moment, I will explain what happens if a message is not waiting in the communication port. For now, assume that a message is waiting in the port. The IP performs the receive operation, removing the object reference from the communication port and moving it to the IP's context object. (Note: To be able to access the message that has been received, AP software must request the IP to set up a map to it. It does this by commanding the IP to execute an UPDATE WINDOW instruction.)

5. When the IP is finished with the instruction, it records the status of the instruction (e.g., instruction successful, instruction faulted) in its context object, and interrupts the AP.

6. The AP examines the status information recorded by the IP in its context object to determine if the operation was successful.

7. Because the IP has completed the instruction, the AP can now command it to execute another instruction by writing the next instruction into the function request area.

The AP causes the IP to execute other instructions in a similar manner. These instructions include some of the 432's object-oriented instructions for interprocess communication, interprocessor communication, etc., plus an instruction for setting up and changing the mapping windows.

One IP instruction that is a bit unusual is the RECEIVE instruction. This instruction is straightforward when a message is waiting in the communication port, but is unusual when a message is not waiting in the communication port. Consider what happens when the communication port is empty:

1.-3. These steps are the same as in the example above. The AP gives the IP the command and the IP begins to respond.

4. When the IP finds out that the communication port does not contain a message, it queues up the IP process object to wait for a message. So far so good. This is the same action taken by a GDP when it finds an empty communication port. But now there is a problem. The IP can only execute one instruction at a time and it is stuck in the middle of a RECEIVE instruction waiting for a message to arrive.

5. This problem is easily resolved. The IP writes status information in its context object that indicates that it is waiting for a message, and then it interrupts the AP.

6. When the AP responds to the interrupt and checks the status, it learns that the IP did not find a message. And, since the IP has, for now, stopped executing the RECEIVE instruction, the AP can give it other commands such as CONDITIONAL SEND or UPDATE WINDOW. Normal use of the IP resumes with only one restriction. One IP process can only be waiting for one message at a time; i.e., the AP cannot request a second RECEIVE instruction until the first one completes (or is cancelled by the IP), or until the AP switches to another IP process.

7. When a message is sent to the port with the waiting IP process, the standard action described in Chapter 4 is taken. The message is given to the IP process, and the IP process is sent to the IP dispatching port. Waiting at the IP dispatching port is the processor object for the IP that initially executed the RECEIVE. This causes the processor performing the SEND to signal the IP (using a hardware interprocessor bus signal) that it has a process that requires service.

8. When the IP receives the interprocessor signal, it first finishes the instruction it is executing (if any). The AP must then cause the IP to execute a SURROGATE RECEIVE (not covered here) on the IP's dispatching port to pick up the now ready IP process. If the AP has switched the IP to another process, the IP's process reference must be changed back to the process just received at the IP's dispatching port. The IP now finishes the RECEIVE, stores status information that indicates that the RECEIVE is complete, and then interrupts the AP.

9. The AP inspects the status and learns that a message has been received. Because the previous RECEIVE is now complete, the AP can once again ask the IP to execute a RECEIVE.

This concludes the discussion of how the IP extends the instruction set of the AP. The important point to remember is that the IP is a slave to the AP and executes instructions whenever the AP loads the IP's function request area. These instructions include the relevant object-oriented instructions provided by the GDP (e.g., interprocess communication instructions) plus the instruction for setting up and changing the mapping windows.

Initialization and Diagnostic Support

Interface processors and their associated I/O subsystems have a second important role to play in addition to their normal I/O duties. The 432 generalized data processors (GDPs) require several data structures existing in memory before they are "turned on." Most of these structures are the kind that need to be modified during program execution and, therefore, cannot pre-exist in ROM. These structures include GDP processor objects, GDP process objects, and other structures discussed in Chapter 3. There must be a way to load the 432's main memory with these structures, and then signal the GDPs to begin their processing. Interface processors and their associated I/O subsystem are used to do this.

Previous sections have discussed the IP in its logical addressing mode. This mode uses the 432's object-oriented addressing and protection mechanism just like the GDP, and is ideal for most operations performed by the IP. There is a problem in trying to use logical addressing for initialization. Logical addressing requires the existence of several data structures in memory (e.g., object references and a physical memory management table). At initialization these structures do not exist, so logical addressing cannot be used. There must be another way to address memory.

The other way to address memory is called physical addressing mode. Physical addressing is similar to logical addressing in several ways. Windows are created into the 432's memory that map addresses from the AP address space. The AP accesses 432 memory by addressing the mapped areas in its address space.

The difference between the logical and physical addressing modes is in how the windows are set up. With logical addressing, you are not concerned about where the object resides in 432 physical memory. When you set up a window, you give the IP a logical address (an object reference) and it figures out where the object is physically located. With physical addressing, the map is set up by telling the IP the desired physical address in 432 memory.

In physical address mode the AP/IP is not constrained by the 432's object-oriented protection mechanism. This allows it to initialize memory with all the structures required to start up the 432 GDPs. After the structures are moved into place in 432 memory by the AP/IP in physical mode, it switches to logical mode and signals the GDPs that they can begin processing.

Physical mode can also be used by an AP/IP subsystem that is performing diagnostic functions. If, for some reason, a failure occurs that damages the data structures used for logical addressing, an AP/IP can be used in physical mode to attempt diagnosis and repair.

The physical mode is a very privileged state of operation because it allows the AP/IP unconstrained access to all of the 432 memory, i.e., it completely overrides all of the object-oriented protection mechanisms. Because of this, there is a mechanism controlling the ability of an AP program to place an IP in physical mode.

To place an IP in physical mode, the IP context must have an object reference for that IP's processor object. Thus, an AP cannot place its IP in physical mode unless the IP context object contains an object reference for the IP's processor object. For example, a multiple-user system can be configured so that only the I/O subsystem used by the system operator has an object reference for its IP processor object and can be placed in physical mode.

In summary, physical mode allows the I/O subsystem to access 432 main memory without the restrictions of the object-oriented addressing and protection mechanism. This is useful for initialization and diagnostics, but must be controlled to insure the integrity of the object-oriented protection mechanism. Therefore, a control mechanism is provided to prevent an IP from accidentally or maliciously being used in physical mode.

AN I/O EXAMPLE

To help you understand how the 432 performs I/O, this section walks through a simple example that illustrates one technique for performing I/O. In the example, a process running on a 432 GDP provides the physical device number and physical address of the data to be read into memory, and a buffer object to hold the incoming data. The I/O subsystem takes it from there -- setting up the peripheral controllers and monitoring the transfer. All higher-level functions, such as managing files, maintaining directories, and translating logical file names into physical device numbers and addresses, are handled by the 432 data processing system.

It is important to note that the amount of functionality placed in the I/O subsystem is a design decision. This example moves minimal functionality to the I/O subsystem to make the example easy to explain. In many applications, the I/O subsystem will provide more functionality; for example, an extremely functional I/O subsystem may manage files, maintain directories, and translate logical file names into physical device numbers and addresses. This off-loads work from the data processing system by allowing it to make I/O requests at a higher level, e.g., READ FILE DATA.F4 instead of READ DEVICE 3 TRACK 5 SECTOR 2.

Chapter 4 presented an example of interprocess communication where several processes running on one or more 432 GDPs communicated with each other by placing messages in in-baskets. The I/O example below also involves interprocess communication, but this time one of the processes is running on a 432 GDP and the other is running on the attached processor in the I/O subsystem.

The GDP process communicates with the I/O process by placing messages in the IP "in-basket." The attached processor uses the receive instructions provided by the IP to receive these messages, which usually request some sort of I/O activity. When the AP finishes the I/O for the GDP process, it sends the GDP process a reply message using the GDP process's in-basket.

The example below has eight different steps. Each step corresponds to one of the frames in Figure 7.8. As you read the example, it is recommended that you flip back and forth between the text and the frames of Figure 7.8 that follow the example.

1. In Figure 7.11.1, the GDP process is ready to ask the I/O subsystem to read a block of data. The message it has prepared for the I/O subsystem tells it to do two things. First, read a block of data from address 0145 in the I/O subsystem and deposit it in a buffer in the 432's main memory. Second, send the buffer as a message to the communication port pointed to by the object reference in the message object. In Figure 7.11.1, the GDP process is executing the SEND instruction, sending the message to the communication port used by the I/O subsystem.

2. Now that the GDP process has made its I/O request by sending the message to the I/O subsystem, it is ready to wait for the reply (i.e., the buffer full of data from address 0145). In Figure 7.11.2, the GDP process is executing a RECEIVE instruction that will cause it to wait in its in-basket until the I/O subsystem sends the reply.

3. In Figure 7.11.3, there are two important things to observe. First, notice that the GDP process has finished executing its RECEIVE instruction and is "inside" the communication port waiting for the I/O subsystem's reply.

   Second, notice that the I/O subsystem is executing a RECEIVE instruction. The AP has commanded the IP to execute a RECEIVE instruction by writing the instruction into the function request area located in the IP's context object. Note that the context object is located in the 432 memory space, and is accessible by the AP because of the mapping facilities provided by the IP. Window 4 is always used to map part of the AP's address space to the context object. This allows the AP to read and write the IP context object by reading and writing the mapped locations in its address space.

4. In Figure 7.11.4, the IP has finished executing the RECEIVE instruction. Note that the IP is now able to access the message object, because the RECEIVE instruction has moved an object reference for the message from the communication port to the IP's context object. The IP has also recorded status information in its context object indicating that the RECEIVE was successful, and has interrupted the AP to let it know that the IP completed the instruction.

5. The AP responds to the IP's interrupt by checking the status information in the context object. The status information tells the AP that the RECEIVE was successful and that the IP is waiting for another instruction.

At this point, the AP can interpret the message to learn what it should do. But, the message is in the 432's address space, which means that the AP software must cause the IP to set up a map so that the AP can read the message.

In Figure 7.11.5, the AP has caused the IP to set up Window 3 to map part of the AP address space to the message object. The AP software now interprets the message and learns that it should read a block of data into the buffer provided and then send the buffer back to the GDP process.

6. The next thing the I/O subsystem needs to do is open a window to the buffer. To do this, the AP first commands the IP to copy the object reference for the buffer object from the message object into the IP's context object. Note that the I/O subsystem can only access this buffer because it now has an object reference for the buffer. The AP then commands the IP to set up Window 2 to map the buffer into the AP address space.

   Now that the buffer is mapped, the AP can proceed with the I/O transfer. AP software accomplishes this by calling an I/O driver for the device requested. The AP and I/O subsystem perform all the functions required to move data from the I/O device to the buffer. Depending on the I/O subsystem configuration, the AP may poll the device, be interrupt-driven, or simply set up a DMA transfer using an 8089.

7. When the AP has finished filling the buffer, it is ready to send the buffer to the GDP process that requested the read. Note that one of the pieces of information in the message sent to the I/O subsystem by the GDP process was an object reference for the communication port where it expected the reply. Figure 7.11.7 shows the IP executing a SEND instruction to send the buffer to the port specified by this object reference.

8. In Figure 7.11.8, the SEND instruction has finished executing. The GDP process has received the buffer and can now proceed with its execution.

Note: Figure 7.11 shows objects being moved about within the 432 memory. This is an illustrative aid only, actually only object references are moved, and the objects themselves (e.g., the buffer object) are <u>not</u> physically relocated.

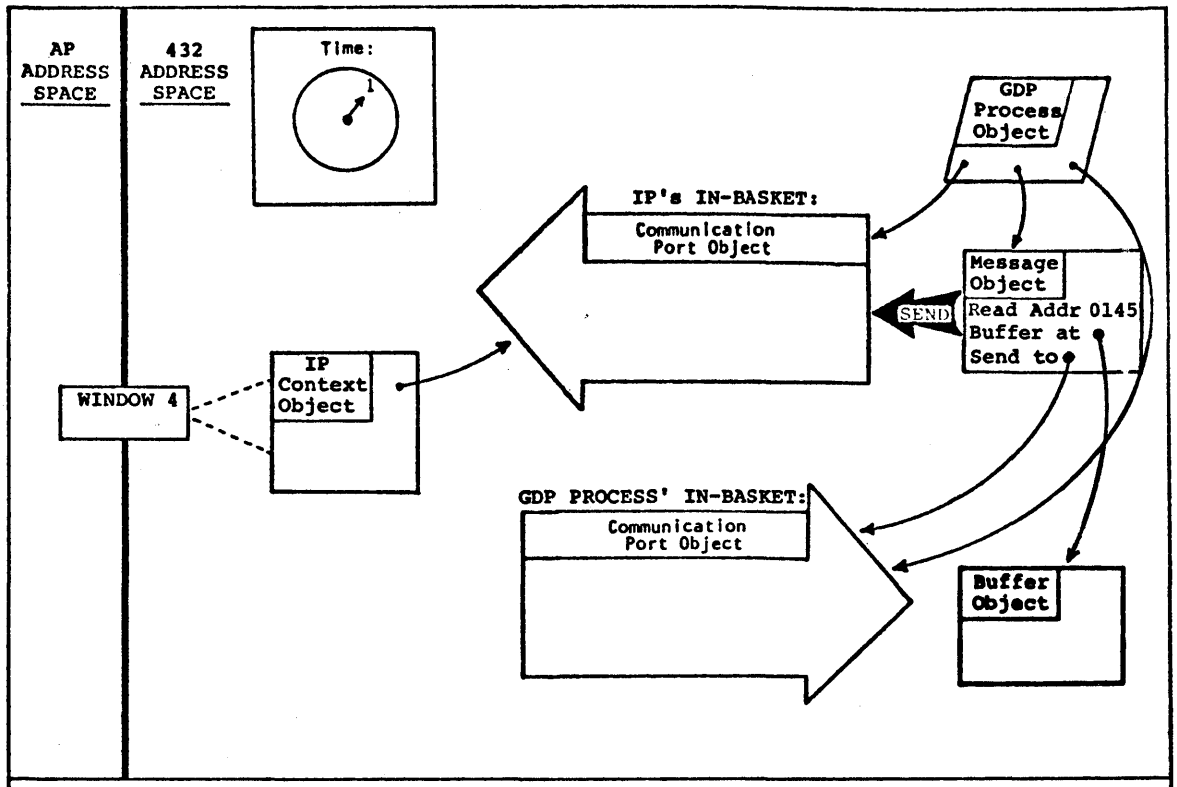


Figure 7.11.1

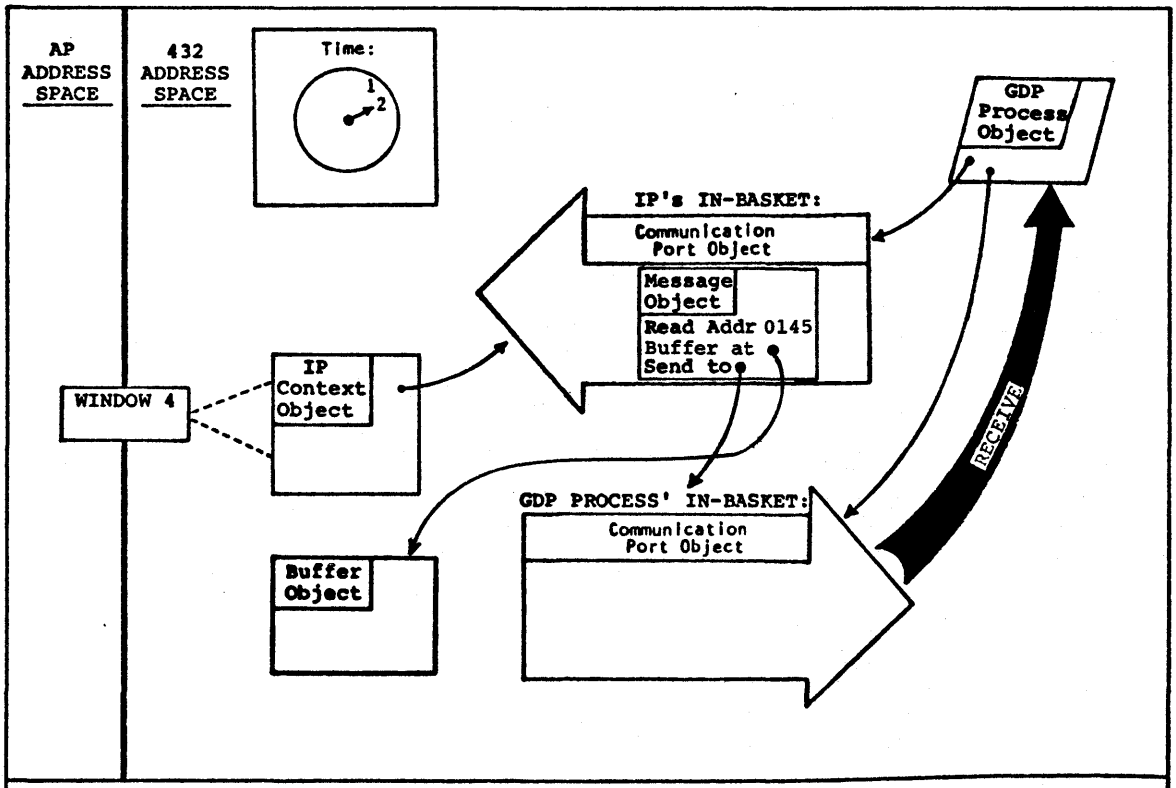


Figure 7.11.2

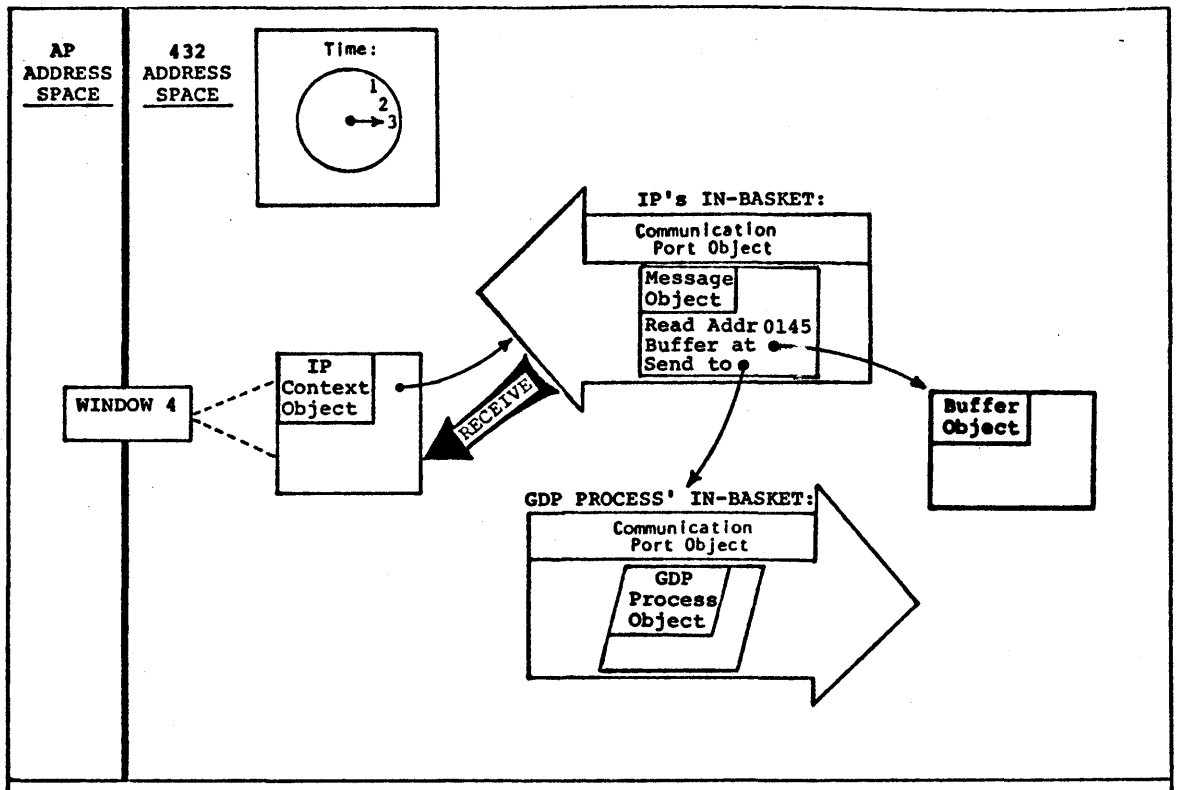


Figure 7.11.3

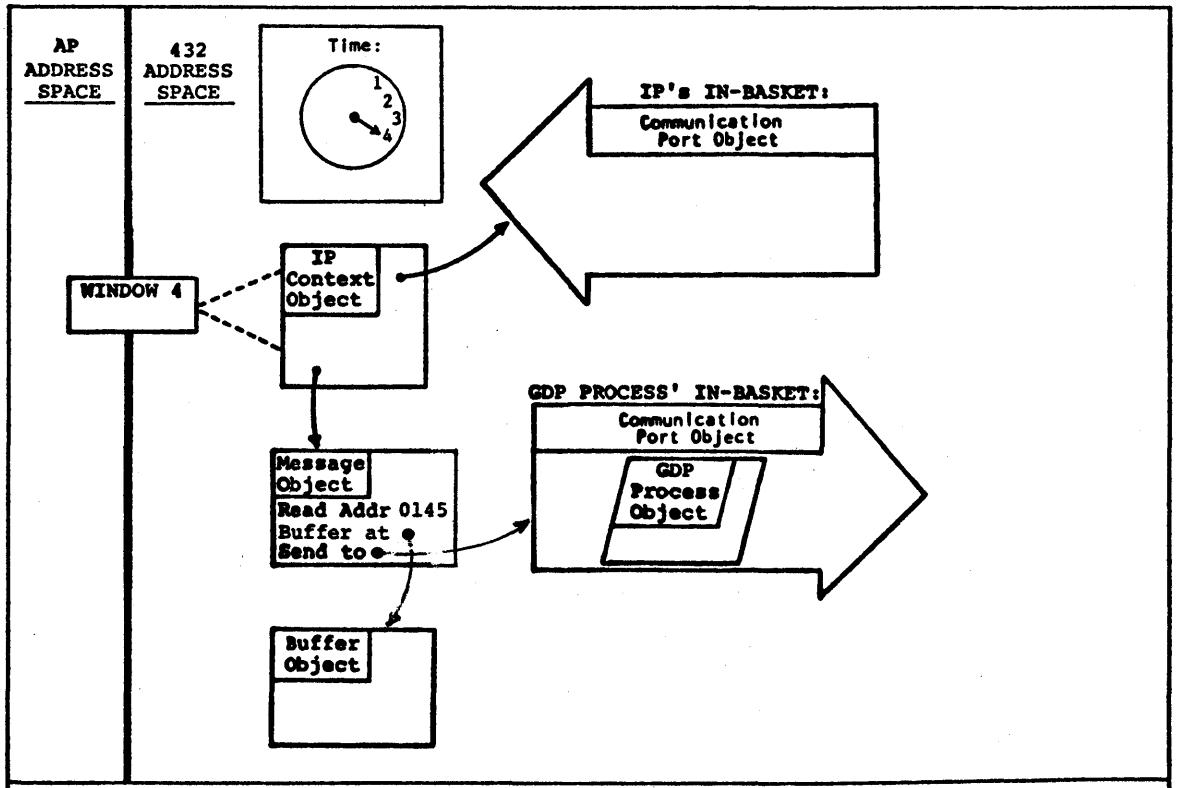


Figure 7.11.4

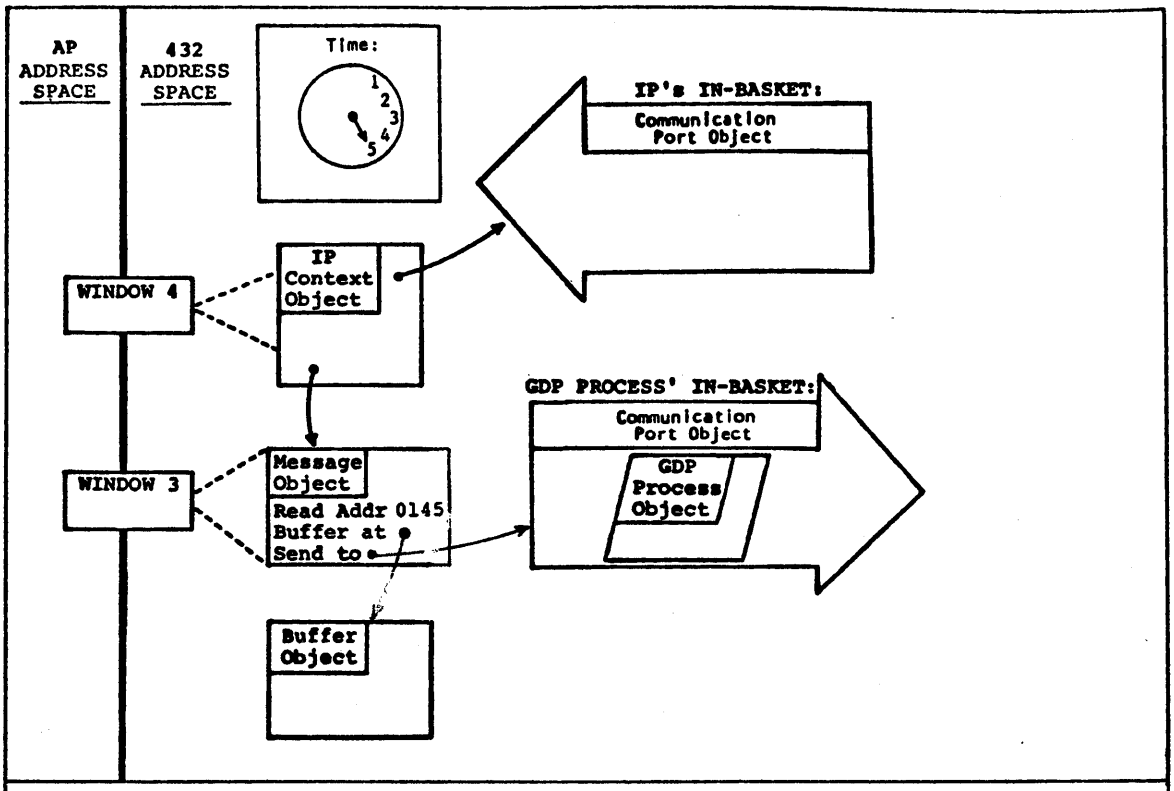


Figure 7.11.5

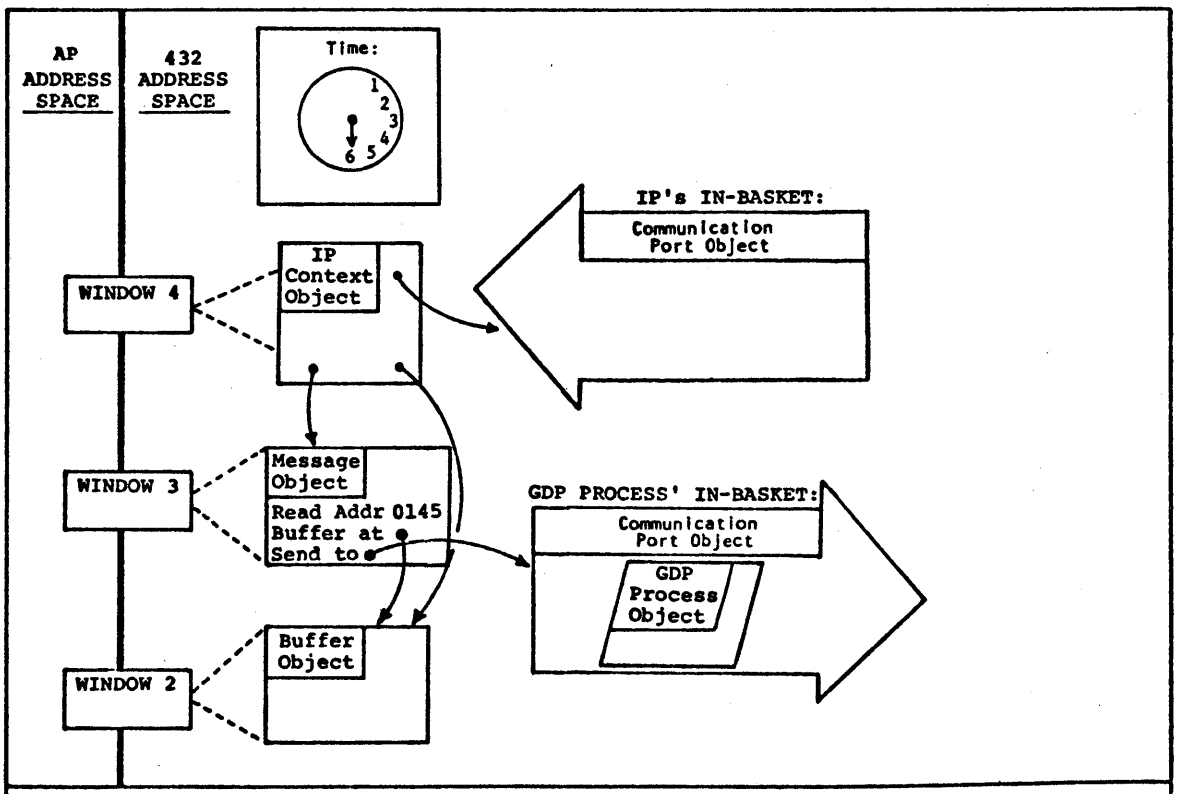


Figure 7.11.6

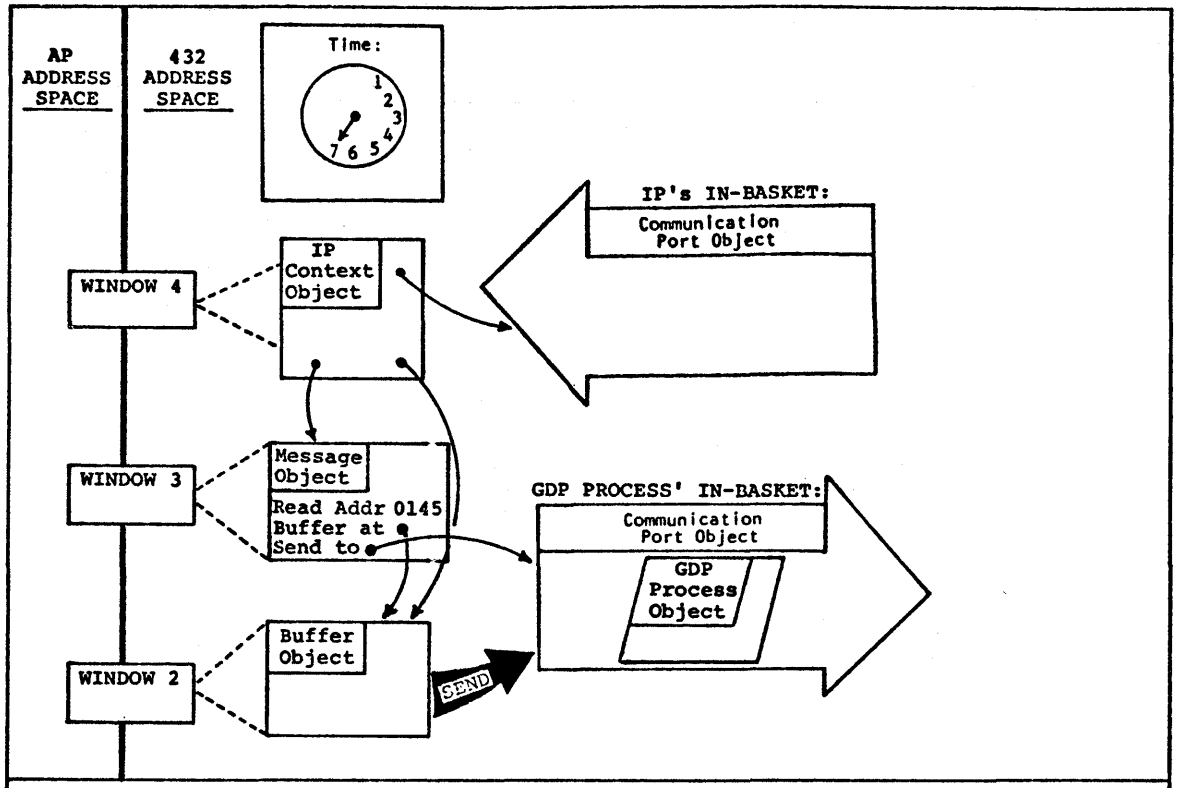


Figure 7.11.7

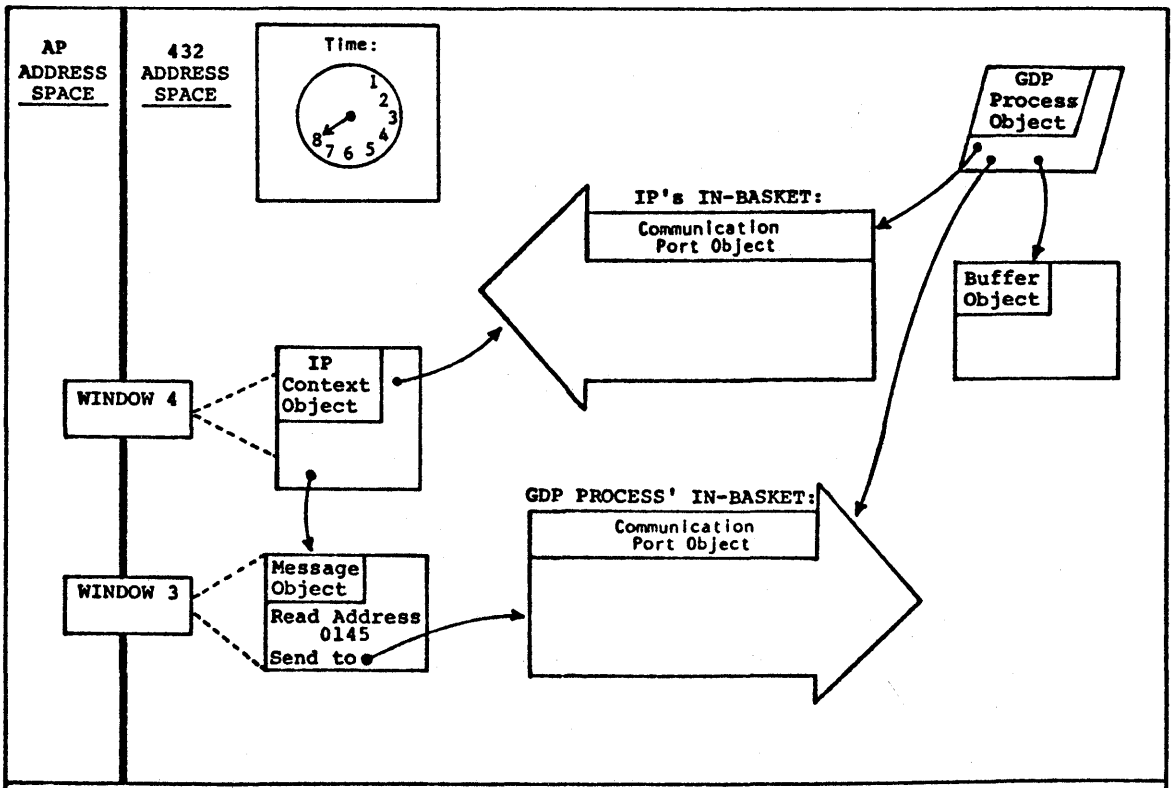


Figure 7.11.8

As a review, take a quick look at what each of the three processors (GDP, IP, AP) does in the example below (see Figure 7.12).

The process running on the 432 GDP does not perform I/O directly. It sends an I/O request to the I/O subsystem (the IP and AP), proceeds asynchronously until it needs the data it requested, and then waits to receive the reply message from the I/O subsystem.

The 432 IP provides services for the software running on the attached processor. It extends the instruction set of the AP with high-level 432 instructions such as SEND and RECEIVE, and provides the AP with protected access to 432 storage.

The AP software is divided into two parts: a peripheral interface executive, and I/O drivers (see Figure 7.12). The executive receives the request message from the GDP process by the message after requesting the IP to set up a map to it; gets access to the buffer by requesting to IP to COPY ACCESS DESCRIPTOR for the buffer into the IP context object; sets up a window to the buffer by asking the IP to execute an UPDATE WINDOW instruction; and then calls the I/O driver for the device requested. The I/O driver sets up the device, monitors the transfer, and returns completion information to the executive. The executive then sends the buffer to the GDP process as a reply by causing the IP to execute a SEND instruction.

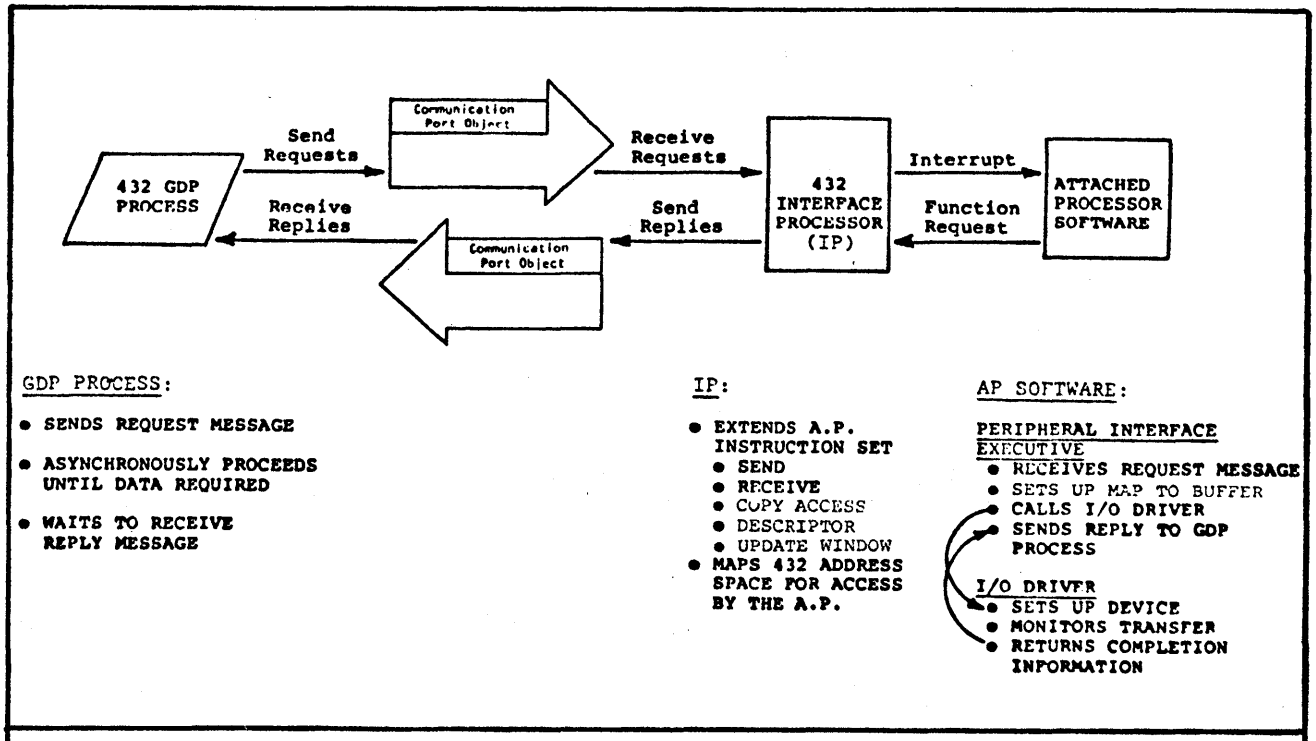


Figure 7.12 -- Coordination of GDP, IP, and AP

The example above describes a very simple situation where a GDP process sends a request to the I/O subsystem. The reverse is also possible, i.e., the I/O subsystem can send a message to a GDP process that asks the GDP process to take some action.

SUMMARY

The 432 micromainframe is divided into two major parts:

- The data processing system made up of one or more general data processors
- One or more I/O subsystems controlled by attached processors

The four major advantages of this organization are:

- Both I/O and data processing performance are extensible.
- I/O subsystems are fully independent and decentralized. They can work in parallel with the data processing system, thus off-loading much of the I/O burden from the generalized data processors.
- System reliability is improved because interrupts are isolated to a relatively small portion of the system.
- Intel's existing line of microprocessors can be connected as attached processors.

Interface processors connect I/O subsystems to the data processing system by providing three important facilities:

- Windows that map addresses from the AP address space into the 432 address space
- Additional instructions that extend the instruction set of the attached processor with object-oriented instructions such as SEND
- Initialization and diagnostic support that enable an I/O subsystem to initialize and diagnose the 432 system

I/O SUBSYSTEM QUIZ

1. What are the three major facilities provided by the interface processor?

2. List three pieces of information contained in an IP context object.

3. How do GDPs perform I/O?

KEY TO I/O SUBSYSTEM QUIZ

1. What are the three major facilities provided by the interface processor?

   - Mapping memory addresses from the AP address space into the 432 address space
   - Extending the AP instruction set with object-oriented instructions such as SEND
   - Initialization and diagnostic support

2. List three pieces of information contained in an IP context object.

   - Object references for all the objects that can be addressed by the I/O subsystem
   - A function request area where the AP writes instructions it wants the IP to execute
   - Status information that tells the AP if the instruction it requested was completed successfully

3. How do GDPs perform I/O?

   They don't. GDP processes send messages to I/O subsystems asking them to do the I/O. The I/O subsystems perform the requested I/O and send reply messages to the GDP processes.

REQUEST FOR READER'S COMMENTS

Intel's Technical Publications Departments attempt to provide publications that meet the needs of all Intel product users. This form lets you participate directly in the publication process. Your comments will help us correct and improve our publications. Please take a few minutes to respond.

Please restrict your comments to the usability, accuracy, readability, organization, and completeness of this publication. If you have any comments on the product that this publication describes, please contact your Intel representative. If you wish to order publications, contact the Intel Literature Department (see page ii of this manual).

1. Please describe any errors you found in this publication (include page number).

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

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

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

5. Please rate this publication on a scale of 1 to 5 (5 being the best rating). _____

NAME _____ DATE _____

TITLE _____

COMPANY NAME/DEPARTMENT _____

ADDRESS _____

CITY _____ STATE _____ ZIP CODE _____
(COUNTRY)

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, WW1-487
3585 SW 198th Ave.
Aloha, OR 97007

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

Printed in U.S.A./Y94/7K/01/04/82/AP