iAPX 432 Interface Processor Architecture Reference Manual

171863-001

INTEL

iAPX 432 INTERFACE PROCESSOR

ARCHITECTURE REFERENCE MANUAL

Manual Order Number 171863-001

Release 1.1 Components

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

The information in this document is subject to change without notice.

Intel Corporation makes no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Intel Corporation assumes no responsibility for any errors that may appear in this document. Intel Corporation makes no commitment to update nor to keep current the information contained in this document.

Intel Corporation assumes no responsibility for the use of any circuitry other than circuitry embodied in an Intel product. No other circuit patent licenses are implied.

The following are trademarks of Intel Corporation and its affiliates and may be used only to identify Intel products:

BXP
CREDIT
i^R
ICE
iCS
i_m
INSITE
Intel^R
int$_e$l^R

Intelevision
Intellec^R
iRMX
iSBC
iSBX
Library Manager
MCS^R
Megachassis
Micromainframe

Micromap
Multibus^R
Multimodule
Plug-A-Bubble
PROMPT
Promware
RMX/80
System 2000^R
UPI
uScope

and the combination of ICE, iCS, iRMX, iSBC, iSBX, MCS, or RMX and a numerical suffix.

PREFACE

Understanding any complex computing system, such as the Intel iAPX 432, requires the assimilation of a great deal of technical information. Before reading this manual on the architecture of the 432 Interface Processor, the reader should have command of the general 432 concepts. Intel offers three documents which provide these prerequisites.

- The INTEL 432 System Summary, Order Number 171867, provides the broad picture of the 432. It should be read as a first introduction to the 432 system.

- The Introduction to the iAPX 432 Architecture, Order Number 171821, restricts discussion to general architecture features which distinguish the 432.

- The iAPX 432 General Data Processor Architecure Reference Manual, Order Number 171860-001, provides detailed information on one type of 432 processor, a General Data Processor (GDP). Its glossary is a concise summary of the most important terminology which is required when reading the Interface Processor manual.

This manual describes another 432 processor, the Interface Processor (IP), similar in many respects to a GDP and different in others. Rather than duplicate all of the general 432 information contained in the companion documents, this manual relies on the above references for descriptions of features of 432 architecture which are common among processors. Unique features and functions of the IP are presented and contrasted with those of the GDP when appropriate.

Chapters 1 through 6 of this manual are composed of descriptions of the Interface Processor, allowing the reader to understand the cooperation between an IP and Peripheral Subsystems in forming a logical I/O processor for a 432 system. Detailed representations for the objects, descriptions of windows and functions, faults, interrupts, and initialization may be found in the appendices.

TITLE PAGE

APPENDICES Page

TABLES

TITLE PAGE

CHAPTER 1
KEY CONCEPTS

This chapter introduces the iAPX 432 Interface Processor (IP). The first four sections cover the IP as it is used normally in connection with input/output operations. Section 1-1 distinguishes Peripheral Subsystems (PS), which are responsible for the bulk of I/O operations, from the 432 data processing system, and shows how Interface Processors link these together. The second section reviews the 432's basic model of input/output, pointing out the need for an interface between a Peripheral Subsystem and the 432 system. Section 1-3 describes the hardware and software that comprise this Peripheral Subsystem interface, with particular emphasis on the role of the IP. In the fourth section the I/O model is summarized and a simple example implementation is reviewed. The final section of the chapter introduces physical reference mode and interconnect addressing, two additional IP facilities that are provided for special situations.

1-1. PERIPHERAL SUBSYSTEMS

A typical application based on the iAPX 432 microprocessor family consists of a 432 system and one or more satellite Peripheral Subsystems. Figure 1-1 illustrates a hypothetical configuration which employs two Peripheral Subsystems. The 432 system hardware is composed of one or more iAPX 432 General Data Processors (GDPs), one or more Interface Processors, and a common memory which is shared by these processors. The 432 system software is a collection of one or more processes which execute on the GDP(s).

A fundamental principle of the 432 architecture is that the 432 system environment is self-contained; neither processors nor processes have any direct contact with the "outside world." Conceptually, the 432 system is enclosed by a wall that protects objects in memory from possible damage by uncontrolled I/O operations.

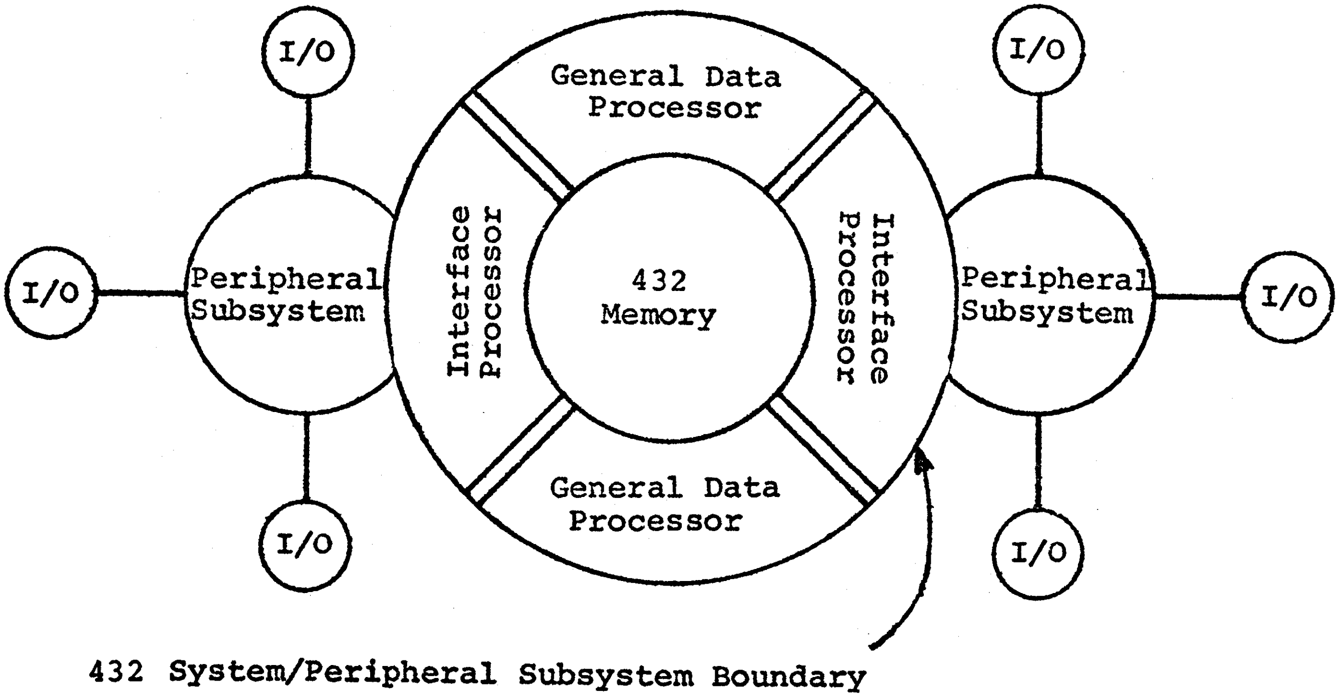


Figure 1-1 432 System and Peripheral Subsystems

In a 432-based system, the bulk of processing required to support input/output operations is delegated to Peripheral Subsystems; this includes device control, timing, interrupt handling and buffering. A Peripheral Subsystem is an autonomous computer system with its own memory, I/O devices and controllers, at least one processor, and software. The number of Peripheral Subsystems employed in any given application depends on the I/O-intensiveness of the application; the number may be varied with changing needs, and is independent of the number of GDPs in the system.

A Peripheral Subsystem resembles a conventional mainframe channel in that it assumes responsibility for low-level I/O device support and executes in parallel with 432 system processor(s). Unlike a simple channel, however, each Peripheral Subsystem can be configured with a complement of hardware and software resources that precisely fits application cost and performance requirements. In general, any system that can communicate over a standard 8- or 16-bit microcomputer bus, such as Intel's Multibus™ design, may serve as a 432 Peripheral Subsystem.

A Peripheral Subsystem is attached to the 432 system by means of an iAPX 432 Interface Processor (IP). At the hardware level, an Interface Processor presents two separate bus interfaces. One of these is the standard 432 processor packet bus and the other is a very general interface that can be adapted to most traditional 8- and 16-bit microcomputer buses.

The Interface Processor is driven by Peripheral Subsystem software. To support the transfer of information through the wall that separates a Peripheral Subsystem from the 432 system, the IP provides a set of software-controlled windows. A window is used to expose a single object (data structure) in 432 system memory so that its contents may be transferred to or from the Peripheral Subsystem. To preserve the integrity of the capability-based protection mechanisms in the 432 system, the IP only provides the PS with windowed access to 432 objects which are of system type data segment.

An Interface Processor additionally provides a set of functions, which are also invoked by Peripheral Subsystem software. While the operation of these functions (and the returned results) varies considerably, they generally permit objects in 432 system memory to be manipulated as entities, and enable communication between 432 system processes and software executing in a Peripheral Subsystem.

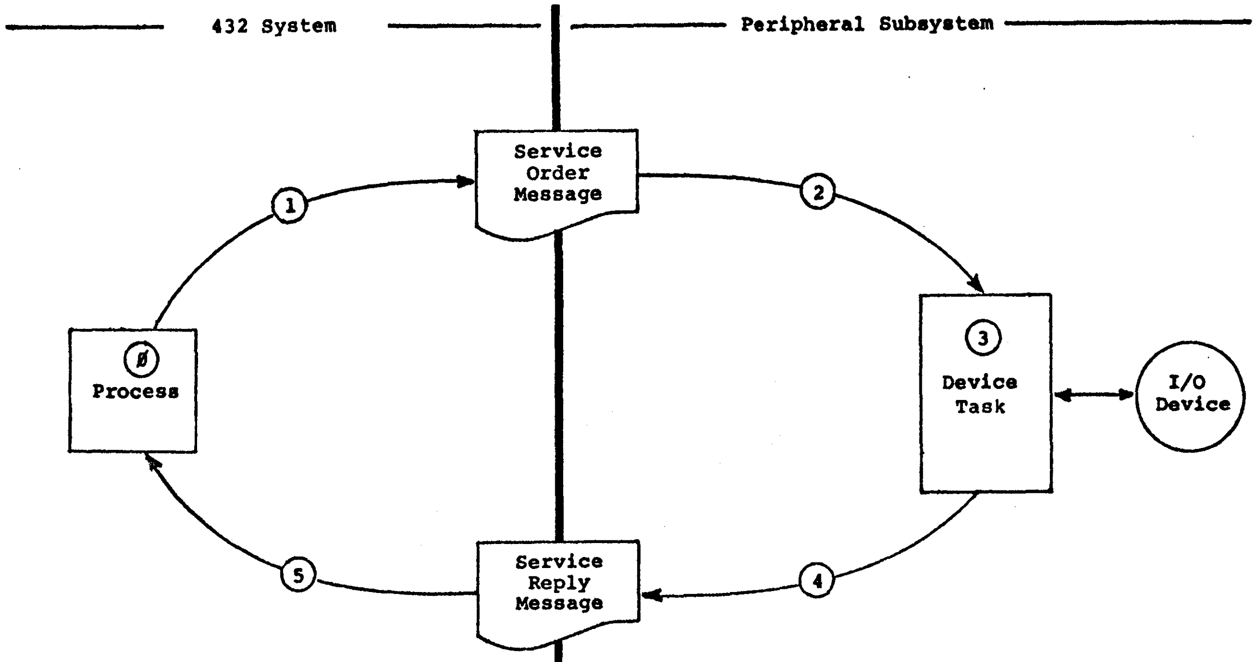
It is important to note that both the window and function facilities utilize and strictly enforce the standard 432 addressing and protection systems. Thus, a window provides protected access to an object, and a function provides a protected way for Peripheral Subsystem software to interact with the 432 system.

1-2. BASIC I/O MODEL

As figure 1-2 illustrates, input/output operations in a 432 system are based on the notion of passing messages between 432 system processes and device tasks located in a Peripheral Subsystem. In this manual, a device task is considered to be the hardware and software in the Peripheral Subsystem which is responsible for managing an I/O device. An I/O device is considered to be either a consumer or producer of data. Thus an I/O device may be a real device (e.g., a terminal), a file, or a pseudo-device (e.g., a spooler).

A message sent from a GDP process which requests I/O service contains information that describes the requested operation (e.g., "read file XYZ"). The device task interprets the message and carries out the operation. If an operation generates input data, the device task returns the data as a message to the originating process. The device task may also return a message to positively acknowledge completion of a request.

A very general and very powerful mechanism for passing messages between processes is inherent in the 432 architecture. A given Peripheral Subsystem may, or may not, have its own message facility, but in any case, such a facility will not be directly compatible with the 432's. By interposing a Peripheral Subsystem interface at the subsystem boundary, the standard 432 interprocess communication system can be made compatible with any device task (see figure 1-3).



0. Process running on GDP needs I/O service
1. Process formulates message describing service, sends it to device task
2. Device task receives service order, interprets it
3. Device task transfers data according to service order parameters
4. Device task formulates reply message containing result of transfer operation, sends it back to originating process.
5. Originating process receives reply, interprets it, executes accordingly

Figure 1-2 Basic I/O Service Cycle

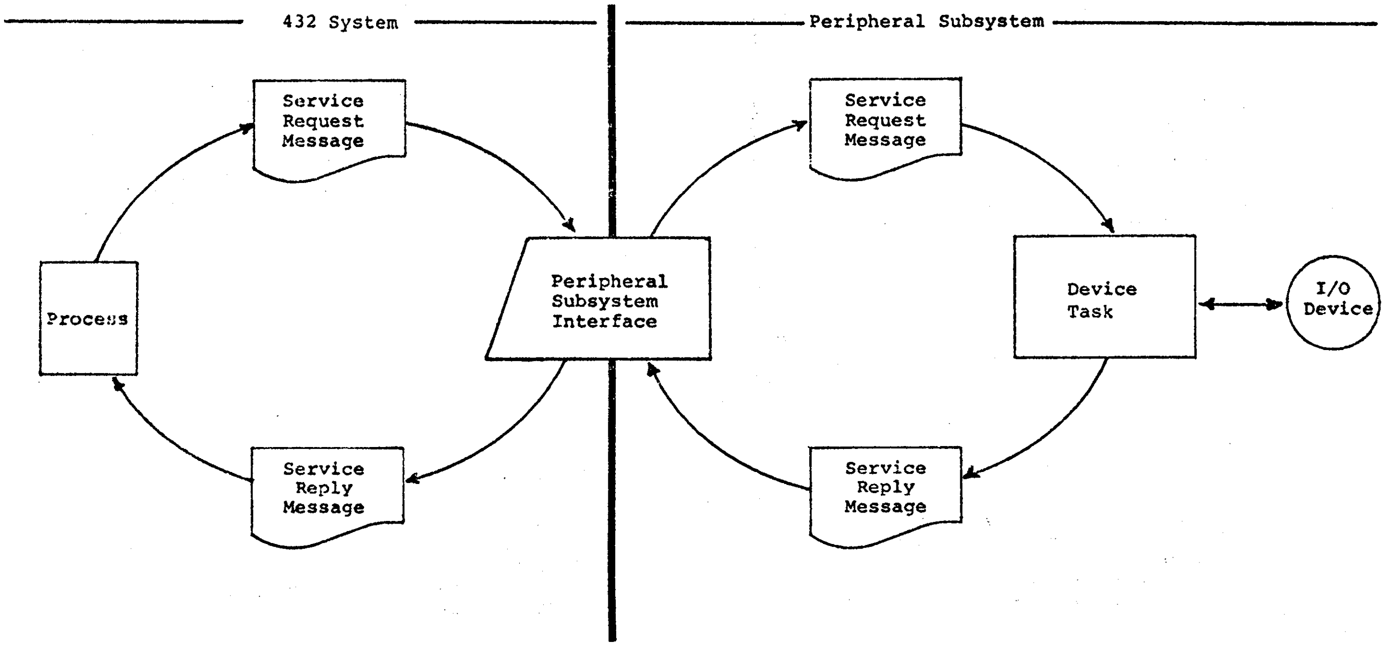


Figure 1-3 Peripheral Subsystem Interface

1-3. PERIPHERAL SUBSYSTEM INTERFACE

A Peripheral Subsystem interface is a collection of hardware and software that acts as an adaptor which enables message-based communication between a process in the 432 system and a device task in a Peripheral Subsystem. Viewed from the 432 side, the Peripheral Subsystem interface appears to be a set of processes. The Peripheral Subsystem interface may be designed to present any desired appearance to a device task. For example, it may look like a collection of tasks.

PERIPHERAL SUBSYSTEM INTERFACE HARDWARE

The Peripheral Subsystem interface hardware consists of a 432 Interface Processor, an Attached Processor (AP), and memory (see figure 1-4). To improve performance, these may be augmented by a DMA controller. The AP and the IP provide complementary facilities. Considered as a whole, the AP/IP pair may be thought of as a logical I/O processor, which supports software operations in both the 432 system and the Peripheral Subsystem.

ATTACHED PROCESSOR

Most any general-purpose processor, such as an 8085, an iAPX 86 or an iAPX 88, can be used as an Attached Processor. The AP need not be dedicated exclusively to working with the Interface Processor. It may, for example, also execute device task software or user applications. Thus, the AP may be the only processor in the Peripheral Subsystem, or it may be one of several. To insure synchronization and coordination, in Peripheral Subsystems with multiple processors, only one of these should be designated to serve as the AP. Other processors (or active agents, such as DMA controllers) may be given access to IP windows, but control of the Interface Processor should be centralized in the Attached Processor.

As figure 1-4 shows, the AP is "attached" to the Interface Processor in a logical sense only. The physical connections are standard bus signals and one interrupt line (which would typically be routed to the AP via an interrupt controller).

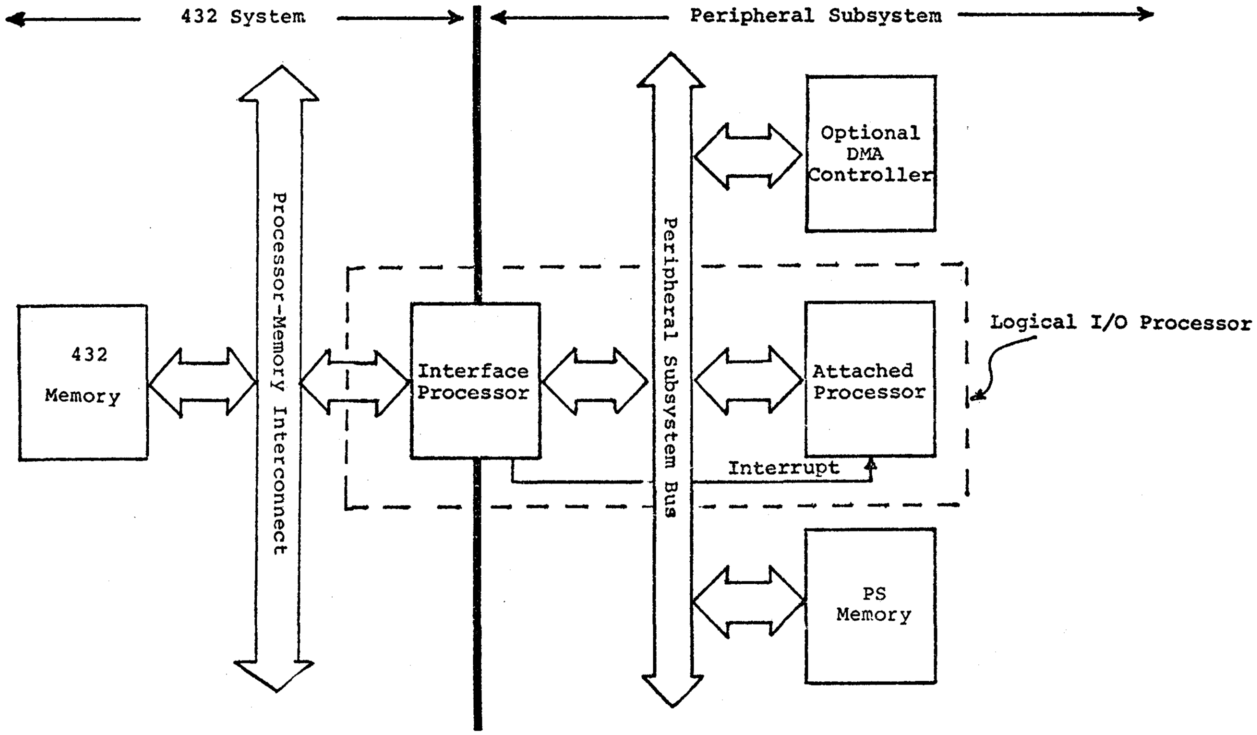


Figure 1-4 Peripheral Subsystem Interface Hardware

Continuing the notion of the logical I/O processor, the Attached Processor fetches instructions, provides the instructions needed to alter the flow of execution, and performs arithmetic, logic and data transfer operations within the Peripheral Subsystem.

INTERFACE PROCESSOR

The IP completes the logical I/O processor by providing data paths between the Peripheral Subsystem and the 432 system. The IP also provides functions which effectively extend the AP's instruction set so that software running on the logical I/O processor can operate in the 432 system. Since these facilities are software-controlled, they are discussed in the next section.

As figure 1-4 shows, the Interface Processor presents both a Peripheral Subsystem bus interface and a standard 432 processor packet bus interface. By bridging the two buses, the IP provides the hardware link that permits data to flow between the 432 system and the Peripheral Subsystem.

The Interface Processor connects to the 432 system in exactly the same way as a GDP. Thus, in addition to being able to access 432 memory, the IP supports other 432 hardware-based facilities, including interprocessor communication, alarm signaling and functional redundancy checking.

On the I/O subsystem side, the IP provides a very general bus interface that can be adapted to any standard 8- or 16-bit microprocessor bus, including Intel's Multibus™ architecture, as well as the component buses of the MCS-85 and iAPX 86 families. The IP is connected to the Peripheral Subsystem bus as if it were a memory component; it occupies a block of memory addresses up to 64k bytes long. Like a memory, the IP behaves passively within the Peripheral Subsystem (except as noted below). It is driven by Peripheral Subsystem memory references that fall within its address range.

The IP generally responds like a memory component. The Interface Processor also supplies an interrupt signal. The Interface Processor uses this line to notify its Attached Processor that an event has occurred which requires its attention. Interrupt handling software on the AP may read status information provided by the IP to identify the nature of the event.

To summarize, the Attached Processor and the Interface Processor interact with each other by means of address references generated by the AP and interrupts generated by the IP. Since the Interface Processor responds to memory references, other active Peripheral Subsystem agents (bus masters), such as DMA controllers, may obtain access to 432 system memory via the IP's windows.

PERIPHERAL SUBSYSTEM INTERFACE SOFTWARE

I/O CONTROLLER

The Peripheral Subsystem interface is managed by software, which this manual refers to as the I/O controller. The I/O controller executes on the Attached Processor and uses the facilities provided by the AP and the IP to control the flow of data between the 432 system and the Peripheral Subsystem.

432 hardware imposes no constraints on the structure of the I/O controller. To help simplify software organization and modification, implementors may wish to consider organizing it as a collection of tasks running under the control of a multitasking operating system (such as iRMX-80™, iRMX-88™, or iRMX-86™). This type of organization supports asynchronous message-based communication within the I/O controller, similar to the 432's intrinsic interprocess communication facility. Extending this approach to the device task as well results in a consistent, system-wide communication model. However, communication within the I/O controller and between the I/O controller and device tasks, is completely application-defined. It may also be implemented via synchronous procedure calls, with "messages" being passed in the form of parameters.

However it is structured, the I/O controller interacts with the 432 system through facilities provided by the Interface Processor. There are three of these facilities: execution environments, windows, and functions.

EXECUTION ENVIRONMENTS

The Interface Processor provides a process addressing environment within the 432 system which supports the operation of the I/O controller in the 432 system. This environment is embodied as a set of system objects that are used and manipulated by the IP. At any time, the I/O controller is represented in 432 memory by IP process objects and associated context objects. Like a GDP, the IP itself is represented by a processor object. Representing the IP and its controlling software like this creates an execution environment that is analogous to the environment of a process running on a GDP. This environment provides a standard framework for addressing, protection and communication within the 432 system.

Like a GDP, an IP supports multiple process environments. The I/O controller selects the environment in which a function is to be executed. This permits, for example, the establishment of separate environments corresponding to individual device processes in the Peripheral Subsystem. If an error occurs while the IP controller is executing a function on behalf of one device task of the I/O controller, that error is confined to the associated process, and processes associated with other device tasks are not affected.

WINDOWS

Every transfer of data between the 432 system and a Peripheral Subsystem is performed via an IP window. A window defines a correspondence, or mapping, between a subrange of consecutive Peripheral Subsystem memory addresses (within the range of addresses occupied by the IP) and an object of system type data segment in 432 system memory (see figure 1-5). When an agent in the Peripheral Subsystem (e.g., the IP controller) reads a windowed address, it obtains data from the associated object; writing into a windowed address transfers data from the Peripheral Subsystem to the windowed object. The action of the IP, in mapping the Peripheral Subsystem address to the system object, is transparent to the agent making the reference. As far as it is concerned, it is simply reading or writing memory.

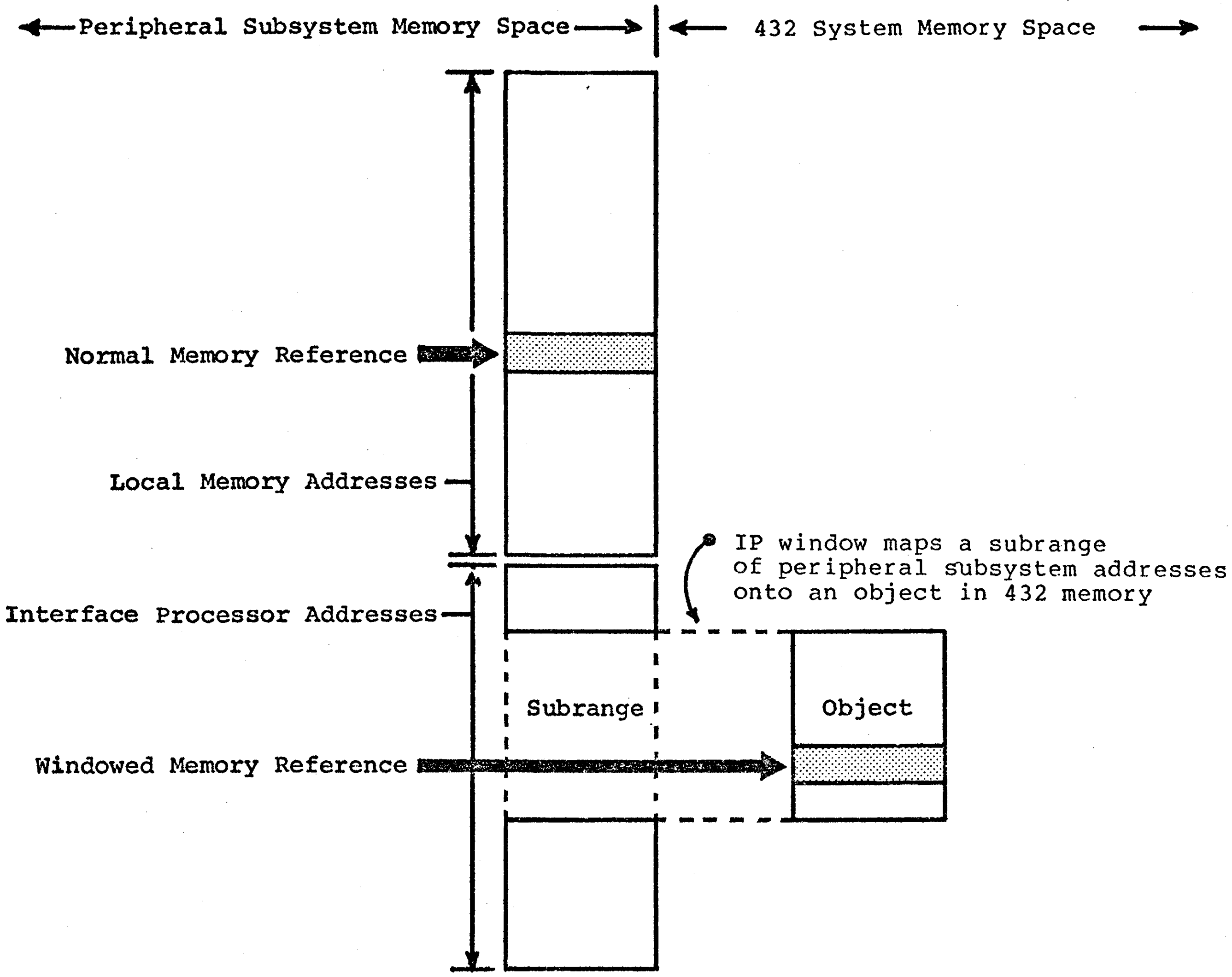

Peripheral Subsystem Memory Space
432 System Memory Space
Normal Memory Reference
Local Memory Addresses
IP window maps a subrange of peripheral subsystem addresses onto an object in 432 memory
Interface Processor Addresses
Subrange
Object
Windowed Memory Reference

Figure 1-5 Interface Processor Window

Since a window is referenced like memory, any individual transfer may be between an object and PS memory, an object and a PS processor register, or an object and an I/O device. The latter may be appealing from the standpoint of "efficiency," but it should be used with caution. Using a window to directly "connect" an I/O device and an object in 432 memory has the undesirable effect of propogating the real-time constraints imposed by the device beyond the subsystem boundary into the 432 system. It may seriously complicate error recovery as well. Finally, since there is a finite number of windows, most applications will need to manage them as scarce resources which will not always be instantly available. This means that at least some I/O device transfers will have to be buffered in PS memory until a window becomes available. It may be simplest to buffer all I/O device transfers in memory, and use the windows to transfer data between PS memory and 432 system memory.

There are four IP windows which may be mapped onto four different objects. The I/O controller may alter the windows during execution to obtain access to different objects. References to windowed subranges may be interleaved and may be driven by different agents in the Peripheral Subsystem. For example, the Attached Processor and a DMA controller may be driving transfers concurrently, subject to the same bus arbitration constraints that would apply if they were accessing memory.

FUNCTIONS

A fifth window, the control window, provides the IP controller with access to the Interface Processor's function request facility. The IP controller requests the execution of an IP function by writing operands and an opcode into predefined locations in the control window's subrange. This procedure is very similar to writing commands and data to a memory-mapped peripheral controller (e.g., floppy disk controller). Upon completion of the function, the IP interrupts the AP and provides status information which the IP controller can read through the control window. The IP can respond to transfer requests to the other four windows while it is executing a function. In addition, data transfers through windows 0 through 3 may be interleaved with function request sequences through the control window.

The IP's function set permits the I/O controller to:

- o alter windows;
- o exchange messages with GDP processes via the standard 432 interprocess communication facility;
- o manipulate objects.

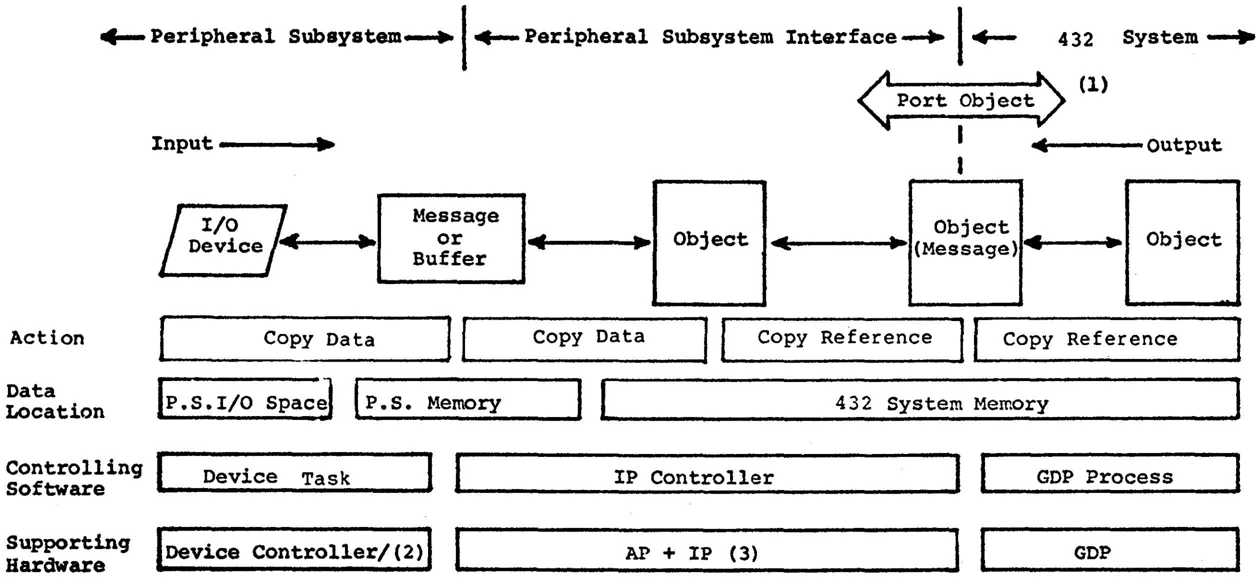
These functions may be viewed as extensions to the Attached Processor's instruction set, which permit the I/O controller to operate in the 432 system.

The combination of the IP's function set and windows, the AP's instruction set, and possibly additional facilities provided by a Peripheral Subsystem operating system, permits great flexibility in designing I/O models. By using the more sophisticated IP functions, powerful I/O controllers can be built which are capable of relieving the 432 system of much I/O-related processing. On the other hand, by utilizing only a subset of the available IP functions, relatively simple I/O controllers can also be constructed.

1-4. I/O MODEL SUMMARY

DATA FLOW SUMMARY

Figure 1-6 summarizes the relationship of the hardware and software components that cooperate to move data between an I/O device and 432 system memory. Notice how the Peripheral Subsystem interface not only bridges the 432 system/Peripheral Subsystem boundary, but also can "hide" the characteristics of the one from the other. As far as a device task is concerned, its job is to move data between memory and an I/O device; it may be completely unaware that it is connected to a 432 system. This means that existing device tasks may be utilized in a 432 system with little or no modification, and that programmers working on device tasks need not be trained in the operation of the 432. Similarly, a GDP process which needs an I/O service need have no knowledge of the details and characteristics of the target I/O device. As far as it is concerned, it "performs" I/O in the same way it communicates with a co-operating process; by sending and receiving messages via the standard 432 interprocess communication facility.



Notes:
(1) Only object reference is moved to and from port.
(2) Supporting processor is defined by application; may be AP, a separate processor, may include a DMA controller.
(3) May also include a DMA controller.

Figure 1-6 I/O Data Flow Summary

I/O EXAMPLE

To illustrate the operation of the 432 I/O model more specifically, this section provides a simple example which shows how line printer output might be implemented. Of course, the example describes only one of many possible approaches that might be taken. Furthermore, the example does not show all the detail of a typical implementation, with the Peripheral Subsystem supporting transfers to and from a number of devices concurrently.

In this example, all Peripheral Subsystem software is assumed to be implemented as a collection of tasks running under the control of a multitasking operating system. This OS is assumed to allow tasks to communicate with one another in a fashion that is analogous to the 432 interprocess communication facility. The mechanisms provided by the OS are messages, mailboxes, a TRANSMIT operator and an ACCEPT operator. Messages are arbitrary data structures in memory, and mailboxes are queue structures that hold tasks waiting for messages or messages waiting for tasks. When executed by a task, TRANSMIT moves a message from a task to a mailbox and ACCEPT moves a message from a mailbox to the issuing task if a message is available; if not, the task is queued at the mailbox until another task TRANSMITs a message to the mailbox. In other words, mailboxes are analogous to 432 ports and TRANSMIT and ACCEPT are analogous to the 432 SEND and RECEIVE operators.

Figure 1-7 shows the overall structure of the example system and the flow of data from one element to another (see also table 1-1). Basically, a GDP process wishing to print data on the line printer sends a message containing the data to the Peripheral Subsystem task which controls the printer; when the data has been printed, the printer task returns the message as a positive acknowledgement to the originating process. The process may then send more data by writing it into the message and sending it off again. In practice, there might be a pool of these messages, with several cycling through the system at one time.

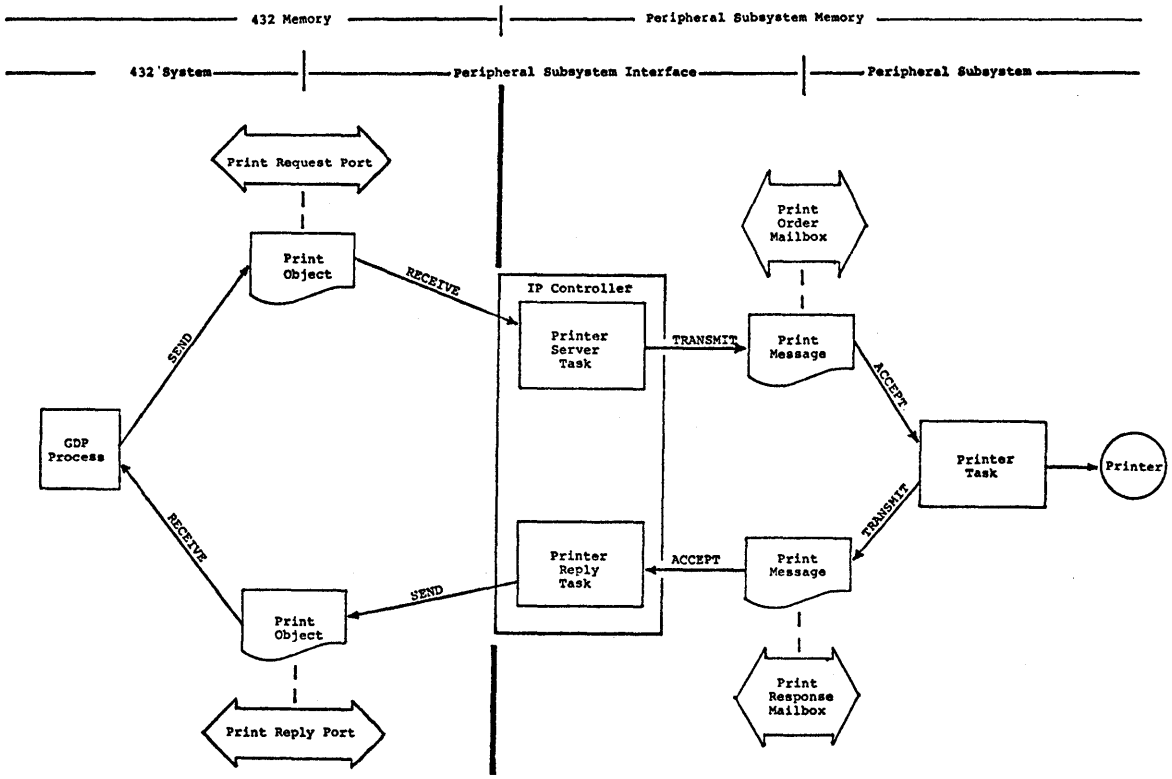


Figure 1-7 Printer Example

Table 1-1 Printer Example Legend

Item	Description
Print_object	Object (message) describing print operation from requesting process's point of view (see figure 1-8).
Print_request_port	432 communications port assigned by convention to queue print objects.
Print_reply_port	432 communications port where GDP process waits for result of operation.
SEND/RECEIVE	432 operators (GDP instructions, IP functions) provided for interprocess communication.
Print_order_mailbox	OS message queue defined to hold print messages waiting for printer task.
Print_response_mailbox	OS message queue defined to hold print messages already processed by the printer task.
TRANSMIT/ACCEPT	OS operators analogous to 432 SEND and RECEIVE operators.

Figure 1-8 shows how the message sent by the GDP process might be organized. It consists of two parts, an object reference part and a text part. The object references are for the text part of the object, the 432 port at which the process will wait for the message to be returned, and a reference for the process itself (GDP or IP). This last reference is not strictly necessary in the example, but is provided to show one way in which a message may identify its originator.

The text part of the message contains a command field which specifies what is to be done (e.g., print one page), a status field which reflects the disposition of the print request, and the data to be printed.

With the exception of the status information, all data in the message is provided by the GDP process; the status field is updated by the printer task.

The next three sections describe the operation of the example system as seen by the GDP process, the printer task, and the IP controller. These descriptions present an overview of the operations. For more detail on how these activities relate to IP facilities, please refer to Appendix F, (Interprocess Communication Example), which refines the printer example.

GDP Process Perspective

To direct output to the line printer, a GDP process builds a print object and sends it as a message to the print_request_port. The port is the process's "connection" to the line printer. After it has sent the message, the process is free to continue running. When it cannot proceed further without acknowledgement of the print operation, the process attempts to receive a message from the print_reply_port it specified in the print_object. When the operation has been completed, the process will receive the message. It then inspects the status field and takes appropriate action, perhaps writing new data into the print_object and sending it off again.

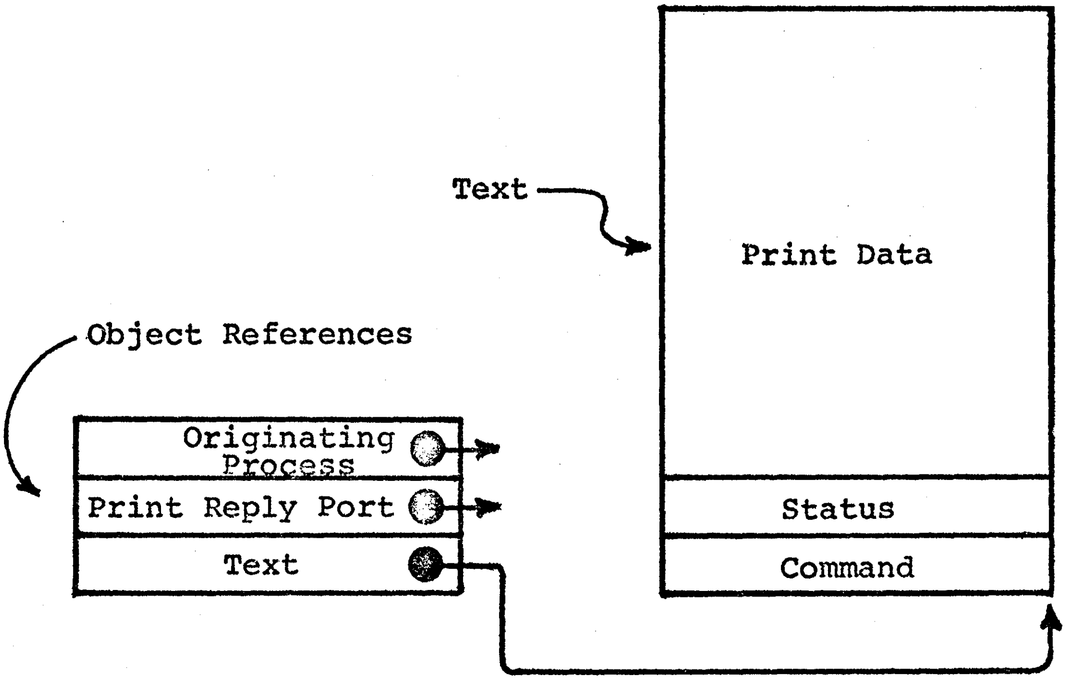


Figure 1-8 Example Print Object

Printer Server Task Perspective

The printer server task may be viewed as a "front end" to the printer task which is responsible for translating the message sent by the GDP process into the form expected by the printer task. The printer server loops through the following steps:

1. RECEIVE a message from the print_request_port.
2. When the message (a print object) is received, obtain an object selector for the message text.
3. Using the object selector, open a window onto the message text.
4. Copy the message text from 432 memory to PS memory through the open window.
5. Close the window.
6. TRANSMIT a message with a reference to the print text to the printer task.
7. Repeat from step 1.

Printer Task (Device Task) Perspective

The printer task runs in an endless loop repeating the following steps:

1. ACCEPT a message from the print_order_mailbox;
2. Interpret the message;
3. Transfer the data from the message to the printer, taking care of all device control (e.g., interrupts);
4. Update the status field of the print_message with the result of the operation;
5. TRANSMIT the updated print_message to the print_response mailbox;
6. Repeat from step 1.

Printer Reply Task Perspective

The printer reply task may be viewed as a "back end" to the printer task. It runs in an endless loop as follows:

1. ACCEPT a message from the print_response_mailbox.
1. Open a window onto the print_object in the 432 system.
2. Formulate a print_reply_message and deposit it in the print object through the open window.
3. Close the window.
4. SEND the print_object to the printer reply port in the 432 system.
5. Repeat from step 1.

1-5. SUPPLEMENTARY INTERFACE PROCESSOR FACILITIES

The preceding sections have described the Interface Processor as it is used most of the time. The IP provides two additional capabilities which are typically used less frequently, often only in exceptional circumstances. These are physical reference mode and interconnect access.

PHYSICAL REFERENCE MODE

An IP normally operates in logical reference mode. This mode is characterized by its object-oriented addressing and protection system. When an IP running in logical mode opens a window, it utilizes an object selector to specify a particular 432 data segment. There are times when logical referencing is impossible because the objects used by the hardware to perform logical-to-physical address development are absent (or, less likely, are damaged). In these situations the IP can be used in physical reference mode.

An IP which is operating in physical reference mode circumvents the protection mechanisms of the 432 system. No distinction is made between data segments and access segments in physical reference mode. The IP provides a reduced set of functions in this mode. Windows map directly onto contiguous segments of 432 physical memory (rather than object structures in 432 memory). The IP controller selects a segment by specifying a 24-bit physical address when it establishes a window. The IP interprets subsequent subrange references as 16-bit displacements from the segment's base address. This simple base-plus-displacement addressing is similar to traditional computer addressing techniques.

Physical reference mode is most often employed during system initialization to load images of objects from a Peripheral Subsystem into 432 memory. Once the required objects are available, processors can begin normal logical reference mode operations. Logical mode cannot be used until the object tables required for logical-to-physical address translation have been constructed and loaded into 432 memory.

INTERCONNECT ACCESS

In addition to memory, the iAPX 432 architecture defines a second, independent address space called the processor-memory interconnect address space. The interconnect address space allows interconnect objects to be maintained which may contain one or more interconnect registers. Interconnect registers are double byte quantities which are aligned on double byte boundaries. With the exception of a few reserved addresses, the definition and use of interconnect locations is not pre-defined for the IP. Appendix E of this manual suggests how the interconnect may be utilized during the initialization of variable-configuration systems.

The IP (like a GDP) requires two register locations in the interconnect space to be defined for any system:

- o the processor ID register (interconnect address 0)
- o the interprocessor communication (IPC) register (interconnect address 2)

The remainder of the interconnect address space may be used to store or acquire other information such as configuration parameters, error logging registers, and other application-specific quantities.

Window 1 is software-switchable between the memory and the interconnect spaces. In logical reference mode, the interconnect space is addressed in the same object-oriented manner as the memory space, with the IP automatically performing the logical-to-physical address development. To access the interconnect space, the I/O controller must specify an object selector for an interconnect object which exposes a segment of the interconnect space to the IP. The normal window addressing scheme is then used to locate individual interconnect registers within the object. Switching window 1 to interconnect access mode gives the IP access to interconnect objects. This operation is equivalent to the MOVE TO INTERCONNECT and MOVE FROM INTERCONNECT operators of the GDP.

In physical reference mode, the interconnect space is addressed as a linear array of even-addressed, double-byte, interconnect registers. As with physical reference mode memory accesses, the switchable window is set up with a 24-bit physical base address. Peripheral subsystem references to the corresponding subrange are likewise interpreted by the IP as 16-bit displacements from the base address to individual interconnect registers.

CHAPTER 2
OBJECTS AND OPERATORS

This chapter describes the 432 environment as it appears to the I/O controller software. It points out what the I/O controller can, and cannot, do in the 432 system. The first section broadly compares the facilities provided by the Interface Processor to those available on the General Data Processor. The remaining sections describe Interface Processor facilities provided for:

- o addressing and protection;
- o objects for program environments;
- o facilities for asynchronous communication;
- o processes and storage resource management;
- o facilities for process scheduling and dispatching.

Because a great many facilities are common to both processors, this chapter adopts the approach of describing IP facilities that are different or unique, and referring the reader to the iAPX 432 General Data Processor Architecture Reference Manual, (Order Number 171860-001) for descriptions of identical features.

2-1. SUMMARY OF IP FACILITIES

This section surveys the Interface Processor by comparing it to the General Data Processor. When reading this section, it is useful to recall the notion, introduced in chapter 1, of the AP/IP pair co-operating as a logical I/O processor. In this arrangement, the Attached Processor fetches instructions, provides arithmetic, logical, and flow-of-control operations, and generates Peripheral Subsystem address references. The Interface Processor completes the logical I/O processor by supplying the facilities for operation within the 432 system, plus the window mechanism for transferring data between the two systems. Windows are discussed in detail in chapter 3.

To permit the I/O controller to function in the 432 system as well as in the Peripheral Subsystem, the IP provides an environment, and operators that it executes within this environment. The environment is embodied in the system objects that the Interface Processor recognizes and manipulates, while the operators take the form of function requests issued by the IP controller and executed by the IP. (Like a GDP, the IP performs other operations in response to interprocessor communications; these are normally transparent to the AP, however.)

The standard 432 object-oriented addressing and protection systems underlie all IP facilities. The IP uses the same descriptor-controlled, segment-based address development mechanism as the GDP. It performs type and rights checking identically. Addressing and protection apply to both the transfer of data through windows and the execution of functions.

ENVIRONMENT

Table 2-1 lists all 432 system objects and compares the IP's implementation of them with that of the GDP. For the most part these objects are handled identically by both machines; the variances noted in the table stem from the different orientation and design of the two machines. The IP does not implement instruction segments, for example, because its Attached Processor takes care of instruction fetching.

IP processor, process and context objects are similar in purpose to the corresponding GDP structures, but are implemented somewhat differently. Importantly, the processor and process objects are compatible with the standard 432 processor and interprocess communication facilities. The IP supports multiple process environments; a separate process can be associated, for example, with each Peripheral Subsystem device task. Each process has a single context object which defines the process's current access environment (i.e., the objects that are instantaneously accessible), and records status information.

Table 2-1 IP/GDP System Object Comparison

Object	IP Implementation
Processor Object	similar
Process Object	similar
Context Object	similar
Operand Stack	none
Instruction Segment	none
Object Table	identical
Domain	identical
Port	identical
Carrier	identical
Storage Resource	none
Type Definition	identical
Communication Segment	identical
Descriptor Controller	identical
Refinement Controller	identical

Legend:

identical	IP and GDP implementations are identical
similar	While conceptually similar, IP implements object differently than GDP
none	IP does not implement object

IP OPERATORS

Table 2-2 compares the operators available in the IP's function set to those provided in the GDP's instruction set. Since windows are unique to Interface Processors, the ALTER MAP AND SELECT DATA SEGMENT function has no counterpart in the GDP. Conversely, the IP has no functions for performing arithmetic (except for the exclusion function INDIVISIBLE ADD SHORT ORDINAL) or logical operations on numeric or character data types, nor any operators to alter the flow of execution (e.g., branch or call functions). To the extent that these classes of operators are needed in a Peripheral Subsystem interface, they can be provided by the combination of the Attached Processor's instruction set and the IP's window facility. For example, by opening a window on a message received from a GDP process, the I/O controller can use AP instructions to test and branch on the value of a message field read through the window.

Through its windows, an IP provides the basic ability to read and write the contents of objects composed of data segments. However, using its function request facility an IP can manipulate an access descriptor which references an object. The IP can examine a complex (multi-segment) object, gaining access to its component segments. It can perform type and rights manipulation on both hardware-recognized typed objects as well as software-recognized types. When manipulating software-recognized types, an I/O controller is acting as a type manager and its actions must be coordinated with the 432 type manager which has created the object.

The Interface Processor provides the I/O controller with both process and processor communication facilities. Interprocess communication is asynchronous and is performed with the aid of ports, system objects which provide synchronization and queuing for messages. Any object may be sent as a message from a process to a port. Interprocessor communication messages are predefined. Some of them apply to all classes of 432 processors, and others are specific to a particular class (e.g., IP or GDP) of processor. The I/O controller can send one of these messages to an individual processor, or it can broadcast a message to all processors in the 432 system.

Table 2-2 IP/GDP Operator Comparison (Part 1 of 2)

Operator	Implementation
WINDOW DEFINITION OPERATOR	
Alter Map and Select Data Segment	IP
ACCESS DESCRIPTOR MOVEMENT OPERATORS	
Copy Access Descriptor	GDP+IP
Null Access Descriptor	GDP+IP
RIGHTS MANIPULATION OPERATORS	
Amplify Rights	GDP+IP
Restrict Rights	GDP+IP
TYPE DEFINITION MANIPULATION OPERATORS	
Create Public Type	GDP
Create Private Type	GDP
Retrieve Public Type Representation	GDP+IP
Retrieve Type Representation	GDP+IP
Retrieve Type Definition	GDP+IP
REFINEMENT OPERATORS	
Create Generic Refinement	GDP
Create Typed Refinement	GDP
Retrieve Refined Object	GDP+IP
SEGMENT CREATION OPERATORS	
Create Data Segment	GDP
Create Access Segment	GDP
Create Typed Segment	GDP
Create Access Descriptor	GDP
ACCESS PATH INSPECTION	
Inspect Access Descriptor	GDP+IP
Inspect Access	GDP+IP
OBJECT INTERLOCK OPERATORS	
Lock Object	GDP+IP
Unlock Object	GDP+IP
Indivisibly Add Short Ordinal	GDP+IP
Indivisibly Add Ordinal	GDP
Indivisible Insert Short Ordinal	similar
Indivisible Insert Ordinal	GDP
CONTEXT COMMUNICATION OPERATORS	
Enter Access Segment	GDP+IP
Enter Process Globals Access Segment	GDP+IP
Set Context Mode	similar
Call Context	GDP
Call Context with Message	GDP
Return	GDP

Table 2-2 continued IP/GDP Operator Comparison (Part 2 of 2)

PROCESS COMMUNICATION OPERATORS	
Send	GDP+IP
Receive	GDP+IP
Conditional Send	GDP+IP
Conditional Receive	GDP+IP
Surrogate Send	GDP+IP
Surrogate Receive	GDP+IP
Delay	GDP
Read Process Clock	GDP
PROCESSOR COMMUNICATION OPERATORS	
Send to Processor	GDP+IP
Broadcast to Processors	GDP+IP
Read Processor Status and Clock	GDP+IP
Move to Interconnect	GDP*
Move from Interconnect	GDP*
BRANCH OPERATORS	GDP
CHARACTER OPERATORS	GDP
SHORT ORDINAL OPERATORS	GDP
SHORT INTEGER OPERATORS	GDP
ORDINAL OPERATORS	GDP
INTEGER OPERATORS	GDP
REAL OPERATORS	GDP
TEMPORARY REAL OPERATORS	GDP

Legend:

GDP+IP	IP and GDP Implementations are identical
IP	IP implements operator, GDP does not
GDP	GDP implements operator, IP does not
similar	While conceptually similar, IP implements operator differently than GDP
*	Window 1 of IP provides equivalent interconnect access

2-2. OBJECT ADDRESSING AND GLOBAL STORAGE MANAGEMENT

Object addressing on the IP follows the same three level sequence as on a GDP. The steps taken to address an object are:

1. Given an access descriptor, a processor uses the directory index field to index the object table directory and gain a storage descriptor for the object table which contains an object reference for the desired object.
2. With the storage descriptor for the object table and the segment index field of the access descriptor, the processor locates a storage descriptor for the requested object.
3. The storage descriptor for the object contains the base and length information required to locate the object in 432 memory.

An IP can be directed to manipulate objects in 432 memory, just as other 432 processors, but lacks any facility to create objects. All original objects used by an IP must be predefined and loaded into 432 memory at system initialization time. Additional objects, which may be required, must be created by a GDP process (e.g. the storage manager).

A 432 operating system type manager might maintain a template for a prototype IP process. When it received a request for a new IP process from the I/O controller the GDP would build one using the prototype and then return it via the standard communication port mechanism.

2-3. OBJECTS FOR PROGRAM ENVIRONMENTS

The IP supports the same program environment hierarchy (process, context, domain) as a GDP but implements each level differently.

The IP does not require that a domain object be implemented but the context object contains a slot for an access descriptor for a domain object should one be required. When implemented, IP domains do not contain instruction segments (since the IP does not fetch instructions) or operand stack segments. The domain may be used to store some static information which may be required by a process.

An IP context is a refinement of an IP process object. Each IP process is bound to a single context for the lifetime of the process. An environment is changed by invoking the ENTER ACCESS SEGMENT or ENTER GLOBAL ACCESS SEGMENT functions.

2-4. FACILITIES FOR ASYNCHRONOUS COMMUNICATION

The IP offers the same set of operators for asynchronous interprocess communication as does a GDP, with the exception that the DELAY operator is not implemented. The DELAY operator, used in scheduling to delay a process from being dispatched (on a GDP), is not required by an IP where process scheduling and dispatching is performed by the I/O controller.

2-5. PROCESSES AND LOCAL STORAGE RESOURCE MANAGEMENT

The IP performs no process scheduling or local storage resource management. Multiple IP process objects may coexist in 432 memory. I/O controller software must select a process environment in which an IP function is to be performed.

Unlike the GDP, where a process may be composed of multiple contexts, an IP process is bound to a single context during its lifetime. In fact, the context is a refinement of an IP process object. Further, since no local storage management is performed by an IP, the size of a process's context is static over the life of the process.

2-6. PROCESS SCHEDULING AND DISPATCHING

Generally, software in the I/O controller is responsible for all IP process scheduling and dispatching. A process is selected and bound to an IP processor object when an IP function is invoked. The process selection index field in the IP's function request facility specifies which process is to be selected. Since the IP is not self-dispatching, a strategy routine in the I/O controller has responsibility for multiplexing the various IP processes over time. The IP does not maintain a process clock. Process time management is performed by the I/O controller.

Consistent with 432 philosophy, the IP provides the mechanisms for process scheduling and dispatching but the policy for deployment is totally under the direction of I/O controller software.

2-7. FACILITIES FOR OBJECT MANAGEMENT

The IP provides a spectrum of facilities which may be used for securely managing objects: communications ports, I/O locks, and indivisible short ordinal operations.

The IP offers the same asynchronous communication port mechanisms as a GDP. Communications ports may be used by processes to asynchronously send and receive messages (objects).

Contained in each object's storage descriptor is an I/O lock which is applied by the IP when a window is opened on the object. This lock serves two purposes: first it guarantees that only one IP window can be opened on a particular 432 object at a time; second it prevents movement of the object (e.g. by a memory compaction process) while it is mapped through a window.

The transfer of data between the PS and the 432 system is a three step process. First, the IP controller opens a window onto the 432 object which is to used in the transfer. In the process of opening the window the IP sets the I/O lock in the affected object. Second, the data transfer phase is entered and a PS processor transfers data between the 432 object and the PS memory. Finally, when the transfer is completed, the IP controller closes the window and the IP clears the I/O lock in the 432 object. The storage manager in the 432 system may query the I/O lock field but this field is not hardware-interpreted by a GDP.

As primitives in the IP hardware function set, two indivisible operators are provided which can be used to guarantee mutually exclusive access to short ordinal fields within 432 objects. These two operators, INDIVISIBLE ADD SHORT ORDINAL and INDIVISIBLE INSERT SHORT ORDINAL, apply an indivisible hardware operation to the specified short ordinal value. For instance, these operators might be employed to provide a counting semaphore. These operators provide only the hardware-specific mutual exclusion mechanisms and must be supplemented by a coordinated software discipline between processes which utilize the semaphore. For a discussion of the read-modify-write memory requirements for these operators, see the Intel iAPX 43203 Interface Processor Data Sheet, Order Number 171874.

2-8. CONTEXT ENVIRONMENT MANIPULATION

The I/O controller, by manipulating the context of an IP process, can access the objects which are available to the process. Like a GDP, the IP allows a context to reference any object for which it holds an access descriptor. Entry access segments contain access descriptors for all the objects which may be manipulated from a specific process's context by an I/O controller.

The Four Entry Access Segments

Of all the access segments which can be referenced from a context, the IP provides direct access to a set of four entry access segments. The entry access segments are referenced by access descriptor slots 4, 5, 6, and 7 in the context access segment. Entry access segment 0, slot 4, contains the access descriptor for the context access segment; entry access segment 0.

Direct vs. Indirect Accessibility

If a copy of an access descriptor for an object is in one of the four entry access segments, the object it references is directly accessible. To reference such an object, two values must be specified:

- The number (0 to 3) of the entry access segment in which the access descriptor is located, and
- The index (0 to 16383) of the access descriptor within the specified entry access segment

When viewed from the standpoint of the 432 system and the Peripheral Subsystem, there are actually several perspectives on accessibility as shown in Table 2-3. A processor (GDP or IP) in the 432 system can directly reference any object for which it holds an access descriptor in one of its entry access lists. In addition, by traversing access paths, the 432 processor can manipulate objects which are indirectly accessible.

If a copy of the access descriptor is not currently in one of the four entry access segments, the desired object may be indirectly accessible. The target object may be part of a complex object structure which must be traversed by following the appropriate access path. Once the particular access descriptor for the object has been located, the object may be made directly accessible by entering the access segment into one of the reuseable entry access lists (1-3). Entry access segment 0 is always reserved for the original context access segment. An access segment of process globals may be entered into one of the other three access lists by the "enter global access segment" function. Together, these two access segments provide access to all the objects which a context can reference.

An AP has a different view of accessibility. The AP can only access 432 data through IP windows which are opened onto 432 data segments. When a window is open, the AP can use its native data manipulation operators to modify the information through the window. When the AP must reference data in a segment which is indirectly accessible, it issues a function request to the IP to traverse an access path to the segment. When the data segment has been made directly accessible for the AP, the IP interrupts the AP.

Table 2-3 Direct/Indirect Accessibility

Viewpoint of IP/GDP in 432 System

Directly Accessible 432 Information

o access descriptors — All access descriptors in the four Entry Access Segments.

o data — All objects of type data segment referenced by access descriptors in the four Entry Access Segments.

Indirectly Accessible 432 Information

o Information, data or access, which can be reached via access path manipulation (i.e. by following a chain of access descriptors using the Enter Access Segment function).

Viewpoint of AP in Peripheral Subsystem

(Controlling an IP operating in logical reference mode)

Directly Accessible 432 Information

o access descriptors — NONE, the AP cannot directly alter access information.

o data — all objects of type data segment for which a window is currently opened. Note, this implies the object is directly accessible to the IP.

Indirectly Accessible 432 Information

o Objects of type data segment which are directly accessible to the IP but which have not been mapped through a window. These objects can be made directly accessible by issuing an IP function request which opens a window to the object.

o Access descriptors in the Entry Access Segments. These can never be made directly accessible to the AP but can be manipulated via the IP function request facility.

o Information, data or access, which can be reached via access path manipulation (i.e. by following a chain of access descriptors using the Enter Access Segment function provided by the IP function request facility). Note that two levels of indirection are involved, traversing the path of access descriptors and the use of the IP function request facility.

Object Selectors

An object selector identifies an object by specifying an access descriptor contained in one of the four entry access segments. The object selector consists of a double byte quantity composed of two fields:

1. The low order two bits of the object selector specify which entry access segment holds the desired access descriptor and are coded as follows:
   00 - Entry Access Segment 0 (Context Access Segment)
   01 - Entry Access Segment 1
   10 - Entry Access Segment 2
   11 - Entry Access Segment 3

2. The high order 14 bits represent a scaled index into the specified entry access segment.

An object selector allows access to any of the 16,384 (2^{14}) access descriptors from each of the 4 entry access segments. An IP can potentially reference 65,536 (2^{16}) objects directly.

Entering an Access Segment

The instruction ENTER ACCESS SEGMENT allows the I/O controller software to enter a given access segment into Entry Access Segment 1, 2, or 3. ENTER ACCESS SEGMENT requires two operands:

- An access descriptor for the access segment to be entered into EAS1, EAS2, or EAS3, and
- An unsigned integer value designating the destination entry access segment, which must be 1, 2, or 3.

Entering the Global Access Segment

Each IP process maintains a global access segment which is always accessible to the I/O controller via the ENTER GLOBAL ACCESS SEGMENT function. Immediate entry of the global access segment allows an I/O controller to gain access to the set of process globals. The I/O controller needs only to specify which of the three available entry access segments is to be used when requesting this function.

CHAPTER 3
WINDOWS

The Interface Processor window mechanism provides the Peripheral Subsystem with protected access to the contents of objects located in the 432 system. There are five windows, labeled 0-4. Each window can be used to access one (single segment) object. To prevent the possible manipulation of access descriptors as ordinary data and corruption of the protection mechanisms, the windowed object must be of base type data segment. Access descriptors, the basis for the 432 protection system, may be manipulated only by IP operations supplied by the IP function request facility. These operations are described in the next chapter.

All IP windows are similar in that they support the transfer of data across the subsystem boundary; this chapter first describes the characteristics common to all windows. The first section covers the attributes that define windows; these are generally specified when the window is opened with the ALTER MAP AND SELECT DATA SEGMENT function. The second section describes the operation of a data transfer through a window that has been defined with a given set of attributes.

Three of the windows have special capabilities; these are covered after the basic properties of all windows have been described. Window 0 may be used to perform high speed block transfers. Window 1 may be opened onto the processor-memory interconnect address space and thus provide access to interconnect objects. Window 4--the control window--is dedicated to providing the data path for the Interface Processor function facility; this is covered in chapter 4.

Throughout this chapter conditions for correct use of windows are described. When any of these conditions are violated, the Interface Processor detects a fault. The IP's fault detection, reporting and handling facilities are covered in chapter 6.

3-1. WINDOW ATTRIBUTES

Each window has a set of attributes which define its state at a given moment; these are summarized in table 3-1. The IP sets the attributes of all five windows when it performs processor qualification. The attributes of the control window are obtained from values recorded in the processor object. Processor qualification closes windows 0-3.

Processor qualification is performed explicitly when the Interface Processor responds to a "suspend and fully requalify processor" interprocessor communication (IPC). The IP performs processor qualification implicitly in response to the startup IPC it receives during system initialization (see appendix E). Thus, window 4 may be made to come up with any set of attributes by encoding the desired values in the processor object image that is loaded during initialization.

Having entered logical reference mode, the I/O controller can change the attributes of windows 0-3 with the ALTER MAP AND SELECT DATA SEGMENT function. Unlike the other windows, window 4's attributes may not be altered during normal execution; its attributes are fixed once logical mode is entered. The IP can be commanded to reenter physical mode by a special IPC from a 432 processor, including itself. Any processor with an access descriptor for a processor object with broadcast rights can send the "enter physical mode" IPC to all processors in the 432 system. GDPs ignore this interprocessor message.

WINDOW STATUS

A window must be open for it to be used to transfer data. An open window establishes an active mapping between a set of addresses in the Peripheral Subsystem and an object in the 432 system; other attributes provide further mapping information.

A closed window is inactive, and has no other attributes. A window may be closed to prevent further access to an object, or to change the attributes of a window. Closing a window which overlays PS memory (see OVERLAY in this section) enables access to the PS memory.

When a window detects a fault, the IP records in 432 memory the fault information describing the circumstance, changes the state of the affected window to the faulted state, and interrupts the AP. In the faulted state the IP will continue to acknowledge transfers through the window though no data will actually be moved to/from the 432 system (see the description of XACK/ and NAK/ in the Intel iAPX 43203 Interface Processor Data Sheet, Order Number 171874). This state is entered to allow DMA-type controllers to proceed safely in the presence of a window fault.

Table 3-1 Window Attribute Summary

Attribute	Description
Window Status	Window is open/closed/faulted
Subrange Base Address	Start of windowed subrange in the PS
Subrange Size	Length of windowed subrange in the PS
Object Reference	Object Selector for windowed 432 object
Base Displacement	Displacement in bytes into windowed 432 object
Direction	Read/write permission for windowed object. When the window is being opened this attribute is the permission requested by the I/O controller. After the window has been opened this attribute is the permission that has been granted.
Transfer Status	Transfer in progress/terminated/faulted
Mode	Window 0: random/block mode Window 1: memory/interconnect mode Window 2-4: always in random mode
Overlay	Windowed subrange does/does not overlay memory
Block Mode Attribute	Description (applies only to window 0)
Byte Count	Count of the number of bytes to be transferred minus one.

Note: In block transfer mode, the base displacement of window 0 specifies the initial address within the windowed object from which consecutive information transfer will begin.

SUBRANGE BASE ADDRESS AND SUBRANGE SIZE

A window's subrange is defined by a subrange base address and a subrange size, in bytes. The subrange is the contiguous set of Peripheral Subsystem memory addresses that are mapped by the window. A Peripheral Subsystem bus master that references an address in a subrange accesses the corresponding object in the 432 system.

A PS subrange is defined in terms of powers of 2. The subrange size of a random mode window may be specified as any power of 2 from 2^0 through 2^{15} (i.e., 1 through 32k bytes). When window 0 is used in block mode it may sequentially access an object as large as 64K bytes. When the target object is not an integral power of 2 in length, the subrange will normally be specified as the next higher power of 2. The subrange may also be smaller than the target object, if access to the full extent of the object is not required.

Note that the size of the window is the lesser of the size of the subrange and the size of the object. That is, a window never provides access to 432 system memory beyond the extent of the windowed object, regardless of the relationship of subrange size to object size. The IP's protection system restricts a larger subrange to behaving as though it is exactly the same size as the windowed object. Any attempt to access locations beyond the extent of an object will cause the IP to generate a fault.

A subrange's base address is specified as an offset in bytes from the beginning of the IP's 64K byte range in the PS. The subrange base address bears a definite relationship to the subrange's size. Given a subrange 2^n bytes long, its base address must be on a 2^n byte boundary. For example, the base address of a 4K subrange must be evenly divisible by 4K. This relationship may also be expressed as: the base address of a 2^n byte subrange, expressed in binary, must contain at least n low-order zero bits.

The following constraints apply to all active subranges:

- o no subranges may overlap, i.e. no two subranges may include the same Peripheral.Subsystem address
- o all subranges must "fit" within the range of addresses (up to 64K) that the IP occupies in the Peripheral Subsystem memory space.

OBJECT REFERENCE

An open window's object reference begins as an object selector and is converted by the IP into an access descriptor for the windowed 432 object. Each open IP window must map a different object in 432 memory, and each object must be represented as a single segment of base type data segment (functions may be used to manipulate multi-segment objects to gain access to their individual segments). No more than one window can be opened on an object, regardless of whether there are multiple IP's in the system. Even if one IP window is opened on a refinement of an object no other window will be allowed access to the base object or any refinement of the object.

When a window is opened on an object, the IP makes the object inaccessible to other IPs by setting the I/O lock bit in the base object's object descriptor; the I/O lock bit in the base object is set when a window is opened on a refinement. The object may, however, remain accessible to GDP processes holding object references for it. If the Peripheral Subsystem requires exclusive access to an object, it must do so by means of a convention. For example, if the object has been defined with a lock field, the IP controller can use the LOCK OBJECT function to prevent GDP processes (which observe the convention) from accessing the object. An alternate convention, might be used for objects which do not contain lock fields. For example, a GDP process sending an object to the I/O controller could agree not to access the object, or pass a reference for it to another process, until the I/O controller sends the object as a message back to the GDP process.

The IP supports the 432 philosophy that software should have access to the minimum set of objects needed to perform its function. Therefore, the I/O controller can only open a window on an object for which an access descriptor exists within a current context's access environment. Typically, an I/O service request message from a 432 processor will contain access descriptors for the objects that need to be transferred or accessed.

DIRECTION

The direction attribute specifies whether the windowed object may be read, written, both read and written, or neither read nor written. When the window is opened the IP checks the requested direction attribute with the access rights granted by the object reference. The access rights requested in the direction attribute must be equal to, or logically less than, the rights granted by the object reference. For example, if the object reference indicates that the object may be read, then the permissable direction attributes are read, or neither read nor write; requesting the ability to write, or to read and write the object would be illegal.

Once a window has been successfully opened, the IP checks every subsequent subrange address reference to insure that it conforms to the direction attribute, otherwise an active window fault occurs. (The IP's read/write line identifies the type of access being attempted.) This permits the IP controller to open a window for reading with the assurance that a mis-programmed DMA controller will not be able to write into it.

TRANSFER STATUS

An open window may take one of four states:

- o transfer in progress;
- o transfer terminated by fault;
- o transfer terminated by count runout; (block mode only)
- o transfer termination forced; (block mode only).

The IP controller will open a window with the status attribute set for "in progress". If the IP detects a fault associated with an active window, it will change the status attribute to "terminated by fault". A random mode window which is closed (set invalid) with a transfer status of "in progress" is considered to have terminated normally since there is no means for an IP to predict when a random mode transfer is finished. The remaining two states are associated with window 0 block mode transfers only and are described in section 3-3.

TRANSFER MODE

Windows 0 and 1 have alternate transfer modes that may be selected by setting the mode attribute when the window is opened. Window 0 may be opened in block mode, which permits buffered high speed transfers of contiguous blocks of data; this is described in section 3-3. Window 1 may be opened onto the interconnect address space; this is described in section 3-4. The transfer mode attribute has no meaning for windows 2-4, which support random transfers to 432 system memory only; the random transfer mode is described in section 3-2. Attempting to set the transfer mode of windows 2-4 will cause a fault.

OVERLAY

Some Peripheral Subsystems (e.g., those based on processors with limited address spaces) may not be able to dedicate a block of memory space for exclusive use as IP window subranges. Such systems may elect to co-locate all or part of the IP's range with real PS memory. If a window is then opened with the overlay attribute, the IP will inhibit the co-located memory from responding to memory references in the subrange. Closing a window that overlaid memory re-enables the memory to respond to subsequent address references in that subrange. Thus, when the IP and PS memory both occupy the same addresses, memory will respond to all references except those that fall in the subrange of a window open with the overlay attribute.

Figure 3-1 illustrates a hypothetical configuration in which a bank of memory and an Interface Processor both occupy a 64K byte block of addresses in the Peripheral Subsystem memory space. A window with a subrange base address of 32K and a subrange size of 4K has been opened with the overlay attribute set. Any address reference falling in the subrange will cause the IP to respond rather than the co-located memory. Any address reference outside the subrange will select the memory rather than the IP.

The overlay facility is implemented by an inhibit signal that the IP asserts when it recognizes an address reference that falls in an overlaid subrange. (See the <u>iAPX 43203 Interface Processor Data Sheet</u>, Order No. 171874, for a description of this signal). Use of the overlay facility slows IP response time somewhat.

Note that opening a window with the overlay attribute set when there is no co-located memory is safe, but it slows IP response unnecessarily. On the other hand, opening a window without specifying overlay when there is co-located memory will produce an undefined result when both components attempt to respond to a subsequent address reference that falls in the overlaid subrange.

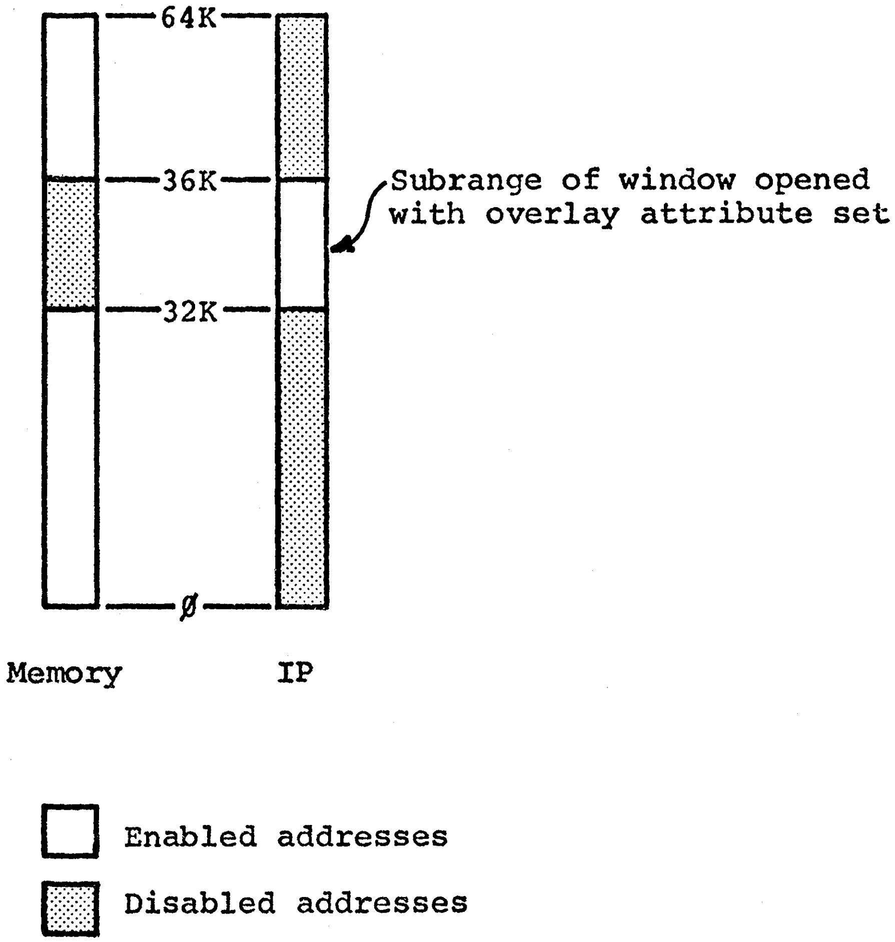


Figure 3-1   Memorv Overlay

3-2. WINDOW OPERATION

This section describes the IP's response to an address reference that falls into the windowed subrange of an open window. The discussion covers random mode transfers to and from ordinary memory-based objects; the special cases of block mode, interconnect objects and function requests are covered in subsequent sections.

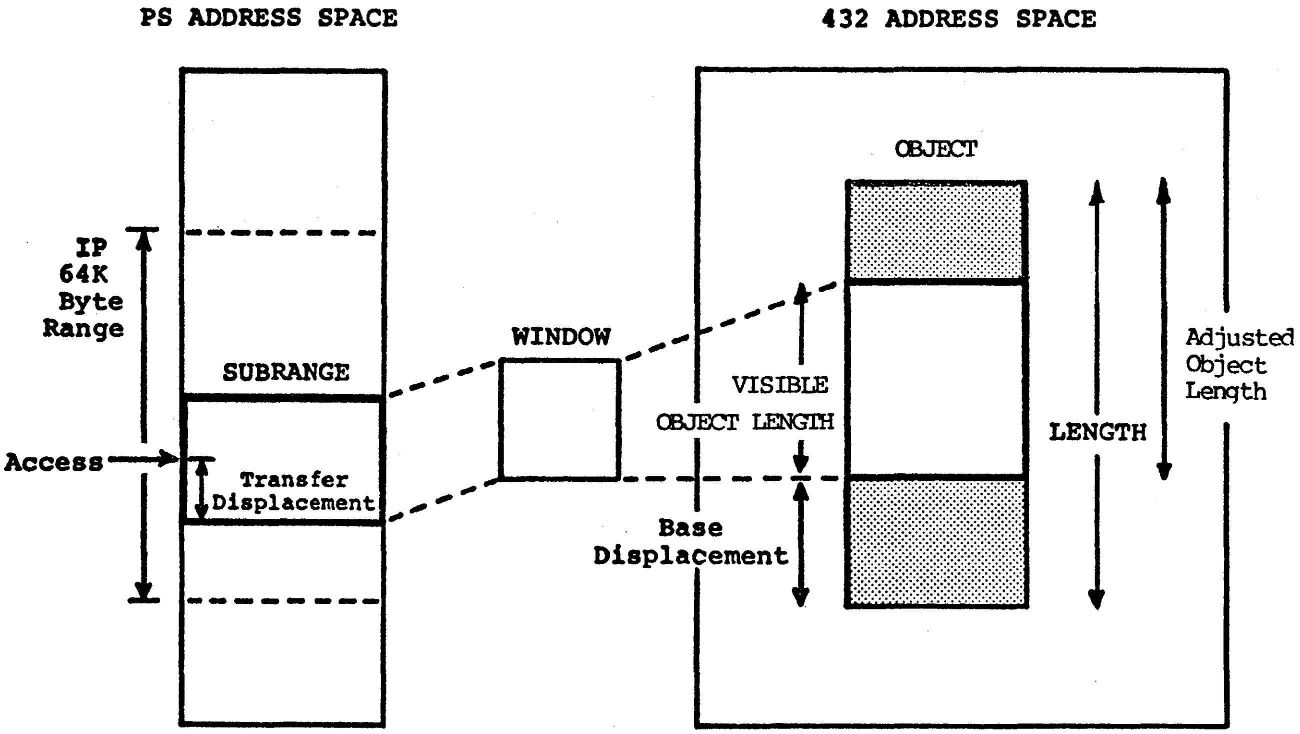
ADDRESS RECOGNITION

The Interface Processor monitors all Peripheral Subsystem address references that fall into its range. It compares each address presented on the Peripheral Subsystem bus to the subranges of all open windows. If an address falls into a subrange, the IP recognizes the reference and responds as described below. If the address does not fall into an active subrange, the IP ignores the reference and does not respond.

CONSISTENCY CHECK

Given that it has recognized an address reference, the IP checks it for consistency before performing the actual transfer. There is a series of these checks which are equivalent to the steps carried out by a GDP when an instruction attempts to access data in an object. Although they are described here as a sequence, the hardware is able to perform some of the checks in parallel.

The IP insures that the transfer direction (as indicated by its read/write line) is consistent with the window's direction attribute. The IP computes the PS transfer displacement, that is, the position of the item (byte or double-byte) relative to the base address of the PS subrange. The visible object length is the difference between the length of the object and its base displacement (see Figure 3-2). The transfer displacement must be less than or equal to the visible object length. The sum of the physical base address and the transfer displacement must be less than the largest physical 432 memory address (2^{24}-1). (A memory bounds error would indicate erroneous information in the object table.) If any of these checks fails, the IP detects a fault and does not perform the transfer. Figure 3-2 illustrates the constraints which the IP applies when the consistency check is performed. Several examples of valid mappings of window onto objects are shown in Figure 3-3.



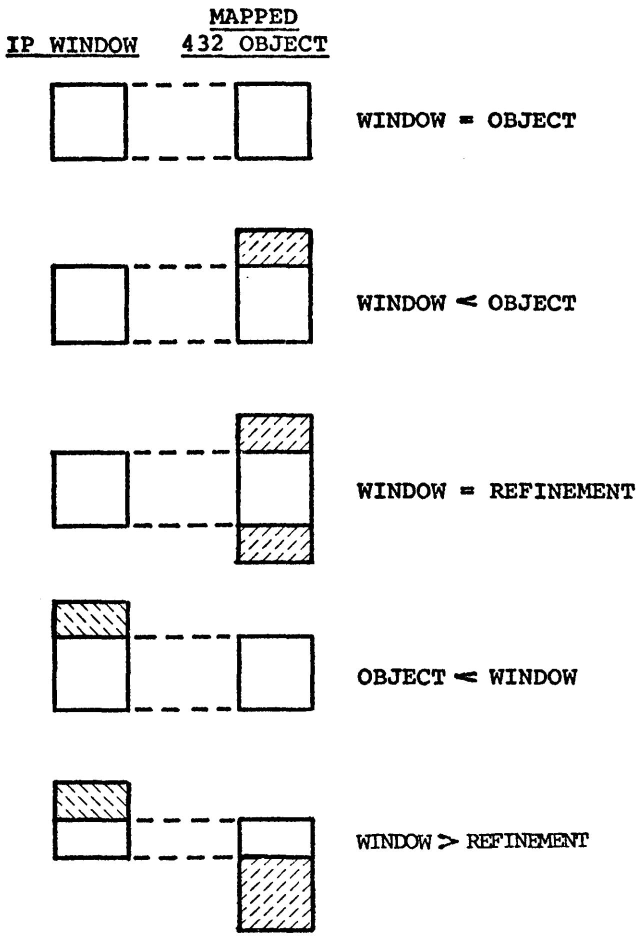
Initial Computations

- Adjusted Object Length = Object Length - Base Displacement
- Visible Object Length = Minimum (Adjusted Object Length, Byte Count) for block mode operation.
- Visible Object Length = Minimum (Adjusted Object Length, Subrange Size) for random mode operation.
- Physical Base Address = Base Address + Base Displacement
- During block transfers in logical mode (window 0 only), the byte count must be less than the Visible Object Length.

Constraints During Data Transfer

- Transfer Displacement must be less than the Visible Object Length
- Physical Base Address + Transfer Displacement must be less than $2^{24}-1$

Figure 3-2 Subrange/Window Attributes (Logical Mode)



▨ — PORTION OF OBJECT INACCESSIBLE TO IP

▧ — PORTION OF WINDOW INACCESSIBLE TO AP

Figure 3-3 Valid Window/Object Mapping

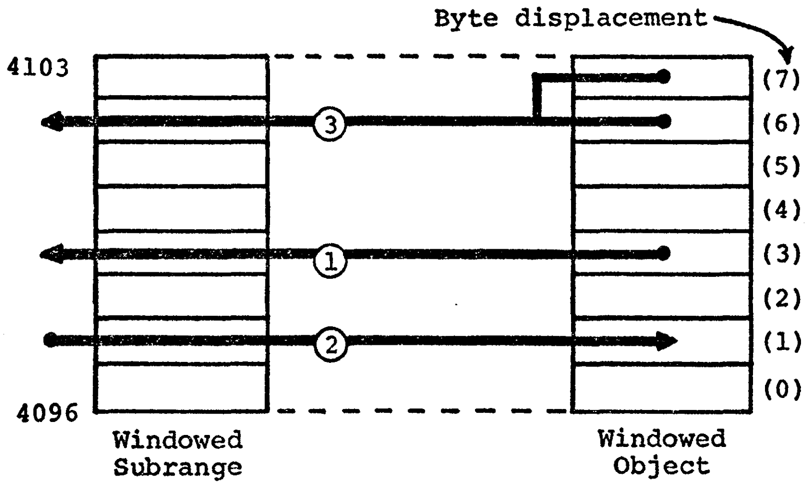
3-3. RANDOM MODE DATA TRANSFER

Given that an IP address reference has passed the consistency checks, the IP finishes the Peripheral Subsystem bus cycle just as a memory component would, accepting data from the bus in a write operation, and placing data on the bus in a read operation.

It follows from the preceding discussion of transfer displacement computation that random mode transfers are always between corresponding relative locations of the windowed subrange and the windowed object. That is, the displacement of a transferred byte or double-byte is identical within the windowed object and the windowed subrange. For example, assume a PS subrange of 128 bytes at base address 4096 mapped onto a 432 object 100 bytes long with a base displacement of 0. If a Peripheral Subsystem bus master initiates a bus cycle that decodes as "read one byte from location 4096", the IP will return the object byte whose displacement is zero, the first byte in the object. If a subsequent bus cycle indicates "write a double-byte into location 4100", then the IP will accept a double-byte from the bus and write it into the object at a displacement of four. If another bus cycle attempts to "read one byte from location 4197", the IP will fault and will not perform the transfer, since the subrange transfer displacement exceeds the bounds of the object.

Random mode is so-called because no ordering is implied between successive references to a windowed subrange. Any location may be read or written (assuming validitv checks are passed) at any time. Figure 3-4, Random Mode Transfers, illustrates the effect of different address references when a window is opened for reading and writing in random mode.

A window opened in random mode may be remapped onto a new 432 data segment with a single invocation of the IP function ALTER MAP AND SELECT DATA SEGMENT. When executing this function the IP will first close the window and then reopen it on the newly select data segment.



Legend

Reference Sequence:	①	②	③
Subrange Address Referenced:	4Ø99	4Ø97	4102
Reference Operation:	Read Byte	Write Byte	Read Double-byte
Object Byte Accessed (disp.)	3	1	6,7

Figure 3-4   Random Mode Transfers

3-4. BLOCK MODE DATA TRANSFER

Window 0 can be opened in random mode or in block mode. Block mode allows the Peripheral Subsystem to take advantage of software instructions (e.g. iAPX 86 string operations) and devices such as DMA controllers, which are capable of generating consecutive address references at high speed. Block mode also permits the transfer of a large amount of data through a small PS address subrange. For example, the full content of any object may be transferred through a one-byte or double-byte PS subrange. This helps to keep more of the IP's range available for use with random mode windows.

While block mode is well-suited for the high speed transfer of large blocks of data, it provides less addressing flexibility than random mode. When window 0 is opened in block mode, the direction attribute can specify reading or writing, but not both. To change access directions requires closing and re-opening the window. Block mode also implies serial addressing of the windowed object. The block of data to be read or written is defined when the window is opened, and the whole block is transferred in sequence.

BLOCK MODE ATTRIBUTES

Window 0 has an additional attribute, byte count, which is applicable only when it is opened in block mode. The byte count specifies the size of the block that is to be moved through the window. The value of this attribute may range from 0-65,535; the value represents one less than the number of bytes to be transferred (a byte count of 0 indicates that a one-byte block is to be transferred). The byte count is independent of the subrange size. However, the IP checks to insure that the sum of the base displacement plus the byte count does not exceed the length of the target object.

The base displacement attribute locates the first byte of the block relative to the beginning of the windowed object. A value of zero indicates that the block starts at the lowest address of the object. The base displacement and byte count essentially define a refinement of the object.

BLOCK MODE CONSISTENCY CHECK

Since the byte count and base displacement effectively predefine the transfer from the perspective of the 432 object, the IP can perform most of the required consistency checks when the window is opened. The only checks made during a transfer are direction and byte count.

BLOCK MODE OPERATION

From the point of view of the Peripheral Subsystem bus, a block transfer proceeds much like a random transfer, except that, like a fast memory, the IP provides much better response time in block mode. The IP acts as a passive agent on the PS bus, all block transfer activity being driven by an active PS processor or DMA controller. When an address reference falls within window 0's subrange, the IP accepts or supplies a byte or double-byte according to the type of PS bus cycle. Note, however, that in block mode, IP acknowledgement of a write operation does not neccessarily imply that the data has actually been written into the windowed object.

The IP employs an on-chip first-in-first-out (FIFO) buffer to achieve high speed block transfers in buffered mode. Since a block mode transfer is predefined by window 0's attributes, the IP is able to optimize the transfer using the FIFO hardware assistance. The Interface Processor buffers block mode transfers to improve response to Peripheral Subsystem transfer requests and to reduce its utilization of the 432 processor packet bus.

In a block read operation, the Interface Processor pre-fetches an eight-byte block of data from the windowed object in one 432 processor packet bus transaction. It holds the block in an internal buffer and supplies bytes or double-bytes from the buffer as requested by Peripheral Subsystem bus cycles. When the buffer has enough free space, the IP prefetches another block.

In a block mode write operation, the IP accepts bytes or double-bytes from the Peripheral Subsystem bus and buffers them internally. When the buffer accumulates more than eight bytes, the IP post-stores an eight-byte block in the windowed object in a single processor packet bus operation.

Completing a block mode write transfer which is shorter than the byte count is a two-step process. First, the AP must issue an ALTER MAP AND SELECT DATA SEGMENT function with the entry state operand to "force termination" on window 0. This causes the IP to empty its FIFO to 432 memory. Then, the AP must issue an additional ALTER MAP AND SELECT DATA SEGMENT FUNCTION with an entry state operand to set window 0 invalid (close the window). If the AP attempts to close a block mode window without first forcing termination, the IP will generate a fault, interrupt the AP, and preserve the block mode window. When the transfer length is the same as the byte count attribute, the IP automatically takes care of the last block which will be short if the transfer size is not a multiple of eight.

BLOCK MODE TERMINATION

A block mode transfer will terminate normally when all bytes have been transferred, or it may terminate prematurely should a fault occur. In both cases, the IP updates the transfer status attribute and issues an interrupt request to notify the Attached Processor. Following termination, any address reference falling in the subrange of window 0 will cause the window to fault and enter the error state. In the error state, requests for data transfer will be will be acknowledged (negatively) by the IP, but no data will be transferred. This prevents a DMA controller, for example, from continuing to transfer data after a fault has been detected. The faulted window cannot be re-used until it is closed and re-opened.

The IP tracks the progress of a block transfer by means of an on-chip byte counter. The IP sets this counter equal to the byte count attribute when the window is opened and decrements it with each byte transferred. When the on-chip counter underflows (is decremented from zero) all bytes have been transferred and the operation is terminated normally.

The IP will terminate a block transfer prematurely if it detects a fault during the transfer. In addition, the I/O controller may itself force termination before the transfer has been completed. This is done by executing an ALTER MAP AND SELECT DATA SEGMENT function with the transfer status attribute set to "termination forced." Finally, termination may be forced by the IP's receipt of of any the interprocessor communication messages "suspend and fully requalify processor", "close windows", or "close windows and enter physical mode".

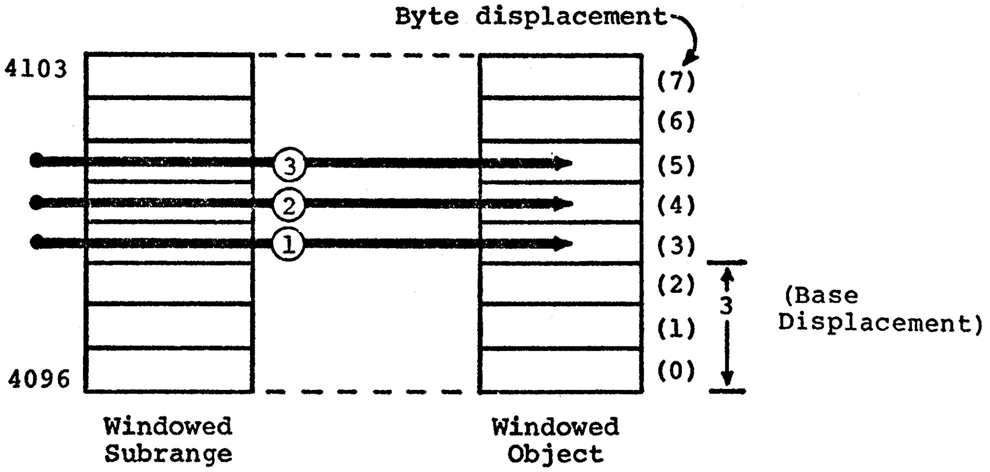
BLOCK MODE ADDRESSING

As mentioned earlier, in a block mode transfer the IP determines the displacement of a transfer into the windowed object by means of its on-chip displacement counter. Unlike random mode, then, the object displacement is independent of the subrange displacement. This gives rise to two addressing techniques that may be used by the Peripheral Subsystem in block mode: swept and source/sink.

In swept addressing, the Peripheral Subsystem bus master driving the transfer operation "sweeps" serially (from low addresses to high) through a block of addresses in the windowed subrange. That is, the address references will be n, n+1, n+2... or n, n+2, n+4... for 8- and 16-bit Peripheral Subsystem buses respectively. The range of PS addresses swept is equal to the number of bytes transferred, so the subrange must be at least as large as the number of bytes transferred. Figure 3-5 illustrates swept addressing in a block mode write operation.

In source/sink addressing, the master driving the transfer repeatedly addresses a single location in the windowed subrange. For a read operation, this single (byte or double-byte) location acts as a data source; for a write operation, the location serves as a data sink. By permitting the transfer of large blocks (up to 64K bytes) of data through a single location, source/sink addressing conserves "subrange space." To transfer 32K bytes in random mode requires setting up a 32K byte subrange, leaving only half of the IP's range available for concurrent use with other windows. Only a byte or double-byte of the range is needed to perform the same transfer in block mode using source/sink addressing. Figure 3-6 shows how source addressing works in a block mode read operation.

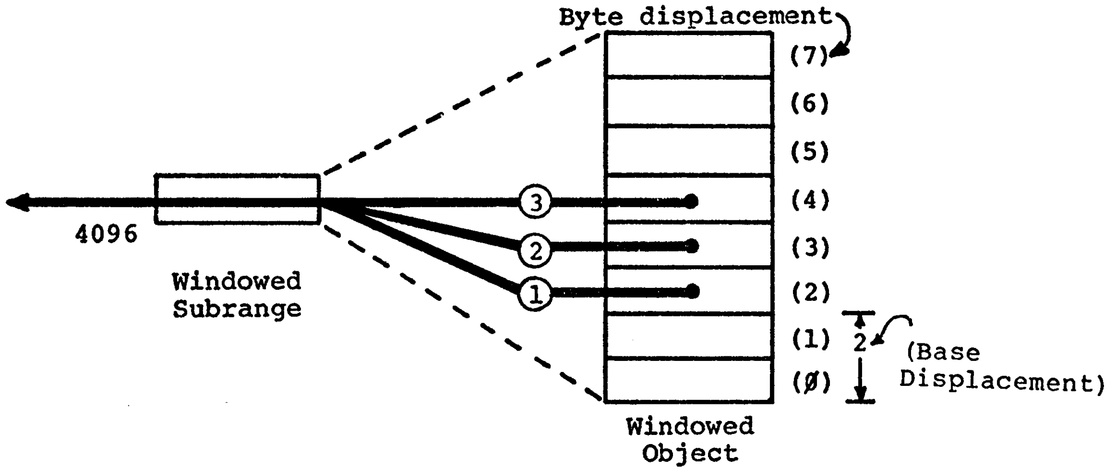
Note that the IP has no knowledge of the addressing technique used in a block mode transfer. It simply considers any address reference in window 0's subrange as a signal to transfer the next byte or double-byte.



Legend

Reference Sequence:	①	②	③
Subrange Address Referenced:	4Ø99	4100	4101
Reference Operation:	Write Byte	Write Byte	Write Byte
Object Byte Accessed (disp.):	3	4	5

Figure 3-5 Block Mode Writes - Swept Addressing



<u>Legend</u>

Reference Sequence:	①	②	③
Subrange Address Referenced:	4Ø96	4Ø96	4Ø96
Reference Operation:	Read Byte	Read Byte	Read Byte
Object Byte Accessed (disp.):	2	3	4

Figure 3-6 Block Mode Writes - Source Addressing

3-5. INTERCONNECT TRANSFERS

Window 1 may be opened onto either the 432 memory space or the 432 processor-memory interconnect space. The address space is selected by the transfer mode attribute when window 1 is opened; it may be changed at any time by closing the window and re-opening it with the transfer mode set differently. Both address spaces appear identical to the Peripheral Subsystem; interconnect objects may be read and written in exactly the same fashion as memory objects.

CHAPTER 4
FUNCTIONS

This chapter describes the common facility that supports the execution of all Interface Processor functions. The first section shows how window 4 is used to provide access to the facility. The next section explains how a function is requested by writing operands and an opcode through the window. The last two sections describe how the IP executes a requested function and returns status information upon completion of the operation.

4-1. FUNCTION FACILITY INTERFACE

Management of the IP function facility centers on the function request area of the processor data segment (see figure 4-1). Both the I/O controller software and the Interface Processor itself update and use the information recorded in this area via the control window. Briefly, the IP records the status of the function request facility in the function state field; the I/O controller may obtain status information by reading this field. The IP controller requests execution of a function by writing operands and an identifiying opcode into the function request area, and the IP reads these fields to obtain the information it needs to execute the function. Finally, the execution of some functions produces a value which the IP records in the return-value field, where the IP controller can inspect it. Upon completion of any function, the IP updates the status information and interrupts its Attached Processor. If desired, successful function completion interrupts can be disabled, thereby allowing only interrupts for unsuccessful completion to reach the AP.

In logical mode, the control window (window 4) is permanently opened onto the processor data segment and its mapping cannot be changed by an ALTERMAP function request. By reading and writing the corresponding PS memory subrange locations, the IP controller obtains access to fields in the function request area located in 432 memory. Notice that this interface mechanism is similar to a conventional memory-mapped peripheral device controller; the function request area fields are read and written like command, data and status registers.

Figure 4-2 illustrates the effect of executing a function, ALTER MAP AND SELECT DATA SEGMENT, which in this case alters the map of window 0 and selects a different 432 data segment. Window 4, the control window, is the only one through which function requests may be issued. Windows 0 through 3 are available for data transfer between a PS processor and 432 memory.

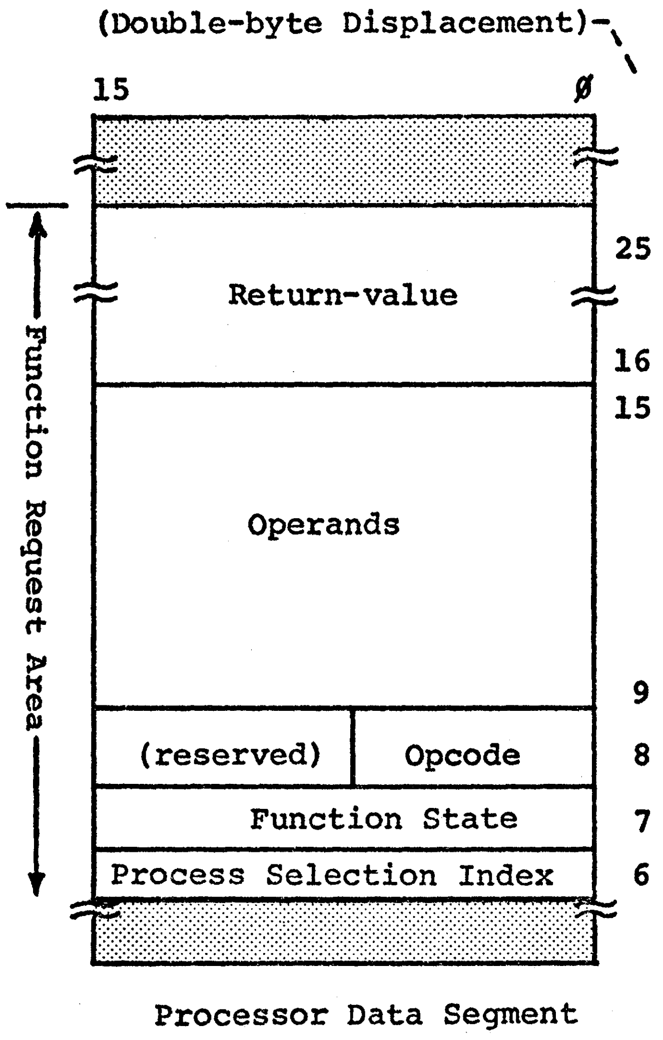


Figure 4-1 Function Request Area

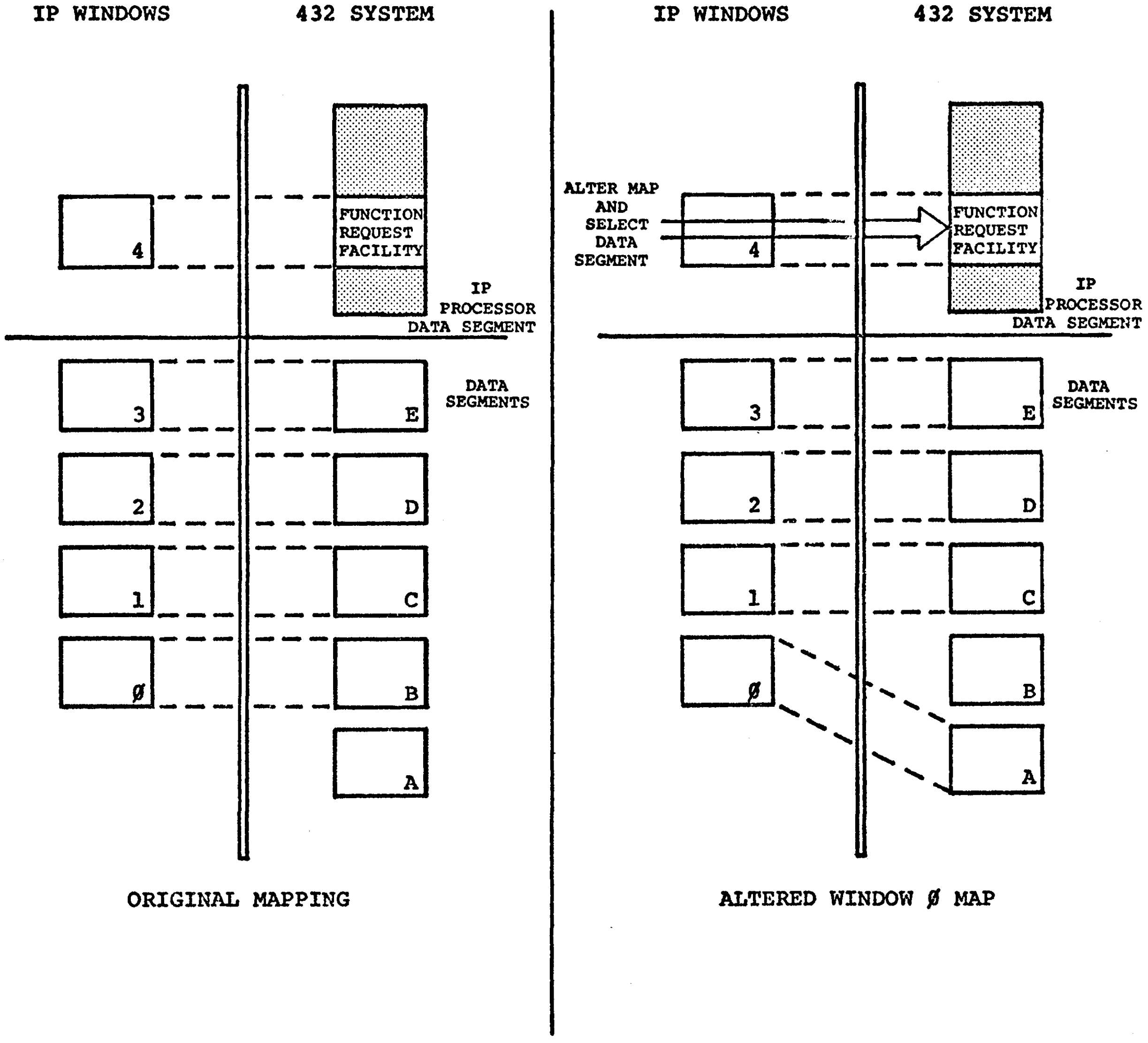


Figure 4-2 Function Example

4-2. FUNCTION REQUESTS

The performance of a function may be considered from the AP point of view as a sequence of three phases, as shown in figure 4-3. The IP controller, running on the AP, performs the first phase, requesting the execution of a function.

The IP executes functions serially; requesting execution of a second function before a prior function has been completed produces an undefined result. The function completion state subfield of the function state field (see appendix A) indicates the IP's readiness to accept a function request. A typical IP controller implementation will assign responsibility for requesting functions to a single routine (task) which will serialize the requests.

Given appropriate Peripheral Subsystem bus arbitration, function requests (which are identical to all windowed transfers) may be issued concurrently with other window activities. For example, consider a DMA controller driving a block mode transfer through window 0. If the DMA controller relinquishes the Peripheral Subsystem bus between transfer cycles, the IP controller (running on the Attached Processor) can use the bus for a function request (or for any other purpose).

PROCESS SELECTION

The IP controller must specify that a function be performed in one of the IP process environments which exist in the 432 system. To select a process, the IP controller must deposit a process selection index into a designated slot in the function request facility area of the processor data segment. With this index, and the process list in the IP's processor object, a process object can be located. The IP will attempt to qualify and lock the specified process as soon as a function opcode is written.

FUNCTION OPCODES

Each function is uniquely identified by a one-byte opcode (see appendix B). The act of writing into the opcode field triggers the execution phase of function performance. Therefore, the function's operands must be in place in the function request area before the opcode is transferred.

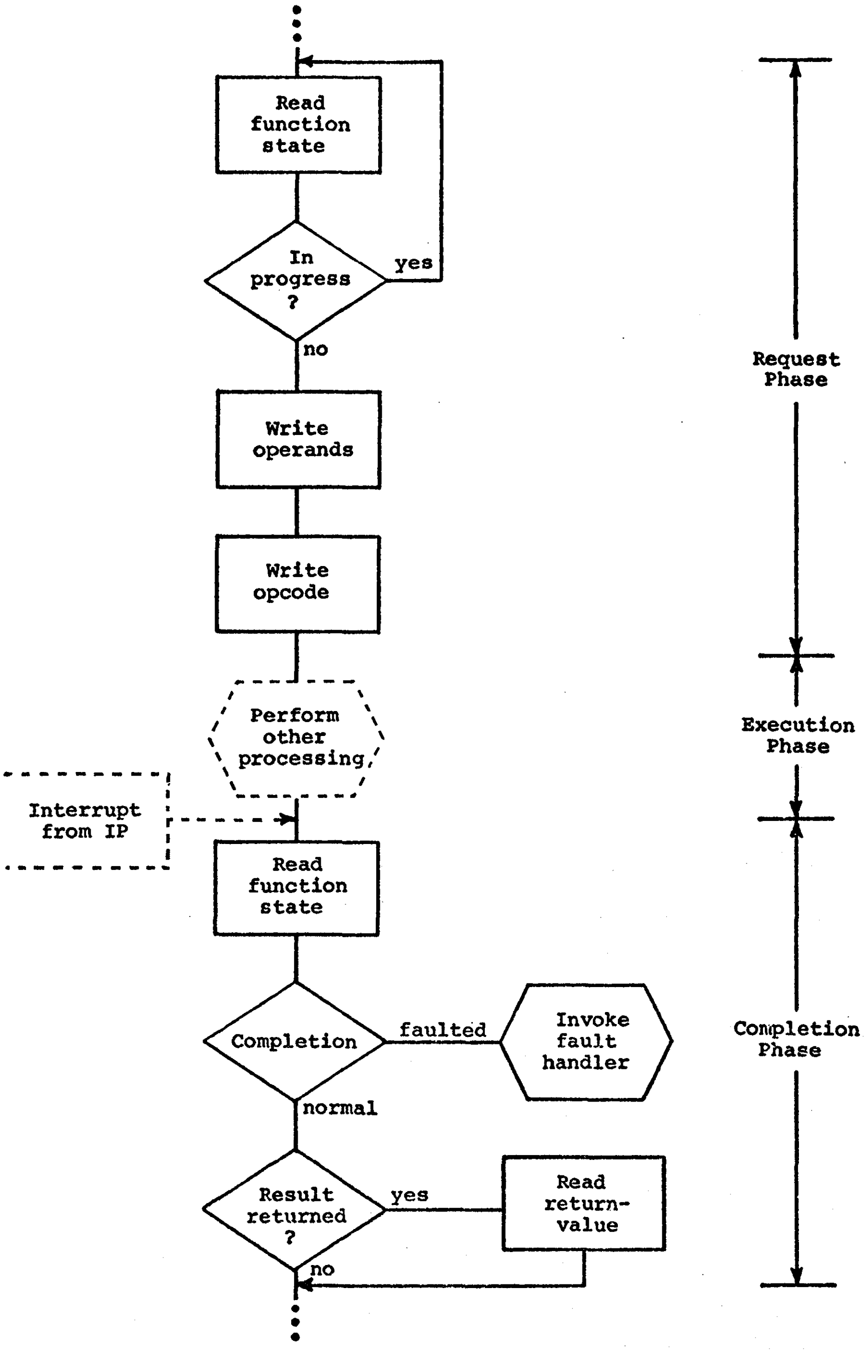


Figure 4-3 Function Performance Phases - AP View

FUNCTION OPERANDS

An Interface Processor function may require from zero to seven double-byte operands. The IP controller specifies a function's operands by writing values into locations of the operands field in the function request area. The first operand goes in the lowest-addressed location of the field and the remaining operands are written to successively higher-addressed locations (in some cases, one or more operand slots may be reserved and are skipped over). Each opcode implicitly identifies the number of operands required, so unused high-order locations in the operands field need not be initialized. See Appendix B for the function summary.

Interface processor functions accept three types of operands as illustrated in figure 4-4; all operand types are stored as double-bytes.

A short ordinal is a a 16-bit unsigned binary integer (range 0-65,535). This type of operand is typically used to specify a length, a displacement, an index, etc. For example, when the ALTER MAP AND SELECT DATA SEGMENT function is used to open a window, it requires a short ordinal operand that specifies the size of the subrange.

A bit field is a string of 16 bits that is divided into a number of subfields. The length, position and definition of each subfield varies according to the function. Subfields in a bit field operand to the ALTER MAP AND SELECT DATA SEGMENT function, for example, specify transfer mode, memory overlay, etc.

An object selector identifies an access descriptor for an object that is the function's actual operand. Figure 4-5 illustrates how the IP uses an object selector operand to obtain access to an object. The low-order subfield of the object selector identifies one of the four currently entered access segments associated with the selected context. The high-order subfield indexes one of the access descriptors in the entered access segment. The selected access descriptor refers, via the object table, to the object that is the actual function operand. This three-level address development is identical to GDP addressing. Note that the IP also performs the standard 432 type, rights and bounds checking as it develops the object's physical address from the object selector.

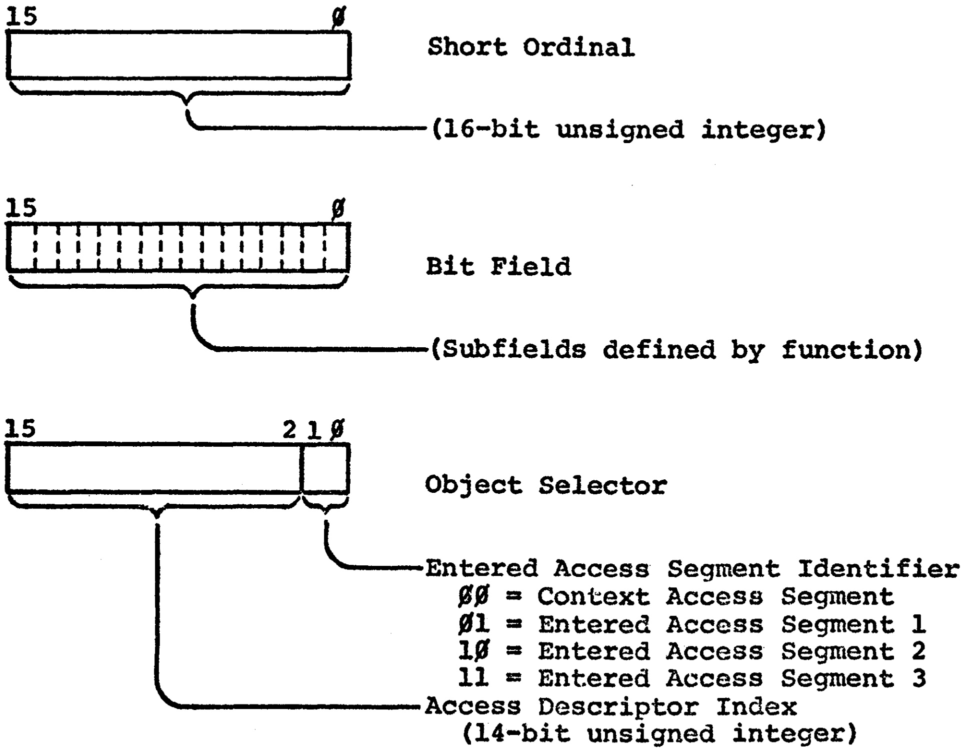

15
ø
Short Ordinal
(16-bit unsigned integer)
15
ø
Bit Field
(Subfields defined by function)
15
2 1 ø
Object Selector
Entered Access Segment Identifier
øø = Context Access Segment
ø1 = Entered Access Segment 1
1ø = Entered Access Segment 2
11 = Entered Access Segment 3
Access Descriptor Index
(14-bit unsigned integer)

Figure 4-4 Function Operand Types

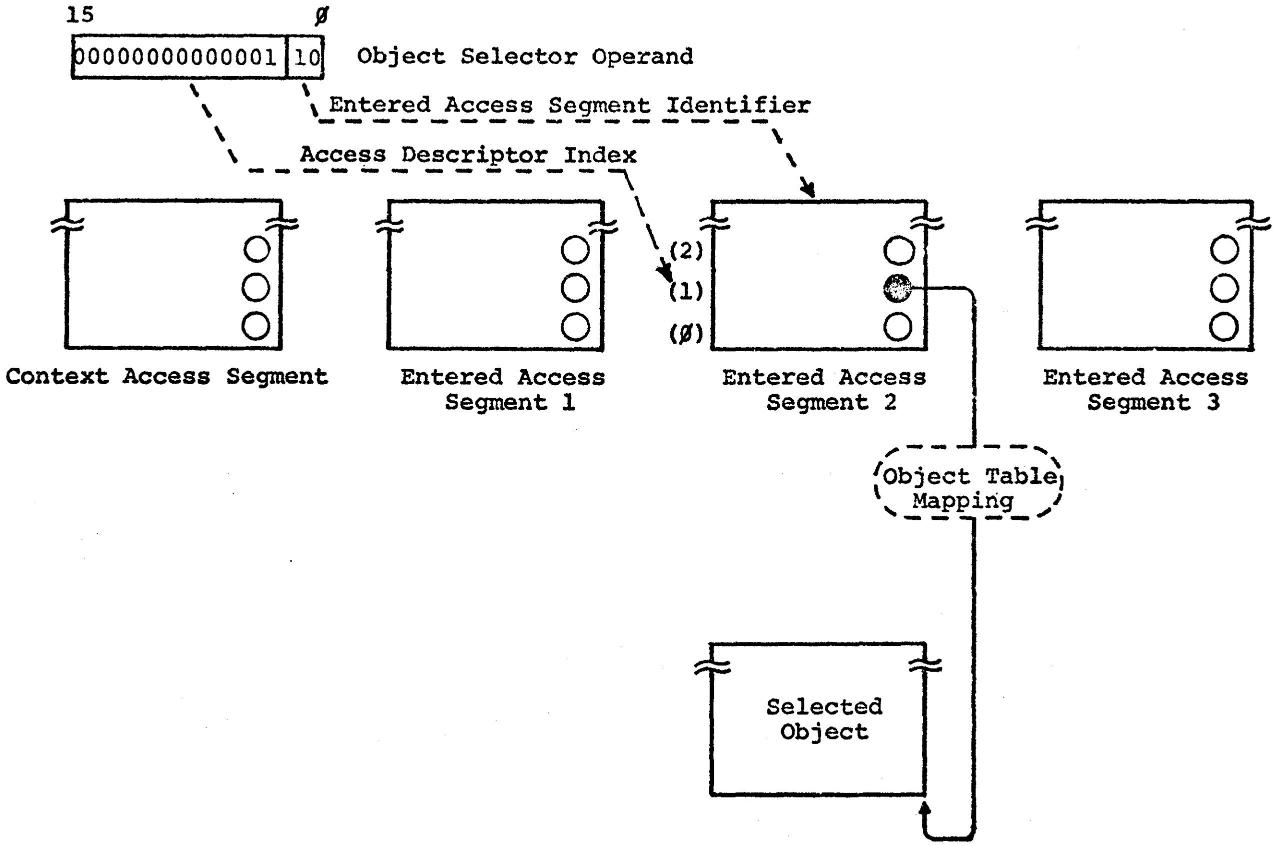


Figure 4-5 Object Selection

4-3. FUNCTION EXECUTION

The IP performs the actual execution of a function independent of the IP controller. Therefore the IP controller (an Attached Processor with associated IP control software) is free do other work after it has requested execution of a function (except that it must refrain from requesting a second function).

Although the IP's execution of any given function necessarily varies, figure 4-6 shows the basic sequence of steps that is common to most functions. Note that the IP checks for faults throughout execution.

Function execution begins when the IP detects that the opcode field of the function request area mapped by window 4 has been written. The IP sets the state of window 4 to "in-progress" during the function execution process to indicate that the function request facility is "in use". The IP reads the opcode from the function request area and decodes it. After decoding the opcode, the IP fetches the operands required by the function from the function request area. It then performs the operation and updates destination operands with the result(s). If the function produces a return-value, the IP writes it into the corresponding field of the function request area.

The IP terminates execution by updating the function completion state subfield and generating an interrupt (see appendix D for information on discriminating IP interrupts). The function completion state subfield indicates successful or faulted execution. The IP records additional information in one or more of the context, process and processor objects when it detects a fault during execution of a function.

4-4. FUNCTION COMPLETION

Normally the IP controller will use the IP's interrupt to detect function completion; it may also poll the function completion state subfield. In any case, the function completion state subfield must be examined to determine if the function completed successfully or faulted.

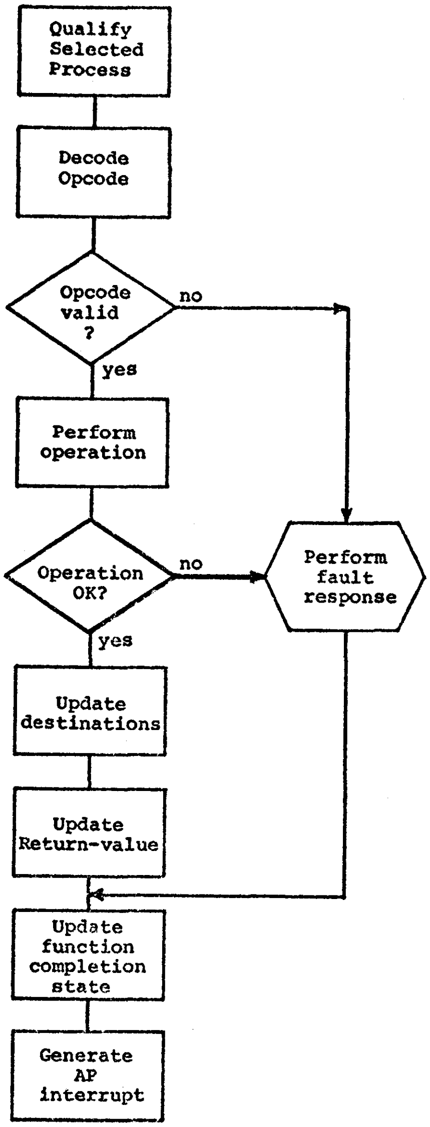


Figure 4-6 Basic IP Function Execution Flow

Successful execution of a function typically causes the alteration of a destination operand (that is, an actual operand; the operands field of the function request area is never changed by function execution). In addition, or alternatively, some functions produce a return-value. For example, the READ PROCESSOR STATUS AND CLOCK function returns the current values of the IP's system clock and status. The IP writes return-values into the results field of the function request area, where they may be inspected through window 4. The low-order byte of any return-value is stored in the lowest-addressed location of the field and any additional bytes are stored in consecutively higher locations. When the length of the return-value is less than the length of the return-value field, the content of excess high-order locations is undefined.

Appendix B provides the format and interpretation of the return-values produced by all functions. Several functions produce a standard type of return-value called a boolean. This is a one-byte value that indicates "true" or "false." The low-order bit of the value "true" is 1 and the low-order bit of the value "false" is 0. In either case the value of the upper seven bits of a boolean is undefined.

If a function faults, the contents of the return-value field is undefined. If a function completes successfully, but it does not produce a return-value, then the IP does not alter the content of the return-value field.

CHAPTER 5
PHYSICAL REFERENCE MODE

The preceding chapters of this manual have implicitly described the Interface Processor's logical reference mode, its normal mode of operation. The IP also provides physical reference mode. Physical reference mode is distinguished from logical reference mode by direct 24-bit base-plus-displacement addressing and a limited subset of functions. It may be characterized as a powerful and rudimentary tool to be utilized in exceptional circumstances such as system initialization (see appendix E) and post-mortem diagnostics. This chapter first describes reference mode switching—how physical mode is entered and exited. The second section covers addressing and functions in physical reference mode.

5-1. REFERENCE MODE SWITCHING

An Interface Processor can switch from physical reference mode to logical reference mode (and vice versa) only under carefully controlled circumstances.

An Interface Processor enters physical reference mode in response to assertion of its INIT line during system initialization (see iAPX 43203 VLSI Interface Processor Data Sheet, Order No. 171874) or upon receiving an "enter physical reference mode" IPC when in logical mode. Since a "send to processor" IPC requires an access descriptor with the proper right for the target processor's processor object, the ability of 432 software to place an IP in physical reference mode can be limited by restricting distribution of this right in IP processor object references. However, any 432 process with an access descriptor for a processor object with "broadcast to processors" rights can place all IPs into physical mode by broadcasting the "enter physical reference mode" IPC. Thus, processors should only be granted broadcast rights with careful precautions. Table E-1 shows the attributes of the IP windows after entering physical reference mode.

An Interface Processor exits physical reference mode and enters logical reference mode when it receives a local IPC (it ignores global IPCs in physical mode). This local IPC is considered a startup IPC. The response of IP is to qualify the processor, enter logical mode, and then respond to the IPC.

5-2. PHYSICAL REFERENCE MODE ADDRESSING

In physical reference mode the object reference attribute of a window is replaced by a 24-bit segment base address. Upon recognition of a subrange address reference the IP determines the transfer displacement as in logical reference mode. It forms the transfer address by adding the displacement to the segment base address. The 432 transfer length is always set to 2^{16} bytes so that no length of transfer faults can occur. No system objects are used in physical reference mode addressing.

Note that in physical reference mode, window 0 may be opened in either random or block transfer mode and window 1 may be opened onto either 432 memory space or the interconnect address space. An IP operating in physical mode may also change the characteristics of window 4, the control window.

5-3. PHYSICAL REFERENCE MODE FUNCTIONS

The IP controller may request execution of four functions in physical reference mode. These correspond closely, but are not always identical, to logical reference functions. The request, execution, fault handling, and completion phases of physical reference mode operations are similar to the logical reference mode counterparts.
See the function summary in Appendix B for detailed descriptions of the operation of these functions.

The physical reference mode functions are
o SET PERIPHERAL SUBSYSTEM MODE;
o READ PROCESSOR STATUS AND CLOCK
o SEND TO PROCESSOR
o ALTER MAP AND SELECT PHYSICAL SEGMENT.

CHAPTER 6
FAULTS

This chapter describes IP faults, exceptional conditions which can occur as the IP performs functions. In general, the IP fault philosophy follows that of the GDP: the processor detects and contains faults so they do not affect other processes or processors in the 432 system. The response to a fault, i.e. fault handling, is not predefined and may be tailored through software to the needs of the 432 system user. The IP's dual role in the 432 system and in the Peripheral Subsystem requires that the strategy for handling faults is somewhat different than for the GDP.

6-1. FAULT REPORTING

When a fault occurs, the IP records information about the fault in a fault information area. Faults are distinguished by a fault code and an operator ID recorded in the fault information area. The fault codes are specified in Appendix C. The operator IDs are specified in Appendix B. The operator ID designates the IP function which was executing when the fault was encountered. A unique operator ID corresponds to each IP function code. Note that the values for the function codes are not the same as the values for the corresponding operator IDs.

When the IP has deposited the information in the respective fault information area and updated the function state, the IP interrupts the AP to inform it of the fault. The AP may check the function state field of the function request facility to acquire the field of bits which contains the fault level. If the IP has faulted, the AP examines the corresponding fault information area for more detail.

For faults which occurred during the execution of a function with a sequence of steps, like SEND or RECEIVE, the IP records the execution state when the function faulted. This information allows the time when the fault occurred to be specified more precisely. Then, software which handles the fault can respond in the most appropriate manner. The execution state information is necessary for software completion of a partially executed function.

The IP records fault information in various areas of IP process and processor objects (refer to Appendix A for detailed description of these fault information areas). There are three categories of IP operation in which faults may be generated: physical reference mode, logical reference mode, and window-mapped data transfer. Each of these modes utilizes specific fault information areas to report faults.

PHYSICAL MODE

Information about faults which occur in physical reference mode is recorded in the processor fault information area of the IP processor object. The function state is set to "context-level fault" when a physical reference mode fault is encountered and an AP interrupt is generated.

LOGICAL MODE

Information about faults which occur in logical reference mode is recorded in appropriate portions of the IP process and processor objects. Each IP process object contains two fault information areas: one for context-level fault information and one for process-level fault information. The IP processor object contains a fault informations area for processor-level fault information.

Depending on the severity level (context, process, or processor) of a fault and the current state of the process and processor, an IP selects an area to be used to record the fault information. The method an IP uses to decide the appropriate site to record fault information is shown in Figure 6-1. Successive faults, encountered during fault recording, reflect the fault state to higher levels of severity until, finally, an IP can no longer continue and must issue the FATAL signal (see iAPX 432 VLSI Interface Processor Data Sheet, Order Number 171874).

CATEGORIES OF LOGICAL MODE FAULTS

There are three categories of logical mode faults, listed in increasing order of severity:

- Context-level faults
- Process-level faults
- Processor-level faults

Context-Level Faults

Context-level faults are the least severe of the IP logical mode faults. A context-level fault arises from exceptions which can be confined to the context in which the IP is operating. The IP may fault when attempting to execute a function or during the movement of data through one of the windows. One example of a context-level fault is the condition which occurs when a request to the function facility contains an erroneous function code. In this case, the IP can detect and report the fault before any execution of a function is begun.

When the IP detects a context-level fault, it places information about the fault in the context-level fault information area of the process object, sets the function state to "context-level fault", and interrupts the Attached Processor. A context-level fault can only be generated by an IP which is bound to a process. If a second fault occurs while handling a context-level fault it is handled like a process-level fault.

Response to context-level faults can usually be performed by IP controller software running in the Peripheral Subsystem. The conditions which generated these faults are contained in a limited portion of the IP's 432 environment.

Process-Level Faults

Process-level faults are generated when an exceptional condition is detected which prohibits further IP execution in the faulted process environment. Some situations when process-level faults are generated are:

- System level consistency failures.
- Normal requests to the operating system interface.
- User errors, which may be misuse of the operating system interface.

When an IP encounters a process-level fault, the IP:

- Records information about the fault in the IP process' process-level fault information area.
- SENDs the faulted process to a fault port.
- Updates the function state to "process-level fault".
- Interrupts the Attached Processor.

If a second fault occurs while the IP is handling a process-level fault, this is considered a processor-level fault. If the IP encounters a fault of process-level severity when it is not bound to a process, the IP treats the situation as a processor-level fault.

The fault port is serviced by a 432 fault handling process where one of four actions may be taken:

- Correct the reason for the fault and complete any partially performed function by completing the unfinished steps.
- Correct the reason for the fault, rewind any partially performed function steps, and then retry the function.
- Decide to reflect the process-level fault to the context-level.
- "Crash" the system.

The first two actions represent the method that an operating system can use to extend the 432 architecture. For example, an operating system's virtual memory implementation considers a "storage not associated" fault as a normal occurrance and retrieves the missing memory segment. With the segment available, the fault handler can decide to simulate the completion of the function or unwind the partially completed function and rerun it.

Processor-Level Faults

Processor-level faults, the most severe level of faults, occur when an IP detects a condition which jeopardizes further operation by the processor. Bus errors and alarms are examples of such occurrences. In response to the first processor-level fault encountered, the IP reports the fault in the fault information area of the processor data segment, updates the processor status to "faulted", and signals an interrupt to inform the attached processor. If a second processor-level fault occurs before the AP has recorded the fault information, the IP closes all five of its windows into 432 memory, including the control window, signals that a fatal error has occurred and indicates that the Peripheral Subsystem should be reset (see FATAL/ and PSR pin descriptions in the iAPX 43203 Interface Processor Data Sheet, Order Number 171874).

WINDOW-MAPPED DATA TRANSFER

Information about faults which occur during data transfer through the windows is recorded in the mapping facility fault information area contained in the IP processor object. This information is accessible to the AP through the control window. Each window (0 through 4) has a separate fault information area. When the fault occurs, the IP deposits the fault information, closes the window, puts the window in the error state, and interrupts the Attached Processor. Only open windows can generate window mapping faults.

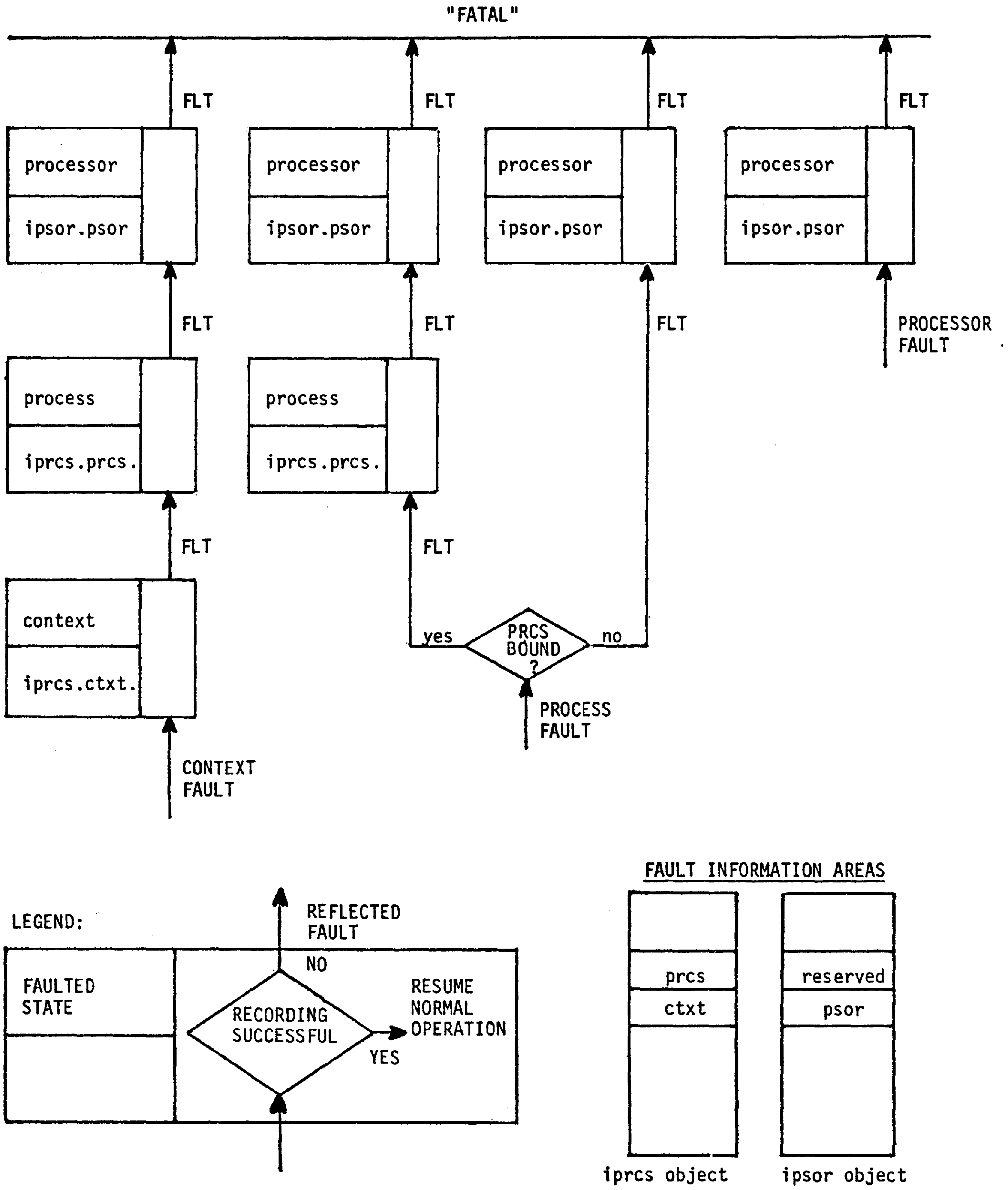


Figure 6-1 Fault Reporting State

6-2. FAULT HANDLING

When an IP process encounters a process-level fault, it is automatically sent to a 432 fault port to await service. A fault handling 432 process is designated to service the faulted processes waiting at the fault port. By design, IPs and GDPs share a common base architecture, so IP faults may often be handled by software similar to that used to service GDP faults. In cases where unique IP attention is required, a special fault port must be constructed to which faulted IP processes may be selectively re-sent and then serviced by AP and/or GDP software.

APPENDIX A
SYSTEM OBJECT STRUCTURES

The object structures of Interface Processors are described below. The only objects structures described are for those whose form or interpretation differ from GDP object structures. Note that the values found in the length fields in the various objects described below are encoded as "actual length minus 1" in bytes. Also note that the object indices refered to below are of the same format as object selectors with the entry access segment index subfield uninterpreted. The displacement subfield is interpreted as an index into the associated domain access segment.

A-1. CONTEXT OBJECTS

In the most general terms, contexts for Interface Processors and General Data Processors serve the same purpose. They are used to represent an access environment in which process execution can take place. On closer inspection, however, the differences are significant. For example, with Interface Processors there is no concept of a sequential instruction stream. Instead the only instructions executed by Interface Processors are functions requested, one at a time, by software executing on the associated Attached Processor. At a mundane level, this means that Interface Processor contexts need not provide access to instruction segments or operand stacks. More significantly, without a sequential instruction stream there are no concepts of intracontext or intercontext control flow either. This results in the binding between Interface Processor processes and contexts being static. In fact, context access and data segments are refinements of the corresponding process access and data segments respectively.

Given these differences, an Interface Processor context represents the access environment available within the 432 system to the logical process being executed on the logical processor comprised of the Interface Processor and the associated Attached Processor. The operators provided by the Attached Processor affect the contents of data segments in this environment via the address mapping facility of the Interface Processor. The operators provided by the Interface Processor affect this environment via the function request facility of the Interface Processor.

A context object is represented by a context access segment and an associated context data segment.

Context Access Segments

Diagrammatically, a context access segment is structured as shown below.

```
                   =                =
             entry !                !
                   !----------------!
                 8 !  domain AD    -!---> domain of definition
                   !----------------!
                   !     AS  AD    -!---> entry access segment 3
                   !----------------!
                   !     AS  AD    -!---> entry access segment 2
                   !----------------!
                   !     AS  AD    -!---> entry access segment 1
                   !----------------!
                   !  context AD   -!---> context
                   !----------------!
                   !       AD      -!---> message
                   !----------------!
                   !       AD      -!---> reserved
                   !----------------!
      context      !       AD      -!---> reserved
      access       !----------------!
      segment ---> 0 !  data seg. AD -!---> context data segment
                   !----------------!
```

The context access segment, context data segment, and domain access descriptors in the context must be created without delete rights. The entry access segment entries never bear delete rights.

The base rights field of a context access segment access descriptor is interpreted in the same manner as for all objects of base type access segment. The system rights field of a context access segment access descriptor is uninterpreted.

Context Data Segments

The only processor interpreted field in the context data segment is the process status field which contains a combination of process and context status. The form and interpretation of this field are described in the process data segment section.

The base rights field of a context data segment access descriptor is interpreted in the same manner as for all objects of base type data segment. The system rights field of a context data segment access descriptor is uninterpreted.

A-2. PROCESS OBJECTS

Logically, a process is the execution by a processor of an instruction stream within a specific environment. In a combined Attached Processor/Interface Processor system, the IP process object extends the execution environment of an AP process to logically include a specific domain in the 432 address space. The execution point moves, of course, as each instruction is executed because a new instruction is automatically specified. Occasionally, as the result of instruction execution, a new instruction stream within the Attached Processor software is specified. Unless the AP process should indicate its termination, the execution point continues to move in this manner forever. There is thus a close and long-term association between the environment provided by an interface process and a particular AP process. When a new AP process specifies a function request, an Interface Processor makes the associated interface process' execution environment available.

A process object is represented by a process access segment and an associated process data segment.

Process Access Segments

The hardware-recognized internal structure of a process access segment is shown below.

```
                 =               =
           entry !               !
                 !---------------!
                 !               !
                 !    refined    !
                 =    context    =
                 !    access     !
                 !    segment    !
              12 !               !
                 !---------------!
              11 !  carrier AD  -!---> surrogate carrier
                 !---------------!
                 !  carrier AD  -!---> current carrier
                 !---------------!
                 !   port  AD   -!---> current port
                 !---------------!
                 !      AD      -!---> current message
                 !---------------!
                 !      AD      -!---> reserved
                 !---------------!
                 !   port  AD   -!---> fault port
                 !---------------!
                 !   port  AD   -!---> dispatching port
                 !---------------!
                 !  carrier AD  -!---> process carrier
                 !---------------!
                 !      AD      -!---> reserved
                 !---------------!
                 !   AS   AD    -!---> global access segment
                 !---------------!
process          ! context AD   -!---> context
access           !---------------!
segment   ---> 0 ! data seg. AD -!---> process data segment
                 !---------------!
```

The base rights field of a process access segment access descriptor is interpreted in the same manner as for all objects of base type access segment. The system rights field of a process access segment access descriptor is uninterpreted.

Process Data Segments

The basic structure of a process data segment is shown below.

```
                  =                =  double byte
                  !                !  displacement
                  !----------------!
                  !                !
                  !    refined     !
                  =    context     =
                  !     data       !
                  !    segment     !
                  !                !  90
                  !----------------!
                  !                !
                  !    process     !
                  =     fault      =
                  !  information   !
                  !                !  77
                  !----------------!
                  !                !
                  !    context     !
                  =     fault      =
                  !  information   !
                  !                !  64
                  !----------------!
                  !                !
                  =    reserved    =
                  !                !  9
                  !----------------!
                  !   process ID   !
                  !----------------!
                  !                !
                  !-              -!
                  !                !
                  !-   reserved   -!
                  !                !
                  !-              -!
                  !                !  4
                  !----------------!
                  !                !
                  !-   q value    -!
                  !                !
                  !----------------!
process           ! process status !
data              !----------------!
segment ------>   !  object lock   !  0
                  !----------------!
```

The format and interpretation of the object lock field is the same as for GDPs.

The organization of the process status field is shown below.

```
----------------------------
!x!x!   9 bits   !x!x!x!x!x!
----------------------------
 ! !       !       ! ! ! ! !
 ! !       !       ! ! ! ! !— bound
 ! !       !       ! ! ! !---- waiting for message
 ! !       !       ! ! !------ process faulted
 ! !       !       ! !-------- reserved
 ! !       !       !---------- context faulted
 ! !       !------------------ reserved
 ! !-------------------------- one vector only
 !---------------------------- first port operation completed
```

The bound bit is interpreted as follows:

0 - this process is bound to a processor
1 - this process is not bound to a processor

The interpretation of the context and process faulted subfields are as follows:

0 - not faulted
1 - faulted

The format and interpretation of the waiting for message, one vector only, and first port operation completed subfields are the same for IPs as they are for GDPs.

Fault information for context, process, and processor level faults has the same organization. Process objects contain fault information for context and process level faults. Processor objects contain fault information for processor level faults. Access to the context fault information is made available to a context via the software convention of providing a refinement for it in a known entry of the process global access segment. The process fault information area in the process object is used when a process-level fault occurs and a process is bound to the processor. The processor fault information area in the processor object is used when a process level fault occurs and a process is not bound to the processor. The organization of the fault information area is described in Appendix C, the Fault Summary.

The base rights field of a process data segment access descriptor is interpreted in the same manner as for all objects of base type data segment. The system rights field of a process data segment access descriptor is uninterpreted.

A-3. PROCESSOR OBJECTS

An 432 Interface Processor consists of two cooperating processing elements: a mapping facility and a function request facility. The mapping facility translates Peripheral Subsystem addresses into 432 system addresses. The function request facility executes the operator set described in Appendix B. The mapping facility and the function request facility can run in parallel.

A processor object is represented by a processor access segment, an associated processor data segment.

Processor Access Segments

Processor access segments are organized as shown below.

```
                   =                =
             entry !                !
                   !----------------!
                   !                !
                   !    process     !
                   =   selection    =
                   !      list      !
                21 !                !
                   !----------------!
                20 !                !
                   !-              -!
                   !                !
                   !-    mapped    -!
                   !      data      !
                   !-   segments   -!
                   !                !
                   !-              -!
                   !                !
                   !----------------!
                   !       AD      -!---> reserved
                   !----------------!
                   !       AD      -!---> reserved
                   !----------------!
                   !       AD      -!---> reserved
                   !----------------!
                   !    port  AD   -!---> normal port
                   !----------------!
                   !   carrier AD  -!---> surrogate carrier
                   !----------------!
                   !   carrier AD  -!---> normal carrier
                   !----------------!
                   !    port  AD   -!---> current port
                   !----------------!
                   !       AD      -!---> current message
                   !----------------!
                   !       AD      -!---> reserved
                   !----------------!
                   !  data seg. AD -!---> control window
                   !----------------!
                   !   carrier AD  -!---> processor carrier
                   !----------------!
                   ! objtab dir AD -!---> object table directory
                   !----------------!
                   ! commo seg. AD -!---> global communication segment
                   !----------------!
                   ! commo seg. AD -!---> local communication segment
                   !----------------!
 processor         !   carrier AD  -!---> current process carrier
  access           !----------------!
  segment   ---> 0 !  data seg. AD -!---> processor data segment
                   !----------------!
```

The base rights field of a processor access segment access descriptor is interpreted in the same manner as for all objects of base type access segment. The low order bit of the system rights field of a processor access descriptor is interpreted as follows:

0 - an interprocessor message may not be broadcast via the global communication segment of this processor
1 - an interprocessor message may be broadcast via the global communication segment of this processor

The mid order bit of the system rights field of a processor access descriptor is interpreted as follows:

0 - an interprocessor message may not be sent to this processor
via the local communication segment of this processor
1 - an interprocessor message may be sent to this processor via the local communication segment of this processor

The high order bit of the system rights field of a processor access descriptor is uninterpreted.

Processor Data Segments

The intended use of this data segment is as instance specific control information, for recording a copy of the processor-resident information contained in the function request facility and the mapping facility, for recording fault information, and as randomly addressable scalar working storage. The copy of processor-resident information in the processor data segment is updated by the processor whenever a significant state change to that information occurs (i.e., function completion or block transfer completion). The area above double byte displacement four is made visible to Attached Processor software through the control window (window 4).

The information in the processor data segment is organized as shown in the diagram below.

```
                                     double byte
                 =                 = displacement
                 !-----------------!
                 !                 !
                 !     control     !
                 =     window      =
                 !      area       !
                 !                 ! 4
                 !-----------------!
                 !    reserved     !
                 !-----------------!
                 ! cur. prcs idx.  !
                 !-----------------!
processor        !   psor status   !
data             !-----------------!
segment    ----> !   object lock   ! 0
                 !-----------------!
```

The processor status field is shown below.

```
-------------------------
!   8 bits  !x!x!x!x!xxxx!
-------------------------
      !      ! ! ! !  !
      !      ! ! ! !  !--- processor state
      !      ! ! ! !------ faulted
      !      ! ! !-------- reference mode
      !      ! !---------- stopped
      !      !------------ broadcast accept. mode
      !------------------- processor ID
```

The processor state subfield is interpreted as follows:

0000 - idle
0001 - process execution
0010 - 1111 - reserved

The interpretation of the faulted subfields is as follows:

0 - not faulted
1 - faulted

The reference mode subfield specifies whether the references in function requests are logical or physical. In logical reference mode, function request references are relative to the four-component access environment generated by the current context. In physical reference mode, function request references are simply 24-bit physical addresses. The reference mode subfield is interpreted as follows:

0 - using physical mode
1 - using logical mode

The stopped bit is interpreted as follows:

0 - running
1 - stopped

The broadcast acceptance mode bit is interpreted as follows:

0 - broadcast interprocessor messages are not being accepted and acknowledged
1 - broadcast interprocessor messages are being accepted or acknowledged

Note that the processor ID fields in the processor data segment and the local communication segment are filled in by the associated processor at initialization time from externally read information.

The base rights field of a processor data segment access descriptor is interpreted in the same manner as for all segments of base type data segment. The system rights field of a processor data segment access descriptor is uninterpreted.

Control Window Area -

The control window area consists of several major subareas and several minor ones. The primary purpose of these areas is to provide Attached Processor software access to state information describing recent state changes in the function request facility and the mapping facility and occurances of asynchronous events.

```
                                  double byte
                     =          = displacement
                     !----------! 80
                     !          !
                     !          !
                     = reserved =
                     !          !
                     !          ! 77
                     !----------!
                     !          !
                     !processor !
                     =  fault   =
                     !information!
                     !          ! 64
                     !----------!
                     ! selected state ! 63
                     !----------!
                     ! selected idx.  ! 62
                     !----------!
                     !          ! 65
                     ! mapping  !
                     ! facility !
                     =  fault   =
                     !information!
                     !          ! 52
                     !----------!
                     !          !
                     ! mapping  !
                     = facility =
                     !          ! 28
                     !----------!
                     ! reserved ! 27
                     !----------!
                     ! IPC fun. req.  ! 26
                     !----------!
                     !          !
                     ! function !
                     = request  =
                     ! facility !
                     !          ! 6
                     !----------!
                     ! reserved !
                     !----------!
                     ! reconfig. state!
                     !----------!
                     ! disp. state    !
                     !----------!
                     ! alarm state    !
                     !----------!
control              ! IPC state!
window               !----------!
area          ---->  ! PS state ! 0
                     !----------!
```

Peripheral Subsystem State Field -

The organization of the Peripheral Subsystem state field is shown below.

```
------------------
! 12 bits !x!xx!x!
------------------
    !       !  ! !
    !       !  ! !-- write sample delay
    !       !  !---- xack delay
    !       !------- interrupt inhibit
    !--------------- reserved
```

The write sample delay field and the xack delay field program the characteristics of the IP component interface to the Peripheral Subsystem. See the iAPX 43203 VLSI Interface Processor Data Sheet, Order Number 171874-001 for details.

If the interrupt inhibit field is a 1 then the IP will inhibit normal function complete interrupts but will continue pass all other interrupts to the AP. If the interrupt inhibit field is 0 then the IP will report successful function completions with interrupts.

IPC State Field -

The IPC state field is used to indicate that the processor has responded to an interprocessor communication signal and signalled the associated Peripheral Subsystem via interrupt. It has the following organization.

```
-----------------
!  14 bits  !x!x!
-----------------
      !      ! !
      !      ! !-- local IPC response
      !      !---- global IPC response
      !----------- reserved
```

With either IPC response flag, a value of zero indicates that no such response has occured and a value of one indicates that such a response has occured.

Alarm, Dispatching, and Reconfiguration State Fields -

The alarm, dispatching ("select process"), and reconfiguration state fields are used to indicate that the processor has responded to that type of signal and signalled the associated Peripheral Subsystem via interrupt. Each has the following organization.

```
----------------
!  15 bits  !x!
----------------
      !      !
      !      !-- response
      !--------- reserved
```

With the response flag, a value of zero indicates that no such response has occured and a value of one indicates that such a response has occured.

Function Request Facility Area -

The function request facility is the part of the Interface Processor which accepts function requests and performs the requested function. The function request facility area of the processor data segment contains a copy of the processor-resident information related to the current or most recent function requested. As shown below, the area consists of five contiguous parts. The first part contains the process selection index for the execution environment in which the function should be performed. The second part contains the function state information. The third part contains the op-code of the operator requested. The fourth part contains the operands operated upon in performing the requested function. The fifth part is used to record the result of the requested function.

```
=                 = double byte
!-----------------! displacement
!                 ! 25
!    function     !
!     result      !
!                 ! 16
!-----------------!
!                 !
!    operands     !
!                 !
!                 ! 9
!-----------------!
!    operator     ! 8
!-----------------!
! function state  ! 7
!-----------------!
!   prcs idx.     ! 6
!-----------------!
=                 =
```

Function State Field -

The function state field is used to describe the current state of the function request facility. It has the following organization.

```
------------------------
!  8 bits  !xx!x!x!xxxx!
------------------------
      !       ! ! !  !
      !       ! ! !  !-- function completion state
      !       ! ! !----- SEND completion state
      !       ! !------- RECEIVE completion state
      !       !--------- fault level
      !----------------- reserved
```

The interpretation of the function completion state subfield is as follows:

0000 - function completed
0001 - function in progress
0010 - 1111 - reserved

The interpretation of the SEND or RECEIVE completion state subfields is as follows:

0 - completed
1 - blocked

The fault level subfield indicates whether a fault which has occured is context-level, process-level, or processor-level. The fault handler requires this information in order to know where the fault information has been stored. The interpretation of the fault level subfield is as follows:

00 - none
01 - context-level fault
10 - process-level fault
11 - processor-level fault

Mapping Facility Area -

The mapping facility consists of five map entries capable of supporting the random mapping of five non-overlapping address subranges from the Peripheral Subsystem into corresponding 432 data segments. One of these map entries (entry 0) is capable of supporting block transfer as well as random mapping. One map entry (entry 1) is capable of supporting mapping into the 432 interconnection address space as well as random mapping. One map entry (entry 4) and its associated Peripheral Subsystem address subrange always maps onto the processor data segment. The two major purposes of this subrange are to capture references to the function request facility and to allow Attached Processor software to read current status information. When operands are read from this subrange or written into this subrange, the processor data segment is accessed. Data written into the part of the subrange representing the function request facility is captured when no function is in progress. During function execution, Attached Processor software must not make further function requests.

At the base of the mapping facility area, the extra information for supporting block transfer via map entry 0 is recorded in a data structure with the following organization.

```
=                  = double byte
!------------------! displacement
!    reserved      ! 31
!------------------!
!   P. S. disp.    !
!------------------!
!   432 disp.      !
!------------------!
!   block count    ! 28
!------------------!
=                  =
```

When the transfer mode subfield of the entry state field for map entry 0 indicates that it is in block transfer mode, the processor-resident copy of the block count field indicates the number of bytes remaining to be transferred for transfer termination to occur normally (i.e., upon count runout). Whenever normal transfer termination occurs, both copies of the block count field are zero. Whenever normal transfer termination does not occur, such as in the case of faults, both copies of the block count field indicate the number of remaining, but not transferred, bytes.

When the transfer mode subfield of the entry state field for map entry 0 indicates that it is in block transfer mode, the processor-resident copy of the 432 displacement field indicates the displacement into the associated data segment of the next byte to be transferred.

When the transfer mode subfield of the entry state field for map entry 0 indicates that it is in block transfer mode, the processor-resident copy of the Peripheral Subsystem displacement field indicates the displacement into the associated Peripheral Subsystem address range of the next byte to be transferred.

Any difference between the values of the two displacement fields accounts for data in the processor-resident buffers which was not successfully transfered.

Above the block transfer information, a copy of the information contained in each of the processor-resident map entries (0 through 4) is represented by a data structure with the following organization.

```
=                  = double byte
!------------------! displacement
!    base disp.    ! 4
!------------------!
!       mask       !
!------------------!
!   base address   !
!------------------!
!    entry state   ! 0
!------------------!
=                  =
```

The entry state field is used to describe the current state of the given map entry. It has the following organization.

```
-----------------------
! 9 bits !x!xx!xx!x!x!
-----------------------
    !      !  !  !  !  !
    !      !  !  !  !  !-- map valid
    !      !  !  !  !---- transfer mode
    !      !  !  !------ transfer direction
    !      !  !--------- transfer state
    !      !------------ memory overlay
    !------------------- reserved
```

The 1-bit map valid subfield indicates whether or not this map entry is currently in use. If the bit is zero, this map entry is not used in Peripheral Subsystem address inspection. If the bit is one, this map entry is used in Peripheral Subsystem address inspection. The processor-resident copy of this subfield is checked by the mapping facility each time a Peripheral Subsystem address is received for inspection.

For map entry 0, the 1-bit transfer mode subfield indicates whether this map entry is in random or block transfer mode. A value of zero indicates that this map entry is in random mode. A value of one indicates that this map entry is in block transfer mode. For map entry 1, the 1-bit transfer mode subfield indicates whether this map entry maps Peripheral Subsystem addresses into the 432 address space or the interconnection address space. A value of zero indicates that this map entry is in 432 mapping mode. A value of one indicates that this map entry is in interconnection mapping mode. For other map entries, the setting of this subfield causes a fault.

The 2-bit transfer direction subfield indicates the types of read/write requests from the associated Peripheral Subsystem which are valid with respect to this map entry. The low order bit of the transfer direction subfield is interpreted as follows:

0 - reading may not occur
1 - reading may occur

The high order bit of the transfer direction subfield is interpreted as follows:

0 - writing may not occur
1 - writing may occur

Note that both bits may not be set when setting block transfer mode.

The 2-bit transfer state subfield indicates the state of the transfer. It is encoded as follows:

00 - transfer in progress
01 - transfer terminated upon count runout
10 - transfer termination forced
11 - transfer termination upon fault

The 1-bit memory overlay subfield indicates whether or not the Peripheral Subsystem address subrange associated with this map entry overlays physical memory in the Peripheral Subsystem. If physical memory is overlayed, whenever an address is mapped via this entry a Peripheral Subsystem bus protocol is employed which prevents that overlayed memory from responding. A value of zero indicates that no memory is overlayed. A value of one indicates that memory is overlayed.

The base address field is used to specify the starting address of the Peripheral Subsystem address subrange mapped by this map entry. Subranges are 2**n bytes in length with n being in the range zero to sixteen. A subrange of a given power of two in size must appear on an addressing boundary of the same power of two (e.g., a 16 byte subrange must begin on a 16 byte boundary). Stated another way, a subrange of 2**n bytes in length will thus have a starting address containing at least n trailing zeros. Base addresses are always an integer multiple of an integer power of two (i.e., m*2**n). The n is as described above. The m is any integer such that the above conditions hold and the value of the starting address is limited to the range 0 to 65,535.

The mask field contains a mask which is used to specify the size of the Peripheral Subsystem address subrange to be mapped by this map entry. The mask is composed of two contiguous bit string subfields. The higher-order bit string contains all ones. The lower-order bit string contains all zeros. The mapped address subrange is 2**(number of zeros in the lower-order bit string) bytes in length beginning at the starting address.

The base displacement field contains the byte displacement into the 432 segment used to construct a refinement of a data segment. See Figure 3-2 for an illustration of the role of a window's base displacement in forming a refinement.

Mapping Facility Fault Information Area -

The mapping facility fault information area consists of an entry fault code and fault displacement pair for each map entry. Diagrammatically, the fault information for each map entry appears as shown below.

```
=               =
!---------------!
!  fault disp   !
!---------------!
!  fault code   !
!---------------!
=               =
```

Each entry fault code field is used to record the cause of the last fault associated with that map entry. It has the following organization.

```
-----------------------------
!x! 6 bits !x!x!x!x!x!x!x!x!x!
-----------------------------
 !    !      ! ! ! ! ! ! ! ! !
 !    !      ! ! ! ! ! ! ! ! !-- read/write
 !    !      ! ! ! ! ! ! ! !---- bus error
 !    !      ! ! ! ! ! ! !------ access rights
 !    !      ! ! ! ! ! !-------- segment bound
 !    !      ! ! ! ! !---------- memory overflow
 !    !      ! ! ! !------------ access direction
 !    !      ! ! !-------------- post termination
 !    !      ! !---------------- partial block overflow
 !    !      !------------------ block overflow
 !    !------------------------- reserved
 !------------------------------ block termination
                                 (internal use)
```

The 1-bit read/write subfield indicates whether the associated fault was caused by a read request or a write request. A value of zero indicates that the fault was caused by a read request. A value of one indicates that the fault was caused by a write request.

The 1-bit bus error subfield indicates whether or not the associated fault was caused by a 432 bus error. A value of zero indicates that the fault was not caused by a bus error. A value of one indicates that the fault was caused by a bus error.

The 1-bit segment bound subfield indicates whether or not the associated fault was caused by a segment bounds violation. A value of zero indicates that the fault was not caused by a segment bounds violation. A value of one indicates that the fault was caused by a segment bounds violation.

The 1-bit memory overflow subfield indicates whether or not the associated fault was caused by a memory overflow. A memory overflow occurs when the sum of the physical base address in bytes of a segment being accessed plus the byte displacement to the operand being accessed exceeds 16,777,215 (i.e. 2**24-1). A value of zero indicates that the fault was not caused by a memory overflow. A value of one indicates that the fault was caused by a memory overflow.

The 1-bit access direction subfield indicates whether or not the associated fault was caused by an access direction error. An access direction error occurs when the transfer direction subfield of the corresponding map entry state indicates that the requested access direction (either read or write) is invalid. A value of zero indicates that the fault was not caused by an access direction error. A value of one indicates that the fault was caused by an access direction error.

The 1-bit post termination subfield indicates whether or not the associated fault was caused by a post termination error. A post termination error occurs when an access is attempted after a transfer via the associated map entry has terminated. A value of zero indicates that the fault was not caused by a post termination error. A value of one indicates that the fault was caused by a post termination error.

The 1-bit partial block overflow subfield indicates whether or not the associated fault was caused by a partial block overflow. A partial block overflow occurs when there is one byte left to be transfered in a block and a double-byte request is made. A value of zero indicates that the fault was not caused by a partial block overflow. A value of one indicates that the fault was caused by a partial block overflow.

The 1-bit block overflow subfield indicates whether or not the associated fault was caused by a block overflow. A block overflow occurs when the block count is zero, the Peripheral Subsystem attempts an access, and the map entry state has not yet been updated. A value of zero indicates that the fault was not caused by a block overflow. A value of one indicates that the fault was caused by a block overflow.

Selected Index and Selected State Fields -

The selected index and selected state fields are filled in by the processor from information found in the process carrier data segment at process selection time, i.e when a "select process" IPC is received. The selected index is a process selection index used to communicate to Attached Processor software which process from the process selection list has just been bound to the processor. The selected index is obtained from the double byte quantity located at a displacement of eight double bytes into the process carrier data segment. The selected state is uninterpreted by processors and is obtained from the double byte quantity located at a displacement of nine double bytes into the process carrier data segment.

Processor Fault Information Area

The organization of the processor fault information area is described in Appendix C.

Local and Global Communication Segments

Both local and global communication segments used by IPs have the same format and interpretation as the corresponding objects employed by GDPs.

IPC Message Field

The IPC message field contains one of the following function request encodings. Message codes 0 through 7 represent IPC messages which are common between GDPs and IPs. Message codes 15, 16, and 17 are messages specific to Interface Processors. Message codes 8 through 14 are defined for GDPs but are unused by IPs.

0 - Select Process - Causes the processor to examine its carrier to determine if a process was received. If a process was received, the process is selected, the dispatching flag is set, and the selected state and selected index fields are copied from the process carrier data segment to the control window. The current process index field is invalidated when this IPC is received.

1 - Start Processor

2 - Stop Processor

3 - Set broadcast acceptance mode

4 - Clear broadcast acceptance mode

5 - Flush object table

6 - Suspend and fully requalify processor

7 - Suspend and requalify processor

8 - 14 - Unused

15 - Close (Invalidate) Windows and Unlock I/O Locks (on windows 0-3)

16 - Generate PS Reset

17 - Close (Invalidate) Windows and Unlock I/O Locks (on windows 0-3) and Enter Physical Mode

The base rights of a communication segment access descriptor are interpreted in same manner as for all segments of base type data segment. The system rights field of a communication segment is uninterpreted.

APPENDIX B
FUNCTION SUMMARY

Appendix B summarizes the Interface Processor functions. Three lists are provided to assist in locating the page which contains a particular function description.

One list, Table B-1, organizes the function set by alphabetical order. Table B-2 organizes the function set by increasing function code number and is particularly useful when debugging IP controller software. Table B-3 organizes the function set by operator id codes and is especially useful when debugging IP fault handling software.

The template for function descriptions is shown on page B-5. All function descriptions follow this style of presentation.

TABLE B-1

ALPHABETICAL INDEX TO IP FUNCTIONS

FUNCTION NAME	HEX FUNCTION CODE	DECIMAL OPERATOR ID	PAGE
(Logical Mode Functions)			
ALTER MAP AND SELECT DATA SEGMENT	00	3	B-6
AMPLIFY RIGHTS	08	11	B-8
BROADCAST TO PROCESSORS	18	27	B-9
CONDITIONAL RECEIVE	15	24	B-10
CONDITIONAL SEND	13	22	B-11
COPY ACCESS DESCRIPTOR	04	7	B-12
ENTER ACCESS SEGMENT	07	10	B-13
ENTER GLOBAL ACCESS SEGMENT	06	9	B-14
INDIVISIBLE ADD SHORT ORDINAL	19	28	B-15
INDIVISIBLE INSERT SHORT ORDINAL	1A	29	B-16
INSPECT ACCESS	0F	18	B-17
INSPECT ACCESS DESCRIPTOR	0E	17	B-18
LOCK OBJECT	10	19	B-19
NULL ACCESS DESCRIPTOR	05	8	B-20
READ PROCESSOR STATUS AND CLOCK	03	6	B-21
RECEIVE	14	23	B-22
RESTRICT RIGHTS	09	12	B-23
RETRIEVE PUBLIC TYPE REPRESENTATION	0B	14	B-24
RETRIEVE TYPE REPRESENTATION	0A	13	B-25
RETRIEVE REFINED OBJECT	0D	16	B-26
RETRIEVE TYPE DEFINITION	0C	15	B-27
SEND	12	21	B-28
SEND TO PROCESSOR	01	4	B-29
SET PERIPHERAL SUBSYSTEM MODE	02	5	B-31
SURROGATE RECEIVE	17	26	B-32
SURROGATE SEND	16	25	B-33
UNLOCK OBJECT	11	20	B-34
(Physical Mode Functions)			
ALTER MAP AND SELECT PHYSICAL SEGMENT	00	3	B-7
READ PROCESSOR STATUS AND CLOCK	03	6	B-21
SEND TO PROCESSOR	01	4	B-30
SET PERIPHERAL SUBSYSTEM MODE	02	5	B-31

TABLE B-2

IP FUNCTION SUMMARY BY FUNCTION CODE

HEX FUNCTION CODE	FUNCTION NAME	DECIMAL OPERATOR ID	PAGE
	(Logical Mode Functions)		
00	ALTER MAP AND SELECT DATA SEGMENT	3	B-6
01	SEND TO PROCESSOR	4	B-29
02	SET PERIPHERAL SUBSYSTEM MODE	5	B-31
03	READ PROCESSOR STATUS AND CLOCK	6	B-21
04	COPY ACCESS DESCRIPTOR	7	B-12
05	NULL ACCESS DESCRIPTOR	8	B-20
06	ENTER GLOBAL ACCESS SEGMENT	9	B-14
07	ENTER ACCESS SEGMENT	10	B-13
08	AMPLIFY RIGHTS	11	B-8
09	RESTRICT RIGHTS	12	B-23
0A	RETRIEVE TYPE REPRESENTATION	13	B-25
0B	RETRIEVE PUBLIC TYPE REPRESENTATION	14	B-24
0C	RETRIEVE TYPE DEFINITION	15	B-27
0D	RETRIEVE REFINED OBJECT	16	B-26
0E	INSPECT ACCESS DESCRIPTOR	17	B-18
0F	INSPECT ACCESS	18	B-17
10	LOCK OBJECT	19	B-19
11	UNLOCK OBJECT	20	B-34
12	SEND	21	B-28
13	CONDITIONAL SEND	22	B-11
14	RECEIVE	23	B-22
15	CONDITIONAL RECEIVE	24	B-10
16	SURROGATE SEND	25	B-33
17	SURROGATE RECEIVE	26	B-32
18	BROADCAST TO PROCESSORS	27	B-9
19	INDIVISIBLE ADD SHORT ORDINAL	28	B-15
1A	INDIVISIBLE INSERT SHORT ORDINAL	29	B-16
	(Physical Mode Functions)		
00	ALTER MAP AND SELECT PHYSICAL SEGMENT	3	B-7
01	SEND TO PROCESSOR	4	B-30
02	SET PERIPHERAL SUBSYSTEM MODE	5	B-31
03	READ PROCESSOR STATUS AND CLOCK	6	B-21

TABLE B-3

IP FUNCTION SUMMARY BY OPERATOR ID

DECIMAL OPERATOR ID	FUNCTION NAME	HEX FUNCTION CODE	PAGE
	(Logical Mode Functions)		
3	ALTER MAP AND SELECT DATA SEGMENT	00	B-6
4	SEND TO PROCESSOR	01	B-29
5	SET PERIPHERAL SUBSYSTEM MODE	02	B-31
6	READ PROCESSOR STATUS AND CLOCK	03	B-21
7	COPY ACCESS DESCRIPTOR	04	B-12
8	NULL ACCESS DESCRIPTOR	05	B-20
9	ENTER GLOBAL ACCESS SEGMENT	06	B-14
10	ENTER ACCESS SEGMENT	07	B-13
11	AMPLIFY RIGHTS	08	B-8
12	RESTRICT RIGHTS	09	B-23
13	RETRIEVE TYPE REPRESENTATION	0A	B-25
14	RETRIEVE PUBLIC TYPE REPRESENTATION	0B	B-24
15	RETRIEVE TYPE DEFINITION	0C	B-27
16	RETRIEVE REFINED OBJECT	0D	B-26
17	INSPECT ACCESS DESCRIPTOR	0E	B-18
18	INSPECT ACCESS	0F	B-17
19	LOCK OBJECT	10	B-19
20	UNLOCK OBJECT	11	B-34
21	SEND	12	B-28
23	RECEIVE	14	B-22
22	CONDITIONAL SEND	13	B-11
24	CONDITIONAL RECEIVE	15	B-10
25	SURROGATE SEND	16	B-33
26	SURROGATE RECEIVE	17	B-32
27	BROADCAST TO PROCESSORS	18	B-9
28	INDIVISIBLE ADD SHORT ORDINAL	19	B-15
29	INDIVISIBLE INSERT SHORT ORDINAL	1A	B-16
	(Physical Mode Functions)		
3	ALTER MAP AND SELECT PHYSICAL SEGMENT	00	B-7
4	SEND TO PROCESSOR	01	B-30
5	SET PERIPHERAL SUBSYSTEM MODE	02	B-31
6	READ PROCESSOR STATUS AND CLOCK	03	B-21

FUNCTION TEMPLATE
Operator ID: ID

Contents	Function Request Facility	Hex Byte Offset
results 0 through 9	reserved	20H-33H
operand 6	reserved	1EH
operand 5	reserved	1CH
operand 4	reserved	1AH
operand 3	reserved	18H
operand 2	reserved	16H
operand 1	reserved	14H
operand 0	reserved	12H
IP function code	0XXH (FUNCTION NAME)	10H
function state	reserved	0EH
process selection index	PROCESS INDEX	0CH

Note:

Required operands and available results are indicated by capital letters. Other areas are marked reserved.

The IP function code must be written into the function request facility last, i.e. only after all operands are provided. The function code occupies only location 10H. Byte location 11H is reserved.

The process selection index field is required on all IP function requests. This value (an access descriptor displacement) is used as an byte offset into the process selection list of the IP processor access segment. For example, the process selection index for process number 5 is 0000000000010100_B. Since it is not modified by function execution, it need not be rewritten if a new function is to be executed in the same process environment as the previous function.

The function state field, shown as reserved in all function summaries, may be examined after the IP receives an interrupt or it may be polled. The function state field should be set to zero before a function code is deposited. Interrupts for successful function completion may be selectively disabled.

ALTER MAP AND SELECT DATA SEGMENT

Operator ID: 3

Contents	Function Request Facility	Hex Byte Offset
results 0 through 9	reserved	20H-33H
operand 6	BLOCK COUNT	1EH
operand 5	BASE DISPLACEMENT	1CH
operand 4	SOURCE OBJECT SELECTOR	1AH
operand 3	MASK	18H
operand 2	BASE ADDRESS	16H
operand 1	ENTRY STATE	14H
operand 0	WINDOW INDEX	12H
IP function code	000H (ALTER MAP AND SELECT DATA SEGMENT)	10H
function state	reserved	0EH
process selection index	PROCESS INDEX	0CH

ALTER MAP AND SELECT DATA SEGMENT allows an operation to alter the inter-address space mapping provided by one of the address subrange map entries and to associate a given 432 or interconnect data segment with that address subrange map entry. The first operand is a double byte specifying which map entry/data segment, segment descriptor register is to be altered. This operator can only be used to affect map entries 0 through 3. The second operand is a double byte containing new entry state information. The third operand is a double byte containing the starting address of the new subrange to be mapped. The fourth operand is a double byte containing the mask used to specify size of the new subrange. The fifth operand specifies an access descriptor for the new data segment. This data segment access descriptor is copied into the mapped segment entry in the current context associated with the map entry being altered. The sixth operand is a double byte specifying the initial displacement into the data segment for the block transfer to start or pseudo-refinement. If the new entry state information specifies that this entry is being set up in block transfer mode, the seventh operand is a double byte containing a count of the number bytes to be transferred. Note that this operator is unique to 432 Interface Processors. If the new entry state information specifies that the window is to be closed (set "invalid") then only the first two operands are required.

ALTER MAP AND SELECT PHYSICAL SEGMENT
Operator ID: 3

Contents	Function Request Facility	Hex Byte Offset
results 0 through 9	reserved	20H-33H
operand 6	reserved	1EH
operand 5	PHYSICAL ADDRESS(high 8)	1CH
operand 4	PHYSICAL ADDRESS(low 16)	1AH
operand 3	MASK	18H
operand 2	BASE ADDRESS	16H
operand 1	ENTRY STATE	14H
operand 0	WINDOW INDEX	12H
IP function code	000H (ALTER MAP AND SELECT PHYSICAL SEGMENT)	10H
function state	reserved	0EH
process selection index	PROCESS INDEX	0CH

ALTER MAP AND SELECT PHYSICAL SEGMENT allows an operation to alter the inter-address space mapping provided by one of the address subrange map entries and to associate a given 432 or interconnect physical segment with that address subrange map entry. This physical mode operator is the equivalent of the logical mode operator ALTER MAP AND SELECT DATA SEGMENT. One difference is that the mapping facility area is not updated by this operator. Another difference is that map entry 4 can be updated by this operator. The first operand is a double byte specifying which map entry/data segment, segment descriptor register is to be altered. The second operand is a double byte containing new entry state information. The third operand is a double byte containing the starting address of the new subrange to be mapped. The fourth operand is a double byte containing the mask used to specify size of the new subrange. The fifth and sixth operands are a word (32 bits) containing the right-justified, 24-bit, physical base address of the segment in the 432 address space. If the new entry state information specifies that this entry is being set up in block transfer mode, the sixth operand is also used as a count of the number bytes to be transferred. Note that this operator is unique to 432 interface processors.

AMPLIFY RIGHTS
Operator ID: 11

Contents	Function Request Facility	Hex Byte Offset
results 0 through 9	reserved	20H-33H
operand 6	reserved	1EH
operand 5	reserved	1CH
operand 4	reserved	1AH
operand 3	reserved	18H
operand 2	reserved	16H
operand 1	DESC CTRL OBJ SELECTOR	14H
operand 0	DEST OBJECT SELECTOR	12H
IP function code	008H (AMPLIFY RIGHTS)	10H
function state	reserved	0EH
process selection index	PROCESS INDEX	0CH

AMPLIFY RIGHTS allows an operation to alter, under control of an protected descriptor control object, the set of rights and descriptor control information in the associated access descriptor. The first operand contains the object selector for an access descriptor for the given object. The second operand contains the object selector for a descriptor control object access descriptor. The resultant new access descriptor overwrites the original access descriptor for the given object. Thus, the destination access segment entry is the same as the source access segment entry.

BROADCAST TO PROCESSORS

Operator ID: 27

Contents	Function Request Facility	Hex Byte Offset
results 1 through 9	reserved	22H-33H
result 0	BOOLEAN	20H
operand 6	reserved	1EH
operand 5	reserved	1CH
operand 4	reserved	1AH
operand 3	reserved	18H
operand 2	reserved	16H
operand 1	DESTINATION PROCESSOR OBJECT SELECTOR	14H
operand 0	IPC MESSAGE	12H
IP function code	018H (BROADCAST TO PROCESSORS)	10H
function state	reserved	0EH
process selection index	PROCESS INDEX	0CH

BROADCAST TO PROCESSORS allows a process to broadcast an interprocessor message to all the processors in the system, including the processor it is executing on, via the interprocessor communication mechanism. The first operand contains the interprocessor message. The second operand contains the object selector for an access descriptor for the desired processor object. The boolean result, which is set to true if the control flags are deposited, is stored in the function result area.

CONDITIONAL RECEIVE
Operator ID: 24

Contents	Function Request Facility	Hex Byte Offset
results 1 through 9	reserved	22H-33H
result 0	BOOLEAN	20H
operand 6	reserved	1EH
operand 5	reserved	1CH
operand 4	reserved	1AH
operand 3	reserved	18H
operand 2	reserved	16H
operand 1	reserved	14H
operand 0	PORT OBJECT SELECTOR	12H
IP function code	015H (CONDITIONAL RECEIVE)	10H
function state	reserved	0EH
process selection index	PROCESS INDEX	0CH

CONDITIONAL RECEIVE allows a process to check for the availability of a message at a port and to indivisibly accept it if one is available. The first operand is used. The boolean result, which is set to true if a message is received, is stored in the function result area.

CONDITIONAL SEND
Operator ID: 22

Contents	Function Request Facility	Hex Byte Offset
results 1 through 9	reserved	22H-33H
result 0	BOOLEAN	20H
operand 6	reserved	1EH
operand 5	reserved	1CH
operand 4	reserved	1AH
operand 3	MESSAGE OBJECT SELECTOR	18H
operand 2	reserved	16H
operand 1	reserved	14H
operand 0	PORT OBJECT SELECTOR	12H
IP function code	013H (CONDITIONAL SEND)	10H
function state	reserved	0EH
process selection index	PROCESS INDEX	0CH

CONDITIONAL SEND allows a process to check for the availability of queue space at a port and to indivisibly deliver a message if space is available. The first and fourth operands are used. The boolean result, which is set to true if a message is deposited, is stored in the function result area.

COPY ACCESS DESCRIPTOR
Operator ID: 7

Contents	Function Request Facility	Hex Byte Offset
results 0 through 9	reserved	20H-33H
operand 6	reserved	1EH
operand 5	reserved	1CH
operand 4	reserved	1AH
operand 3	reserved	18H
operand 2	reserved	16H
operand 1	SOURCE OBJECT SELECTOR	14H
operand 0	DEST OBJECT SELECTOR	12H
IP function code	004H (COPY ACCESS DESCRIPTOR)	10H
function state	reserved	0EH
process selection index	PROCESS INDEX	0CH

COPY ACCESS DESCRIPTOR allows an operation to copy an access descriptor from a specified entry in any directly accessible access segment to a specified entry in any directly accessible access segment. The first operand contains the object selector for the destination access segment entry. The second operand contains the object selector for the access descriptor to be copied.

ENTER ACCESS SEGMENT
Operator ID: 10

Contents	Function Request Facility	Hex Byte Offset
results 0 through 9	reserved	20H-33H
operand 6	reserved	1EH
operand 5	reserved	1CH
operand 4	reserved	1AH
operand 3	reserved	18H
operand 2	reserved	16H
operand 1	SOURCE OBJECT SELECTOR	14H
operand 0	EAS INDEX	12H
IP function code	007H (ENTER ACCESS SEGMENT)	10H
function state	reserved	0EH
process selection index	PROCESS INDEX	0CH

ENTER ACCESS SEGMENT allows an operation to gain direct access to the access descriptors in a specified access segment. The first operand contains the index (range 1 - 3) for the destination access segment entry. The second operand contains the object selector for an access descriptor for the access segment to be entered.

ENTER GLOBAL ACCESS SEGMENT
Operator ID: 9

Contents	Function Request Facility	Hex Byte Offset
results 0 through 9	reserved	20H-33H
operand 6	reserved	1EH
operand 5	reserved	1CH
operand 4	reserved	1AH
operand 3	reserved	18H
operand 2	reserved	16H
operand 1	reserved	14H
operand 0	EAS INDEX	12H
IP function code	006H (ENTER GLOBAL ACCESS SEGMENT)	10H
function state	reserved	0EH
process selection index	PROCESS INDEX	0CH

ENTER GLOBAL ACCESS SEGMENT allows an operation to gain direct access to the access descriptors in the access segment provided implicitly via the currently associated process object. The operand contains the index (range 1 - 3) for the destination access segment entry.

INDIVISIBLE ADD SHORT ORDINAL

Operator ID: 28

Contents	Function Request Facility	Hex Byte Offset
results 1 through 9	reserved	22H-33H
result 0	ORIGINAL VALUE	20H
operand 6	reserved	1EH
operand 5	reserved	1CH
operand 4	reserved	1AH
operand 3	reserved	18H
operand 2	VALUE	16H
operand 1	DISPLACEMENT	14H
operand 0	SOURCE OBJECT SELECTOR	12H
IP function code	019H (INDIVISIBLE ADD SHORT ORDINAL)	10H
function state	reserved	0EH
process selection index	PROCESS INDEX	0CH

The result of adding the short-ordinal source value located by the first two operands (object selector and displacement) to the short-ordinal third operand indivisibly replaces the source value. The original source value is stored in the function result area.

A short-ordinal overflow fault cannot occur.

INDIVISIBLE INSERT SHORT ORDINAL
Operator ID: 29

Contents	Function Request Facility	Hex Byte Offset
results 1 through 9	reserved	22H-33H
result 0	ORIGINAL VALUE	20H
operand 6	reserved	1EH
operand 5	reserved	1CH
operand 4	reserved	1AH
operand 3	MASK	18H
operand 2	VALUE	16H
operand 1	DISPLACEMENT	14H
operand 0	SOURCE OBJECT SELECTOR	12H
IP function code	01AH (INDIVISIBLE INSERT SHORT ORDINAL)	10H
function state	reserved	0EH
process selection index	PROCESS INDEX	0CH

The short-ordinal fourth operand is used as a mask (as presented on the third operand and inverted on the source value). The result of ORing the short-ordinal source value located by the first two operands (object selector and displacement) to the short-ordinal third operand indivisibly replaces the source value. The original source value is stored in the function result area.

INSPECT ACCESS
Operator ID: 18

Contents	Function Request Facility	Hex Byte Offset
results 2 through 9	OBJECT DESCRIPTOR IMAGE	24H-33H
results 0 through 1	ACCESS DESCRIPTOR IMAGE	20H-23H
operand 6	reserved	1EH
operand 5	reserved	1CH
operand 4	reserved	1AH
operand 3	reserved	18H
operand 2	reserved	16H
operand 1	reserved	14H
operand 0	SOURCE OBJECT SELECTOR	12H
IP function code	00FH (INSPECT ACCESS)	10H
function state	reserved	0EH
process selection index	PROCESS INDEX	0CH

INSPECT ACCESS allows an operation to read the access information for the first level of any access path to which it holds an access descriptor. The first operand contains the object selector for an access descriptor for the level in the access path which is to be inspected. The ten double-byte result is stored in the function result area.

INSPECT ACCESS DESCRIPTOR
Operator ID: 17

		Hex Byte
Contents	Function Request Facility	Offset
results 2 through 9	reserved	24H-33H
results 0 through 1	SOURCE ACCESS DESCRIPTOR IMAGE	20H
operand 6	reserved	1EH
operand 5	reserved	1CH
operand 4	reserved	1AH
operand 3	reserved	18H
operand 2	reserved	16H
operand 1	reserved	14H
operand 0	SOURCE OBJ SELECTOR	12H
IP function code	00EH (INSPECT ACCESS DESCRIPTOR)	10H
function state	reserved	0EH
process selection index	PROCESS INDEX	0CH

INSPECT ACCESS DESCRIPTOR allows an operation to inspect an access descriptor to which it holds access. The first operand contains the object selector for an access descriptor which is to be inspected. The ordinal result is stored in the function result area.

LOCK OBJECT
Operator ID: 19

Contents	Function Request Facility	Hex Byte Offset
results 1 through 9	reserved	22H-33H
result 0	BOOLEAN	20H
operand 6	reserved	1EH
operand 5	reserved	1CH
operand 4	reserved	1AH
operand 3	reserved	18H
operand 2	reserved	16H
operand 1	DISPLACEMENT	14H
operand 0	OBJECT SELECTOR	12H
IP function code	010H (LOCK OBJECT)	10H
function state	reserved	0EH
process selection index	PROCESS INDEX	0CH

LOCK OBJECT allows an operation to lock an object lock located within a data segment. The first operand contains the object selector for a data segment access descriptor. The second operand contains the displacement within that data segment of the desired object lock. The boolean result, which is set to true if the object becomes locked, is stored in the function result area.

NULL ACCESS DESCRIPTOR
Operator ID: 8 .

Contents	Function Request Facility	Hex Byte Offset
results 0 through 9	reserved	20H-33H
operand 6	reserved	1EH
operand 5	reserved	1CH
operand 4	reserved	1AH
operand 3	reserved	18H
operand 2	reserved	16H
operand 1	reserved	14H
operand 0	DEST OBJECT SELECTOR	12H
IP function code	005H (NULL ACCESS DESCRIPTOR)	10H
function state	reserved	0EH
process selection index	PROCESS INDEX	0CH

NULL ACCESS DESCRIPTOR allows an operation to overwrite and thus logically clear a given access descriptor entry. At the same time, access to any object previously available via that access descriptor entry is given up. The operand contains the object selector for the destination access segment entry.

READ PROCESSOR STATUS AND CLOCK (Logical and Physical Mode)
Operator ID: 6

Contents	Function Request Facility	Hex Byte Offset
results 2 through 9	reserved	24H-33H
result 1	SYSTEM CLOCK	22H
result 0	PROCESSOR STATUS	20H
operand 6	reserved	1EH
operand 5	reserved	1CH
operand 4	reserved	1AH
operand 3	reserved	18H
operand 2	reserved	16H
operand 1	reserved	14H
operand 0	reserved	12H
IP function code	003H (READ PROCESSOR STATUS AND CLOCK)	10H
function state	reserved	0EH
process selection index	PROCESS INDEX	0CH

The 16-bit processor status field of the current processor is read from the processor object, right appended to the current value of the processor resident system clock, and stored in the function result area. The processor status field includes both processor unit number and processor status information.

READ PROCESSOR STATUS AND CLOCK is performed the same in both physical and logical modes.

RECEIVE
Operator ID: 23

Contents	Function Request Facility	Hex Byte Offset
results 0 through 9	reserved	20H-33H
operand 6	reserved	1EH
operand 5	reserved	1CH
operand 4	reserved	1AH
operand 3	reserved	18H
operand 2	reserved	16H
operand 1	reserved	14H
operand 0	PORT OBJECT SELECTOR	12H
IP function code	014H (RECEIVE)	10H
function state	reserved	0EH
process selection index	PROCESS INDEX	0CH

RECEIVE allows a process to receive a message at a specified port. The first operand is used.

RESTRICT RIGHTS
Operator ID: 12

Contents	Function Request Facility	Hex Byte Offset
results 0 through 9	reserved	20H-33H
operand 6	reserved	1EH
operand 5	reserved	1CH
operand 4	reserved	1AH
operand 3	reserved	18H
operand 2	reserved	16H
operand 1	DESC CTRL OBJ SELECTOR	14H
operand 0	DEST OBJECT SELECTOR	12H
IP function code	009H (RESTRICT RIGHTS)	10H
function state	reserved	0EH
process selection index	PROCESS INDEX	0CH

RESTRICT RIGHTS allows an operation to restrict its access to an object by altering, under control of an unprotected descriptor control object, the access descriptor for that object to have either restricted rights or restricted rights and restricted descriptor control. The first operand contains the object selector for an access descriptor for the given object. The second operand is an unprotected descriptor control object. The resultant new access descriptor overwrites the original access descriptor for the given object. Thus, the destination access segment entry is the same as the source access segment entry.

RETRIEVE PUBLIC TYPE REPRESENTATION
Operator ID: 14

Contents	Function Request Facility	Hex Byte Offset
results 0 through 9	reserved	20H-33H
operand 6	reserved	1EH
operand 5	reserved	1CH
operand 4	reserved	1AH
operand 3	reserved	18H
operand 2	TYPE DEF OBJ SELECTOR	16H
operand 1	SOURCE OBJ SELECTOR	14H
operand 0	DEST OBJECT SELECTOR	12H
IP function code	00BH (RETRIEVE PUBLIC TYPE REPRESENTATION)	10H
function state	reserved	0EH
process selection index	PROCESS INDEX	0CH

RETRIEVE PUBLIC TYPE REPRESENTATION allows an operation to retrieve the type representation for a public type. The first operand contains the object selector for the destination access segment entry. The second operand contains the object selector for an access descriptor for the type whose representation is to be retrieved.

RETRIEVE TYPE REPRESENTATION

Operator ID: 13

Contents	Function Request Facility	Hex Byte Offset
results 0 through 9	reserved	20H-33H
operand 6	reserved	1EH
operand 5	reserved	1CH
operand 4	reserved	1AH
operand 3	reserved	18H
operand 2	TYPE DEF OBJ SELECTOR	16H
operand 1	DESC CTRL OBJ SELECTOR	14H
operand 0	DEST OBJECT SELECTOR	12H
IP function code	00AH (RETRIEVE TYPE REPRESENTATION)	10H
function state	reserved	0EH
process selection index	PROCESS INDEX	0CH

RETRIEVE TYPE REPRESENTATION allows an operation to retrieve the type representation for any type for which it holds appropriate access to the associated type definition. The first operand contains the object selector for the destination access segment entry. The second operand contains the object selector for an access descriptor for the type whose representation is to be retrieved. The third operand contains the object selector for an access descriptor for the associated type definition.

RETRIEVE REFINED OBJECT
Operator ID: 16

Contents	Function Request Facility	Hex Byte Offset
results 0 through 9	reserved	20H-33H
operand 6	reserved	1EH
operand 5	reserved	1CH
operand 4	reserved	1AH
operand 3	reserved	18H
operand 2	REFIN CTRL OBJ SELECTOR	16H
operand 1	SOURCE OBJECT SELECTOR	14H
operand 0	DEST OBJECT SELECTOR	12H
IP function code	00DH (RETRIEVE REFINED OBJECT)	10H
function state	reserved	0EH
process selection index	PROCESS INDEX	0CH

RETRIEVE REFINED OBJECT allows an operation to retrieve access to the object to which it holds refined access. The first operand contains the object selector for the destination access segment entry. The second operand contains the object selector for an access descriptor for the refinement. The third operand contains the object selector for an refinement control object access descriptor.

RETRIEVE TYPE DEFINITION
Operator ID: 15

Contents	Function Request Facility	Hex Byte Offset
results 0 through 9	reserved	20H-33H
operand 6	reserved	1EH
operand 5	reserved	1CH
operand 4	reserved	1AH
operand 3	reserved	18H
operand 2	reserved	16H
operand 1	SOURCE OBJECT SELECTOR	14H
operand 0	DEST OBJECT SELECTOR	12H
IP function code	00CH (RETRIEVE TYPE DEFINITION)	10H
function state	reserved	0EH
process selection index	PROCESS INDEX	0CH

RETRIEVE TYPE DEFINITION allows an operation to retrieve an access descriptor for the type definition associated with a type. The first operand contains the object selector for the destination access segment entry. The second operand contains the object selector for an access descriptor for the type.

SEND
Operator ID: 21

Contents	Function Request Facility	Hex Byte Offset
results 0 through 9	reserved	20H-33H
operand 6	reserved	1EH
operand 5	reserved	1CH
operand 4	reserved	1AH
operand 3	MESSAGE OBJECT SELECTOR	18H
operand 2	reserved	16H
operand 1	reserved	14H
operand 0	PORT OBJECT SELECTOR	12H
IP function code	012H (SEND)	10H
function state	reserved	0EH
process selection index	PROCESS INDEX	0CH

SEND allows a process to send a specified message to a specified port. The first and fourth operands are used.

SEND TO PROCESSOR (Logical Mode)

Operator ID: 4

Contents	Function Request Facility	Hex Byte Offset
results 1 through 9	reserved	22H-33H
result 0	BOOLEAN	20H
operand 6	reserved	1EH
operand 5	reserved	1CH
operand 4	reserved	1AH
operand 3	reserved	18H
operand 2	reserved	16H
operand 1	DEST PROCESSOR OBJ SEL	14H
operand 0	IPC MESSAGE	12H
IP function code	001H(SEND TO PROCESSOR)	10H
function state	reserved	0EH
process selection index	PROCESS INDEX	0CH

SEND TO PROCESSOR allows a process to send an interprocessor message to one specific processor, including the processor it is executing on, via the interprocessor communication mechanism. The first operand contains the interprocessor message. The second operand contains the object selector for an access descriptor for the desired processor object. The boolean result, which is set to true if the control flags are deposited, is stored in the function result area.

SEND TO PROCESSOR (Physical Mode)

Operator ID: 4

Contents	Function Request Facility	Hex Byte Offset
results 1 through 9	reserved	22H-33H
result 0	BOOLEAN	20H
operand 6	reserved	1EH
operand 5	reserved	1CH
operand 4	reserved	1AH
operand 3	reserved	18H
operand 2	PHYSICAL ADDR (high 8)	16H
operand 1	PHYSICAL ADDR (low 16)	14H
operand 0	IPC MESSAGE	12H
IP function code	001H(SEND TO PROCESSOR)	10H
function state	reserved	0EH
process selection index	PROCESS INDEX	0CH

SEND TO PROCESSOR allows external processor software to send an interprocessor message to one specific processor, including the processor it is executing on, via the interprocessor communication mechanism. The first operand contains the interprocessor message. The second operand is a word (here shown as two consecutive double bytes) containing the right-justified, 24-bit, physical base address of the 432 memory segment which contains the image of the IP's processor object. The boolean result, which is set to true if the control flags are deposited, is stored in the function result area. This physical mode operator is the equivalent of the logical mode operator SEND TO PROCESSOR.

SET PERIPHERAL SUBSYSTEM MODE (Logical and Physical Mode)
Operator ID: 5

Contents	Function Request Facility	Hex Byte Offset
results 0 through 9	reserved	20H-33H
operand 6	reserved	1EH
operand 5	reserved	1CH
operand 4	reserved	1AH
operand 3	reserved	18H
operand 2	reserved	16H
operand 1	reserved	14H
operand 0	PS MODE	12H
IP function code	002H (SET PS MODE)	10H
function state	reserved	0EH
process selection index	PROCESS INDEX	0CH

SET PERIPHERAL SUBSYSTEM MODE allows an operation to change the mode settings for the connected peripheral subsystem, both on the processor and in the peripheral subsystem status field of the processor data segment. The operand contains a set of new peripheral subsystem mode flags. Note that this operator is unique to 432 Interface Processors.

SET PERIPHERAL SUBSYSTEM MODE when performed in physical mode is of the same form and provides the same function as SET PERIPHERAL SUBSYSTEM MODE performed in logical mode.

SURROGATE RECEIVE
Operator ID: 26

Contents	Function Request Facility	Hex Byte Offset
results 0 through 9	reserved	20H-33H
operand 6	reserved	1EH
operand 5	reserved	1CH
operand 4	reserved	1AH
operand 3	reserved	18H
operand 2	CARRIER OBJECT SELECTOR	16H
operand 1	DEST OBJECT SELECTOR	14H
operand 0	PORT OBJECT SELECTOR	12H
IP function code	017H(SURROGATE RECEIVE)	10H
function state	reserved	0EH
process selection index	PROCESS INDEX	0CH

SURROGATE RECEIVE allows a process to wait, via a surrogate carrier, at a port for a message from some process. The first three operands are used.

SURROGATE SEND
Operator ID: 25

Contents	Function Request Facility	Hex Byte Offset
results 0 through 9	reserved	20H-33H
operand 6	reserved	1EH
operand 5	reserved	1CH
operand 4	reserved	1AH
operand 3	MESSAGE OBJECT SELECTOR	18H
operand 2	CARRIER OBJECT SELECTOR	16H
operand 1	DEST OBJECT SELECTOR	14H
operand 0	PORT OBJECT SELECTOR	12H
IP function code	016H (SURROGATE SEND)	10H
function state	reserved	0EH
process selection index	PROCESS INDEX	0CH

SURROGATE SEND allows a process to send, via a surrogate carrier, a specified message to a specified port. All four operands are used.

UNLOCK OBJECT
Operator ID: 20

Contents	Function Request Facility	Hex Byte Offset
results 0 through 9	reserved	20H-33H
operand 6	reserved	1EH
operand 5	reserved	1CH
operand 4	reserved	1AH
operand 3	reserved	18H
operand 2	reserved	16H
operand 1	DISPLACEMENT	14H
operand 0	OBJECT SELECTOR	12H
IP function code	011H (UNLOCK OBJECT)	10H
function state	reserved	0EH
process selection index	PROCESS INDEX	0CH

UNLOCK OBJECT allows an operation to unlock an object lock located within a data segment. The first operand contains the object selector for a data segment access descriptor. The second operand contains the displacement within that data segment of the desired object lock.

APPENDIX C
FAULT SUMMARY

C-1. FAULT REPORTING

Both logical and physical mode faults are reported in fault information areas as described below. The fault information area for context, process, and processor level faults has the same organization. Process objects contain fault information for context and process level faults which occur in logical mode. Processor objects contain fault information for processor level faults which occur in logical mode. The process level fault information area in the process object is used when a process level fault occurs and a process is bound to the processor. The processor level fault information area in the processor object is used when a process level fault occurs and a process is not bound to the processor. Physical mode faults, which are all treated as context level faults, are reported in the processor fault information area.

C-2. FAULT INFORMATION AREAS

The fault information area is a 13 double-byte record organized as follows.

```
                  =                  = double byte
                  !                  ! displacement
                  !------------------!
                  ! execution state! n+12
                  !------------------!
                  !   operator id    !
                  !------------------!
                  !   system timer   !
                  !------------------!
                  !   psor status    !
                  !------------------!
                  ! cxt/prcs status!
                  !------------------!
                  !    PS status     !
                  !------------------!
fault             !    fault code    !
information       !------------------!
area              !  fault os/disp   !
                  !------------------!
                  !   pclk buffer    !
                  !------------------!
                  !    dir index     !
                  !------------------!
                  !    obj index     !
                  !------------------!
                  !      tempB       !
                  !------------------!
                  !      tempA       ! n
                  !------------------!
                  =                  =
```

The tempA, tempB, and pclk buffer fields contain the values of the corresponding on-chip registers at the time of the fault. If the fault is associated with object qualification, the directory index and object table index specify the object. The interpretation of the fault object selector/displacement vary depending on the fault.

The fault code, together with the operator id indicates the nature of the fault. The fault code field has the following format:

XRRXXXXX XXXXXXXX

RR	TYPE	Faults
10	MA	Memory Access Faults.
11	TS	Test Segment Type or Descriptor Type Faults.
0X	FF	All other faults.

The Peripheral Subsystem status, context/process status, processor status, and system timer fields contain the values of the the corresponding on-chip registers at the time of the fault. The operator id, which differs from the opcode field in an instruction, specifies the operator that causes the fault. If a fault occurs during instruction decoding, the operator id is zero. The operator id value of each operator is the same as the index found in Appendix B.

The execution state indicates the phase of execution when the fault occured. It is used to identify fault handling strategies in the more complex operators. A value of zero indicates the instruction can be re-executed with no rewind necessary. Non-zero execution state occurs in port and IPC operators only. The semantics of each execution state in the port operators is described in the 432 GDP Architecture Reference Manual. The organization of the execution state field is shown below.

```
-----------------------------
!    8 bits   !   8 bits    !
-----------------------------
       !             !
       !             !-------- execution state
       !---------------------- reserved
```

Memory Access Faults

The ZZ field specifies the type of memory access attempted The encoding
of the ZZ field is specified below.

	ZZ	Access Type
	--	----------
X10TTTTT	0XMWBBBB	Access Memory
X10TTTTT	10MWBBBB	Access Interconnect
X10TTTTT	11MWBBBB	Access Access Segment

The TTTTT field specifies the type of memory access fault. The encoding of the TTTTT field is specified below. Note that combinations of these encodings can occur.

XXXX1	AR	Access Rights Fault
XXX1X	SB	Segment Bounds Fault
XX1XX	MO	Memory Overflow Fault (physical address >= 2**24)
X1XXX	BE	Bus Error Fault
1XXXX	WR	Test Write Rights Fault

The M field specifies whether the fault was on a read-modify-write access. A value of zero indicates a normal access. A value of one indicates a read-modify-write access.

The W field specifies whether the fault was on a read or write access. A value of zero indicates a read access. A value of one indicates a write access.

The faulted displacement is recorded in the fault displacement (in access memory, or interconnect), and in the object index field of the fault object selector (in access access segment).

The BBBB field, which designates which segment was being accessed when the fault occurred, is defined as follows:

BBBB	Segment Name
0000	Context AS
0001	Entry AS 1
0010	Entry AS 2
0011	Entry AS 3
0100	Object Table Directory
0101	Object Table
0110	Processor AS
0111	Processor DS
1000	Context DS
1001	Process AS
1010	Process DS
1011	WorkA (Carrier DS)
1100	WorkB (Carrier AS)
1101	WorkC (Port DS)
1110	WorkD (Port AS)
1111	Mapping Facility

System Type Or Descriptor Type Faults

The fault code for system type or descriptor type faults is as follows:

D11XXXXX KKKKKKKK

The D field indicates which on-chip register was being tested. A value of zero indicates that the type information being tested was in TempA. A value of one indicates that the type information being tested was in TempB.

The KKKKKKKK field indicates the desired system type, or the desired object descriptor type, descriptor validity, base type, and storage associated bit fields.

All Other Faults

The fault code for all other faults is as follows:

X0XXXXXX XXTTEEEE

The TT and EEEE fields specify the fault level and the fault type. The TT bits are interpreted as follows:

TT	Description
00	Context Level Faults
01	Process Level Faults (group 1)
10	Process Level Faults (group 2)
11	Processor Level Faults

There are 16 fault types within each of the 4 groups. The encoding column of the tables in the following sections contains the TT and EEEE fields if the type is FF (all other faults).

C-3. OBJECT LEVEL OPERATOR FAULTS

Faults Common To All Operators Or Sub-operations

The following faults can occur anywhere during the execution of an operator or sub-operation (which includes instruction decoding, process dispatching, binding etc.). These faults are not explicitly referenced in the later sections.

FAULT GROUPS	TYPE	ENCODING
Memory Reference Faults =>		
Access Rights Fault	AR	
Segment Bound Fault	SB	
Memory Overflow Fault	MO	
Bus Error Fault	BE	
Write Rights Fault	WR	
Invalid Opcode Fault	FF	00 1100
Processor Stopped Fault	FF	00 1101
Object Table Cache Qualification Faults =>		
Object Descriptor Type Fault	TS	00010111
Object System Type Fault	TS	00000010
(Access) Segment Altered Faults =>		
=> Object Qualification Faults		

Sub-operations Faults

FAULT GROUPS	TYPE	ENCODING
Store Access Descriptor Faults =>		
Level Fault	FF	01 0100
Destination Delete Rights Fault	FF	01 0011
Object Qualification Faults =>		
Access Descriptor Validity Fault	FF	01 0000
Object Descriptor Fault	FF	01 0001
Object Descriptor Type Fault	TS	00010111
	TS	00011111
Memory Overflow Fault	FF	01 1011
Read/Write Rights Fault	FF	01 0110
Port Operation Faults =>		
=> Object Qualification Faults (Carrier AS)	TS	00001000
=> Object Qualification Faults (Carrier DS)	TS	00001000
=> Object Qualification Faults (Port AS)	TS	00000111
=> Object Qualification Faults (Port DS)	TS	00000111
Send Rights Fault	FF	01 1110
Carrier Lock Fault	FF	01 1001
Wakeup IPC Fault	FF	11 0100
Port Lock Fault	FF	01 1010
Carrier Queued Fault	FF	11 0110
Context Qualification Faults =>		
=> Object Qualification Faults (Context AS)	TS	00000100
=> Object Qualification Faults (Context DS)	TS	00000100
=> Entry Access Segment Qualification Faults (Entry 1, 2, and 3)		
Process Binding and Qualification Faults =>		
=> Object Qualification Faults (Process AS)	TS	00000101
=> Object Qualification Faults (Process DS)	TS	00000101
Process Level Objects Lock Fault	FF	11 0010
=> Context Qualification Faults		

Operator Faults

OPERATOR	TYPE	ENCODING
Alter Map and Select Data Segment		
Interconnect Descriptor Fault	FF	00 0100
I/O Lock Fault	FF	00 0101
Transfer Direction Fault	FF	00 0110
Length Validity Fault	FF	00 0111
Window Subrange Overlap Fault	FF	00 1000
Incomplete Block Transfer Fault	FF	00 1001
Operand Validity Fault	FF	00 1010
Forced Termination Fault	FF	00 1011
Copy Access Descriptor		
=> Store Access Descriptor Faults		
Null Access Descriptor		
Destination Delete Rights Fault	FF	01 0011
Amplify Rights		
Descriptor Control Object Rights Fault	FF	01 0110
=> Object Qualification Faults (Descriptor Ctl Obj)	TS	00001011
Destination Access Segment Rights Fault	TW	
Source Object Validity Fault	FF	01 0101
Type Fault	FF	01 1000
Race Condition Fault (the access descriptor was changed before the amplified value is stored back)	FF	01 1000
Restrict Rights		
no explicit fault cases		
Retrieve Public Type Representation		
Source Object Validity Fault	FF	01 0101
Object Descriptor Type Fault	TS	00010111
=> Store Access Descriptor Faults		
Retrieve Type Representation		
Type Definition Validity Fault	FF	01 0110
Source Object Validity Fault	FF	01 0101
Object Descriptor Type Fault	TS	00010111
Type Definition System Rights Fault	FF	01 0110
Type Fault	FF	01 1000
=> Store Access Descriptor Faults		
Retrieve Type Definition		
Source Object Validity Fault	FF	01 0101
Object Descriptor Type Fault	TS	00010111
=> Store Access Descriptor Faults		

Retrieve Refined Object		
Refinement Control Object System Rights Fault	FF	01 0110
=> Object Qualification Faults (Refinement Ctl Obj)	TS	00001100
Source Object Validity Fault	FF	01 0101
Type Fault	FF	01 1000
=> Store Access Descriptor Faults		
Inspect Access Descriptor		
no explicit fault cases		
Inspect Access		
Access Path Object Descriptor Type Faults	FF	01 0101
Lock Object		
=> Object Qualification Faults (data segment)		
Source Object Access Rights Fault	FF	01 0110
Unlock Object		
=> Object Qualification Faults (data segment)		
Source Object Access Rights Fault	FF	01 0110
Object Lock ID or Type Fault	FF	01 1001
Indivisibly Add Short Ordinal		
Indivisibly Insert Short Ordinal		
no explicit fault cases		
Enter Access Segment		
Enter Global Access Segment		
Entry Index Range Fault	FF	01 0101
Access Segment Read Rights Fault	FF	01 0110
=> Object Qualification Faults (access segment)		
Set PS Mode		
no explicit fault cases		
Send		
Receive		
Conditional Send		
Conditional Receive		
Port System Rights Fault	FF	01 0110
=> Port Operation Faults		
Surrogate Send		
Surrogate Receive		
Surrogate Carrier Validity and System Rights Fault	FF	01 0101
Port System Rights Fault	FF	01 0110
=> Port Operation Faults		
Send to Processor		

Broadcast to Processors		
Processor System Rights Fault	FF	01 0110
=> Object Qualification Faults (Processor AS)	TS	00000110
=> Object Qualification Faults (Commo Segment)	TS	00001010
Communication Segment Lock Fault	FF	01 1001
Read Processor Status and Clock		
no explicit fault cases		

C-4. NON-INSTRUCTION INTERFACE FAULTS

OPERATOR	TYPE	ENCODING
Initialization =>		
=> Object Qualification Faults (processor AS)	TS	00000110
=> Object Qualification Faults (object table directory)	TS	00000010
=> Object Qualification Faults (processor DS)	TS	00000110
Processor Object Lock Fault	FF	11 0001
=> IPC Faults		
Base/Mask Incompatibality Fault	FF	11 1000
IPC Faults =>		
=> Object Qualification Faults (Commo Segment)	TS	00001010
Communication Segment Lock Fault	FF	11 0011
Response Count Fault	FF	11 0010
Process Binding =>		
=> Object Qualification Faults (Carrier AS)	TS	00001000
=> Object Qualification Faults (Carrier DS)	TS	00001000
Process Object Lock Fault	FF	11 0001
=> Process Qualification Faults		
=> Port Operation Faults		
Process Selection =>		
=> Object Qualification Faults (Carrier AS)	TS	00001000
=> Object Qualification Faults (Carrier DS)	TS	00001000
=> Port Operation Faults		

Whenever the Interface Processor detects an event that may require attention from the IP controller, it records the nature of the event in the current IP processor data segment and emits a pulse on its interrupt line. There are several different types of events which may be sources of interrupts, and their occurrence and timing is not necessarily predictable. In this sense IP interrupts are similar to several I/O devices that are wire-ORd to a common interrupt line.

Thus, the IP controller must respond to an interrupt by "polling" the possible interrupt sources to determine which event has actually occurred. It may do this by examining fields of the IP processor data segment through the control window (window 4). The IP controller (and related hardware, such as latches and Intel 8259A interrupt controllers) must also accommodate the possibility that the IP may detect a second event at any time, including while the IP controller is handling a previous interrupt. The IP responds to all such events identically, noting the event in the IP processor data segment and emitting an interrupt pulse. Again, this is analagous to tying multiple independent I/O devices to one interrupt line.

The principal requirement of IP interrupt handling hardware and software, then, is to field interrupt requests that may be closely-spaced, and to respond individually to the different types of events that an interrupt may signal.

Figure D-1 shows one approach to the overall design of an IP interrupt handler. This strategy assumes that hardware latches the IP's interrupt request pulse. As soon as it is invoked, the interrupt handler masks further IP interrupt requests and resets the hardware latch. This insures that a second request is unlikely to be missed, and prevents the interrupt handler from being reentered. Then the environment of the interrupted routine is saved and higher-priority interrupts are enabled, so that the interrupt handler itself can be interrupted if necessary.

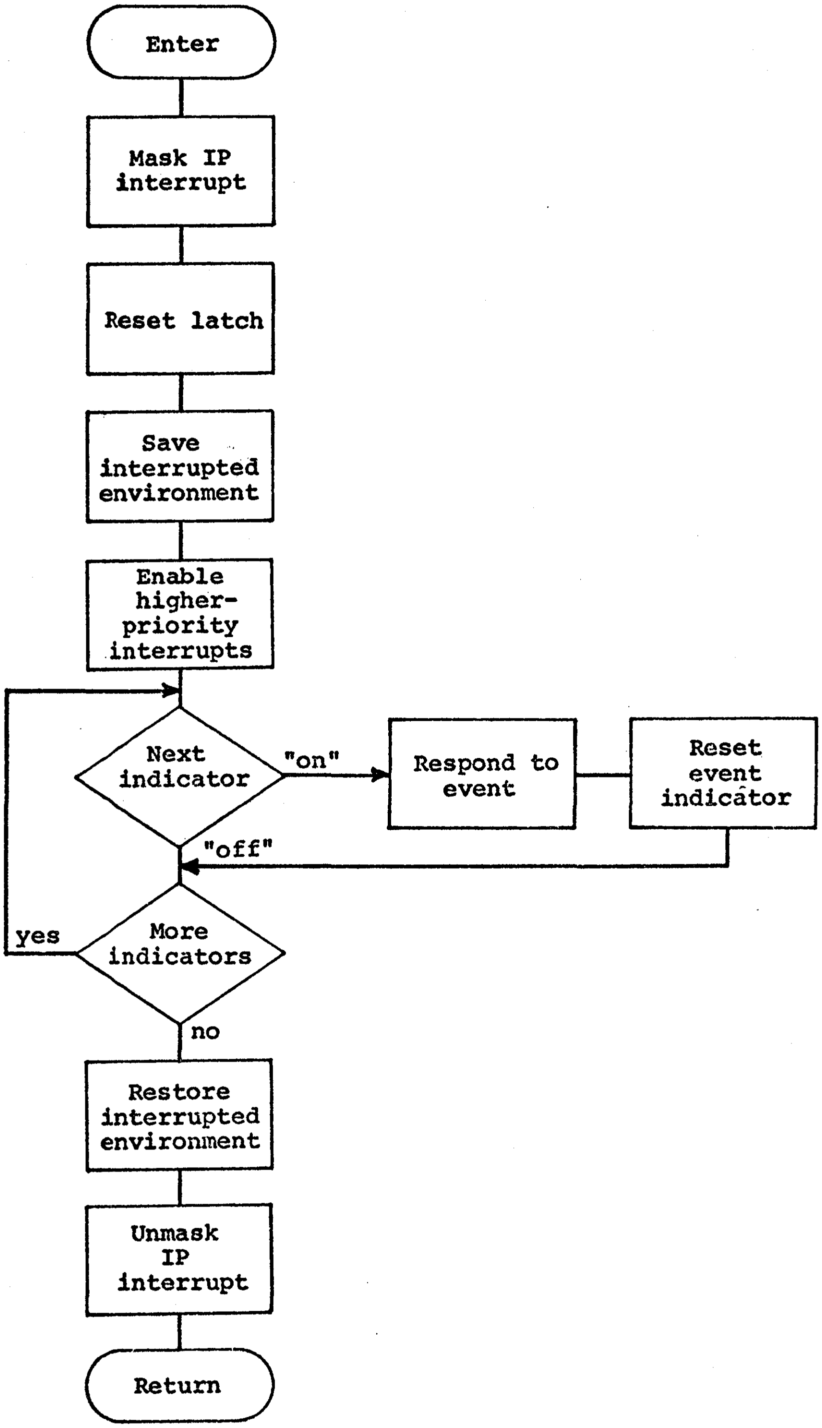


Figure D-1 Interrupt Handler

The central logic of this approach assumes that there is a "list" of possible interrupt sources to be scanned, and that passing through this list may uncover one (the usual case), multiple, or zero events that require responses. To illustrate the second two cases, assume that the possible events are labelled A through K, and that the interrupt handler tests for A, then B, and so on. Assume also that event B occurs followed quickly by event J. The interrupt handler is invoked for event B, shortly thereafter the IP updates J's indicator and emits a second interrupt pulse, which is latched. The handler scans its list of event indicators, finds that both B and J have occurred and responds to them both. Reaching the end of the list, the interrupt handler enables the IP interrupt and returns. Immediately, J's latched interrupt request is recognized and the handler is invoked again. This time, however, it will find no events indicated in the IP processor data segment, since it responded to both B and J in the previous invocation. It will simply clear the interrupt latch, pass through the list, unmask the IP interrupt, and return, effectively making a null response.

Table D-1 lists the IP processor data segment subfields that the IP interrupt handler must examine to determine the source of an interrupt. Note that as soon as the handler recognizes that an event indicator is "on", it should turn it "off" by indivisibly zeroing the field using the INDIVISIBLE INSERT SHORT ORDINAL function. This is necessary to prevent the interrupt handler from being misled in its next invocation.

Table D-1 Interrupt Sources

Processor Data Segment Subfield	Value	Event
Function state field		
		Function completion state subfield
	0000_B	Function completed normally (this interrupt may be masked)
		Fault level subfield
	01_B	Context-level fault
	10_B	Process-level fault
	11_B	Processor-level fault
Entry state field (One per map entry)		Transfer state subfield
	01_B	Transfer terminated by byte count[1]
	10_B	Transfer termination forced[1,3]
	11_B	Transfer terminated by fault[2]
Local IPC response	1_B	IP has responded to local IPC
Global IPC response	1_B	IP has responded to global IPC
Alarm response	1_B	IP has responded to a alarm request
Reconfiguration response	1_B	IP has responded to a reconfiguration request
Dispatching response	1_B	IP has received a "select process" IPC

Notes:

(1) Applies to window 0, buffered mode only.
(2) Separate indications are provided for each transfer window.
(3) Only done via the ALTER MAP AND SELECT DATA SEGMENT function.

APPENDIX E
SYSTEM INITIALIZATION

System initialization may be considered as a sequence of activities that brings a 432-based system from an arbitrary state to a known state where execution can begin. Although the initialization sequence will vary widely among applications, this appendix outlines the basic procedure. The first section describes how the system may be reset to a known state. The second section shows how an Interface Processor running in physical reference mode may be used to initialize memory and interconnect components thereby establishing an environment in which execution can take place. The final section discusses system startup, the procedure for commencing execution.

E-1. SYSTEM RESET

Most systems include a reset switch that is used to initialize the system after power-up and to restart the running system if necessary. In a 432 system, the INIT pins of all IPs (see iAPX 43203 VLSI Interface Processor Data Sheet, Order No. 171874, for details)and GDPs, and the RESET (or equivalent) pins of all Peripheral Subsystem components must be activated when a full system reset is performed. However, system designers may also decide to provide the option to selectively initialize elements of a 432 system.

Although this is subject to variation, a typical Attached Processor responds to a reset pulse by aborting any current operation, disabling interrupts and then vectoring execution to the code located at some predefined address (typically in non-volatile memory). The code will normally initialize I/O devices and enable interrupts, at which point normal execution begins. The 432 makes no special demands of the Peripheral Subsystem except that it should be prepared to handle an interrupt request from the IP shortly after system reset.

An Interface Processor responds to an INIT pulse by aborting any current operation, entering physical reference mode, configuring its windows as shown in table E-1, clearing broadcast acceptance mode, and then issuing an interrupt request to its Attached Processor. The interrupt request signals the IP controller that the Interface Processor has initialized itself and will accept subrange address references, including physical reference mode function requests written through window 4. Any attempt by the IP controller (or any active agent in the Peripheral Subsystem) to reference a subrange prior to receiving the IP's interrupt request produces an undefined result. An IP switches from physical to logical reference mode upon receipt of the startup IPC as defined below.

A General Data Processor responds to an INIT pulse by aborting any current activity and then waiting in a quiescent state for the startup IPC. The startup IPC is defined as the first local IPC received following an INIT pulse; a GDP will ignore any intervening global IPC.

To summarize, shortly after system reset, Attached Processors (and Peripheral Subsystems) will be able to run as desired, IPs will be able to run in physical reference mode, and GDPs will be waiting for a signal to begin execution.

E-2. ESTABLISHING AN EXECUTION ENVRIONMENT

Prior to starting any GDP (or switching any IP to logical reference mode) an environment in which the processor can execute must be created in 432 memory. This environment consists of a set of interrelated system objects; a minimal environment, sufficient to start one process running on a GDP, could be characterized as follows:

- o the initial object table directory (loaded at physical address 8);
- o an object table;
- o a processor object;
- o a dispatching port;
- o a process object (queued at the dispatching port).

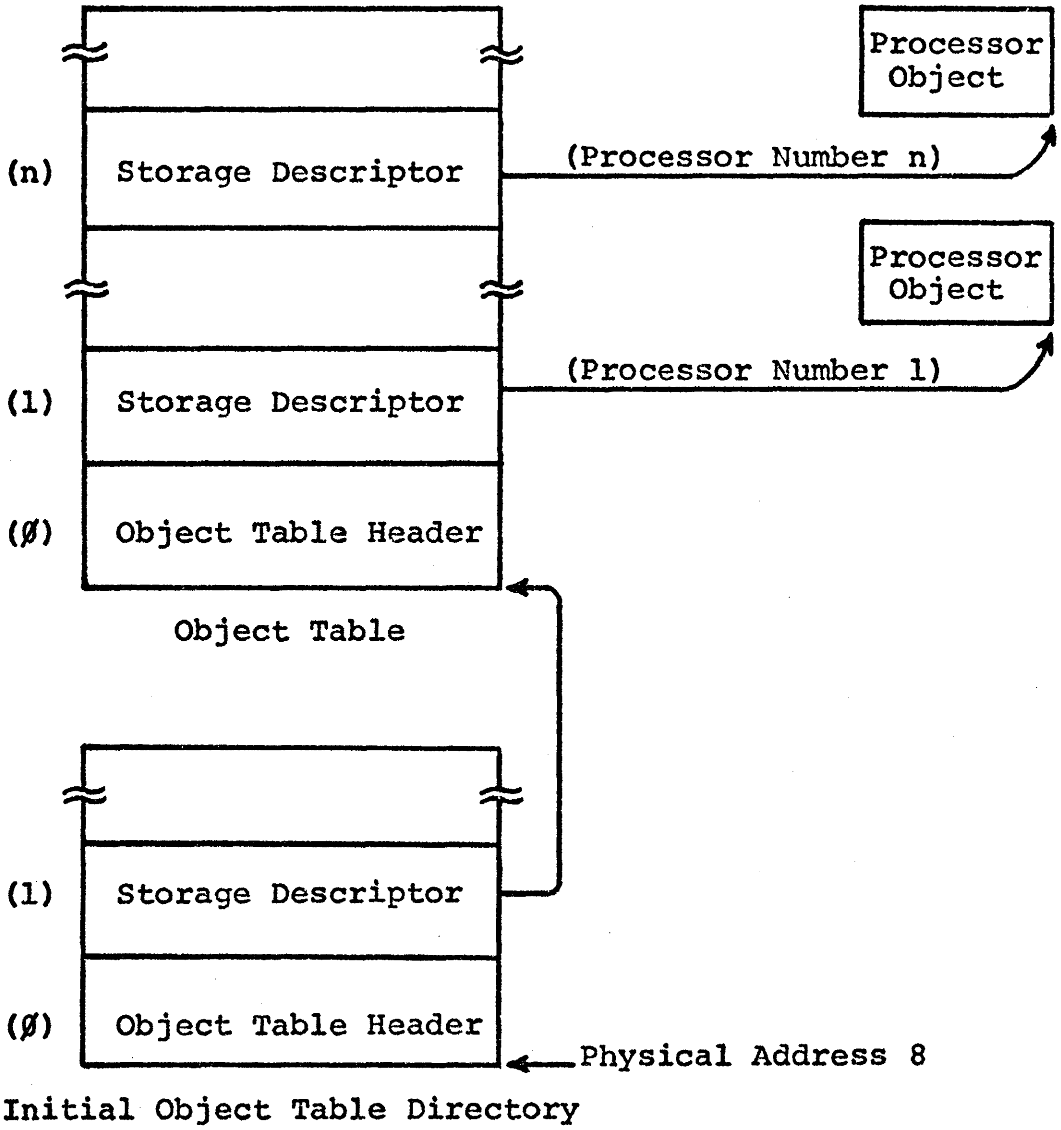


Figure E-1 Processor Object Location

Note that the term "processor object" above is meant to include communication segments, and a processor carrier, in addition to processor access and data segments. Likewise, "process object" includes a domain, instruction segments, context objects, etc. This environment may be extended to include more processors, processes, ports and so on, as is appropriate for a given application.

The initial execution environment may not pre-exist in 432 non-volatile memory, since the processors routinely update the objects during execution. Therefore, the initial environment must be loaded from a Peripheral Subsystem (where it may, in fact, reside in non-volatile storage). One Peripheral Subsystem will typically be designated to load the initial environment in physical reference mode; in this discussion this Peripheral Subsystem is referred to as the initializing AP.

At no time during system initialization should more than one Peripheral Subsystem be updating 432 system memory. In most applications, the remaining Peripheral Subsystems will refrain from accessing the 432 system until their IPs have switched to logical reference mode. It is possible, however, for a second Peripheral Subsystem to read 432 system memory while still in physical reference mode; some applications may wish to designate a second Peripheral Subsystem to monitor the activity of the initializing AP in this way.

Some systems will need to perform a number of preliminary activities before the initial environment can be loaded. These activities, which will be defined by each application, may include:

- ascertaining the system configuration (i.e., the number and type of processors present, and the amount of memory available);
- verifying that system components are operational;
- initializing registers located in the interconnect space (e.g., address range or error count registers in memory controllers);
- initializing error checking and correcting (ECC) memory.

Windows 0 and 1 may be useful in connection with these preliminary activities. Window 1 could be used to read system configuration information encoded in predefined registers of the interconnect address space, for example. Window 1 may also be used to initialize registers in memory controllers, provided these registers are located in the first 32K bytes of the interconnect address space.

Before any function request is made by the IP, enough 432 memory must be initialized to allow IP execution. This is necessary because the IP will attempt to update the segment mapped by window 4 in response to the function request. Once this path to memory has been established, window 1 can be opened onto another 32K byte segment by the ALTER MAP AND SELECT PHYSICAL SEGMENT function if additional interconnect components need to be referenced; this should normally be necessary only in very large systems.

If a system employs error checking and correcting memory (ECC) that does not initialize itself, the initializing AP can initialize it if the memory is organized in units eight or fewer bytes wide. Window 0 comes up in block mode set for a 64K byte transfer starting at physical address 0. Any data written through this window (e.g. all zero bits) is written by the IP in eight-byte blocks. The window can be moved through the entire memory space in 64K byte segments.

Once the system configuration has been established, the interconnect path set up and memory initialized, the initializing AP can load the initial execution environment. The simplest and fastest way to do this is to write all the required binary images through window 0. An alternative is to load the minimal object set required to support one IP in logical reference mode, and possibly one GDP. The rest of the environment (other processes, etc.) can then be loaded in logical reference mode by the initializing AP working alone, or under the direction of a GDP process. This approach has the advantage of getting the system into logical reference mode as soon as possible, where operations are inherently more protected than in physical reference mode.

E-3. SYSTEM STARTUP

Each processor in the system must be started independently by sending it a startup IPC (the first local IPC after INIT). At least one 432 processor, perhaps its own IP, must be started by the initializing AP using the SEND TO PROCESSOR function (physical mode). The remaining processors must be started one at a time, and this can be done by the initializing AP, or by a processor already started by it. Note that the initializing AP (as well as all IPs) remains in physical reference mode until it receives a startup IPC.

GDPs and IPs respond to the startup IPC identically except that the IP additionally switches to logical reference mode. The basic response is to first qualify its execution environment and then to interpret the IPC and respond to it normally. The processor qualifies its execution environment by first reading a unique processor ID contained in the low order byte of interconnect register 0.

Having established its identity, the processor proceeds to locate its processor object. It does this by assuming that the initial object table directory is located at physical memory address 8 (see figure E-1). A segment header field of eight bytes precedes the initial object table directory. It further assumes that the first storage descriptor in the directory locates an object table containing storage descriptors for processor objects. Using its processor ID as an index, the processor selects the storage descriptor from the object table which locates its processor object. After qualifying its processor object, the IP is able to find its local communcation segment, where it examines the IPC message field. Now in logical reference mode, the IP can respond to the IPC message and perform all normal operations.

As usual, an IP will generate an interrupt after it responds to the IPC message. This second interrupt following reset indicates to the IP controller software that the IP is in logical reference mode and that normal execution may begin. Note that window 4 will then be configured as defined by the attributes encoded in the IP's processor object. Since window 4 provides the data path to the function request facility, the other windows may be configured immediately by means of the ALTER MAP AND SELECT DATA SEGMENT function.

Table E-1 Window Configuration Following INIT

Attribute	Window 0	Window 1	Window 4
Window Status	Open	Open	Open
Transfer Mode	Block	Interconnect	Random
Subrange Base Address	$07E00_H$	08000_H	$07F00_H$
Subrange Size	00100_H	08000_H	00100_H
Segment Base	0	0	0
Segment Length	65,535	65,535	65,535
Direction	Write	Read/Write	Read/Write
Transfer State	In Progress	In Progress	In Progress
Overlay	Yes	Yes	Yes

APPENDIX F
INTERPROCESS COMMUNICATION AND DISPATCHING EXAMPLE

In Chapter 1, a printer example was used to demonstrate the flow of data between 432 processes and AP tasks. In this appendix, the printer example is again discussed. However, this time the view taken is that of a programmer writing an Attached Processor task to direct an IP to accomplish printer output. The program contained in this appendix is written in a PL/M-86-like dialect typical of the development environment which will be at the disposal of the AP program developer. This program is included to clarify an earlier example and is not suggested as a scheme for actual implementation.

The program example which follows assumes that a set of 432 system objects preexists in 432 memory. These objects are illustrated in Figure F-1. This system contains:

- IP processor object;
- a print request port to which a 432 process (GDP or IP) can send print requests;
- a print reply port to which an IP process can return the status of the print action;
- an IP dispatching port where IP processes await service.
- several IP processes are shown, though only one is required for the purposes of the example;
- one print object, a simple data segment, which carries printer data and is reused when returning printer status.

There are four main sections to this program:

- Variable declarations;
- Utility procedures;
- Initialization;
- Print driver body.

In the variable declarations section, notice that the control window, window 4, is declared as a structure whose components are defined from the definition in Appendix A. This program assumes that window 4, the control window, is opened onto the function request facility in the IP's processor object. It also assumes that all initialization has been performed and that the IP is operating in logical reference mode.

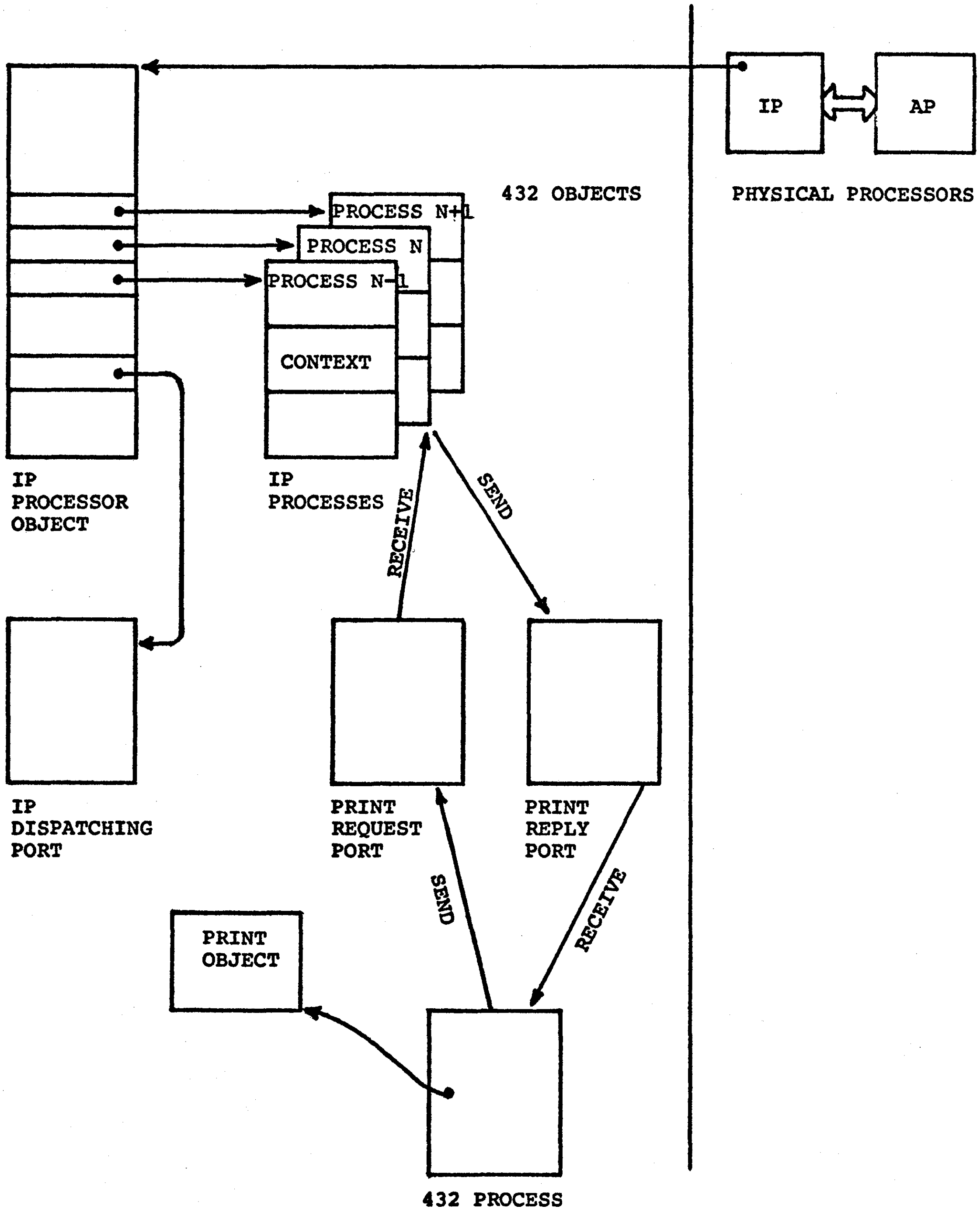


Figure F-1 Print Example Objects

Procedures in the utilities section demonstrate how a programmer can construct facilities to invoke IP functions. Recall from the function summary in Appendix B that an AP requests an IP function by writing a process selection index, all required operands, and finally depositing a function code into the appropriate slots in the function request facility (frf). The IP begins execution of the function only after the function code has been written. This is demonstrated by the procedures Open_window and Close_window.

The initialization section of the program points out some simplifying assumptions which are made for the purpose of this example. First, interrupts are disabled. This converts the three tasks of the printer example (printer server task, printer task, and printer reply task) of Chapter 1 into sequential tasks rather than concurrent tasks. It also makes it easier to demonstrate changes in the state of the system and illustrate them with the accompanying figures. Second, the call on the Dispatch procedure assumes that only one IP process exists in the 432 system. The IP supports multiple process environments but only one is required in this example.

The print driver body contains an aggregation of code which accomplishes the three tasks of the Chapter 1 example. Notice that the three tasks are performed sequentially.

Imbedded in the program text are references to Figures 2 through 6 which depict the state of the 432 system objects and the logical I/O processor (the IP/AP pair).

```
Printer_task:
  Procedure;


             /**********************************************/
             /*                                            */
             /*        Data Structures and Constants       */
             /*                                            */
             /**********************************************/


    /**********************************************************************************/
    /* Declare the 256 byte structure for the Control Window and map it beginning at  */
    /* an offset of 07F00H into the 64K byte segment which is reserved for the IP.    */
    /* For the purposes of this example, the base of the IP's reserved area is at location*/
    /* 080000H of the Attached Processor memory space.                                */
    /**********************************************************************************/
    Declare IP_base literally '080000H';
    Declare Window_4 structure (
             ps_state                        word,
             ipc_state                       word,
             alarm_state                     word,
             disp_state                      word,
             reserved_1                      word,
             frf_prcs_idx                    word,
             frf_function_state              word,
             frf_operator (7)                word,
             frf_result (10)                 word,
             ipc_fun_req                     word,
             reserved_2                      word,
             mf_block_count                  word,
             mf_432_disp                     word,
             mf_ps_disp                      word,
             reserved_3                      word,
             mf_window_info (5) structure (
               entry_state                   word,
               mask                          word,
               base_disp                     word),
            mf_fault_information (14)        byte,
            selected_idx                     word,
            selected_state                   word,
            psor_fault_information (13)      byte,
            reserved_4 (2)                   word) at (IP_base + 07F00H);

    Declare subrange (1024) byte at (IP_base + 4096);
                                             /* byte array comprising windowed subrange */
    Declare offset          word;            /* offset into subrange                    */

    Declare true            literally '0001H'; /* Logical value true                    */
    Declare false           literally '0000H'  /* Logical value false                   */
```

```
/*************************************************************************************/
/* Seven object selectors are required. One for the message slot in the Context      */
/* Access Segment, since this is where the hardware will put the Access              */
/* Descriptor (AD) for the Print Request Message following the Receive instruction.  */
/*                                                                                   */
/* One for the Print Request Port and one for the Print Reply Port. We assume        */
/* that at system initialization ADs for these ports were stored in slots nine       */
/* and ten of the Context Access Segment in Process Object 1.                        */
/*                                                                                   */
/* One for the IP Dispatching Port, one for the IP Processor Carrier data segment,   */
/* one for the IP Processor Carrier access segment, and for a null access descriptor. */
/* These are required so that blocking Receives and blocking sends can be handled.   */
/* We assume ADs for these objects are stored in slots eleven, twelve, and thirteen, */
/* respectively of the Context Access Segment in Process Object 1 at initialization. */
/*************************************************************************************/

Declare message_obj_sel                   literally '001100B';
Declare request_port_obj_sel              literally '100100B';
Declare reply_port_obj_sel                literally '101000B';
Declare dispatching_port_obj_sel          literally '101100B';
Declare psor_carrier_as_obj_sel           literally '110000B';
Declare psor_carrier_ds_obj_sel           literally '110100B';
Declare null_destination_obj_sel          literally '111000B';

/*************************************************************************************/
/* The process_selection_index for process number 1.  Note that this number is a byte */
/* index into the process selection list in the IP processor access segment.         */
/*************************************************************************************/
Declare process_1                 literally '0000000000000100B';


         /*************************************************/
         /*                                               */
         /*              Utility Procedures               */
         /*                                               */
         /*************************************************/


Await_function_completion:
  Procedure;

    /*********************************************************************************/
    /* This procedure busy waits for the previous function request to complete. It   */
    /* Spins waiting for the function completion field of the function  state to     */
    /* equal zero.                                                                   */
    /*********************************************************************************/

    Do While (Window_4.frf_function_state and 000FH) <> 0; End;
  End
Await_function_completion;
```

```
Dispatch:
  Procedure;

    /*********************************************************************************/
    /* This procedure hangs the IP's processor carrier on the IP's dispatching       */
    /* port. This allows blocking sends and receives to be handled.                  */
    /* This example assumes that the IP processor carrier blocks at the dispacthing  */
    /* port.  No "select process" IPC is received if the Surrogate Receive does not  */
    /* block.                                                                        */
    /*********************************************************************************/

    Window_4.disp_state  = false;

      /* Unlock the IP's processor carrier.                                          */
    Window_4.frf_prcs.idx  = process_1;                  /* Use process object 1.    */
    Window_4.frf_operand(0)  = psor_carrier_ds_obj_sel;  /* Data segment             */
    Window_4.frf_operator  = 011H;                       /* Unlock function code.    */

    Call Await_function_completion;

      /* Hang processor carrier on the dispatching port.                             */
    Window_4.frf_prcs_idx  = process_1;                  /* Use process object 1.    */
    Window_4.frf_operand(0)  = dispatching_port_obj_sel; /* port                     */
    Window_4.frf_operand(2)  = null_destination_obj_sel; /* destination              */
    Window_4.frf_operand(3)  = psor_carrier_as_obj_sel;  /* carrier                  */
    Window_4.frf_operator  = 017H;                       /* Surrogate receive        */
                                                         /* function code.           */
    Call Await_function_completion;
  End
Dispatch;


Open_window:
  Procedure;

  /***********************************************************************************/
  /* Open a window to the message, Figure F-5                                        */
  /***********************************************************************************/

    Window_4.frf_prcs_idx    = process_1;                /* process object index     */
    Window_4.frf_operand(0)  = 3;                        /* window index             */
    Window_4.frf_operand(1)  = 0000101B;                 /* entry state              */
    Window_4.frf_operand(2)  = 4096;                     /* base address             */
    Window_4.frf_operand(3)  = 1111110000000000B;        /* mask                     */
    Window_4.frf_operand(4)  = message_obj_sel;          /* data segment             */
    Window_4.frf_operand(5)  = 0;                        /* base displacement        */
    Window_4.frf_operator    = 000H;                     /* Alter Map and Select Data */

    Call Await_function_completion;
  End                                                    /* Segment function code    */
Open_window;
```

```
Get_print_message:
  Procedure;

    /**********************************************************************************/
    /* Attempt to Receive a message from the Print Request Port, Figure F-2          */
    /**********************************************************************************/

    Window_4.frf_prcs_idx  = process_1;                   /* Use process object 1.    */
    Window_4.frf_operand(0)  = request_port_obj_sel;      /* port                     */
    Window_4.frf_operator  = 014H;                        /* Receive function code.   */

    Call Await_function_completion;

    If (Window_4.frf_function_state and 0020H) <> 0 Then
      Do
        /******************************************************************************/
        /* Receive instruction blocked, no outstanding print requests                 */
        /* Busy wait until a GDP process sends a print request to the print           */
        /* request port. See Figure F-3 for the SEND unblocking the blocked RECEIVE   */
        /* Such an event will trigger an interrupt in the AP                          */
        /* (which we have disabled) and set Window_4.disp_state true                  */
        /* indicating the nature of the interrupt.                                    */
        /* See Figure F-4 for details on the wakeup IPC and subsequent interrupt.     */
        /******************************************************************************/

        Do While not Window_4.disp_state; End;

        /******************************************************************************/
        /* At this point Window_4.selected_index contains the index of the            */
        /* process object which was dispatched. Since we are using only process       */
        /* object one selected_index will equal one. Window_4.selected_state          */
        /* contains software defined information concerning the action taken,         */
        /* if any, by software in completing this instruction.                        */
        /******************************************************************************/

        Call Dispatch;     /* Hang IP processor carrier on dispatching port.          */
      End;
    End;
  End
Get_print_message;
Close_window;
```

```
Close_window:
  Procedure;

    /*********************************************************************************/
    /* Close window, note only two operands are required.                          */
    /*********************************************************************************/

    Window_4.frf_prcs_idx      = process_1;              /* process object index      */
    Window_4.frf_operand(0)    = 3;                      /* window index              */
    Window_4.frf_operand(1)    = 0000100B;               /* entry state               */
    Window_4.frf_operator      = 000H;                   /* Alter Map and Select Data */
                                                         /* Segment function code     */
    Call Await_function_completion;
  End
Close_window;
```

```
Return_print_message:
  Procedure;

    /*********************************************************************************/
    /* Send message to Print Reply Port.  See Figure F-6                             */
    /*********************************************************************************/

    Window_4.frf_prcs_idx    = process_1;               /* process object index      */
    Window_4.frf.operand(0)  = reply_port_obj_sel;      /* port                      */
    Window_4.frf_operand(1)  = message_obj_sel;         /* message                   */
    Window_4.frf_operator    = 016H;                    /* Send function code        */

    Call Await_function_completion;

    If (Window_4.frf_function_state and 0010H) <> 0 Then
      Do
        /*****************************************************************************/
        /* Send instruction blocked, wait for a GDP process to receive a             */
        /* message from the Print Reply Port. Busy wait for a GDP process            */
        /* to receives a message from the Print Reply Port. Such an event            */
        /* will trigger an AP interrupt  and set Window_4.disp_state true            */
        /* to indicate the nature of the interrupt.                                  */
        /*****************************************************************************/
        Do While not (Window_4.disp_state = 1); End;

        /*****************************************************************************/
        /* At this point Window_4.selected_index contains the index of the           */
        /* process object which was dispatched. Since we are using only process      */
        /* object one selected_index will equal one. Window_4.selected_state         */
        /* contains software defined information concerning the action taken, if     */
        /* any, by software in completing this instruction.                          */
        /*****************************************************************************/

        Call Dispatch;    /* Hang IP processor carrier on dispatching port.          */
      End;
    End;
  End
Return_print_message;
```

```
             /************************************************/
             /*                                              */
             /*               Initialization                 */
             /*                                              */
             /************************************************/

    Call Disable_Interrupts; /* Busy waiting will be used, not the interrupt mechanism      */
                             /* Also assume that no faults will occur                       */

    Call Dispatch;

             /************************************************/
             /*                                              */
             /*            Print Driver Body                 */
             /*                                              */
             /************************************************/

    Do While true;                      /* loop forever                                     */

      Call Get_print_message;           /* Receive a message from the Print Request Port.   */

      Call Open_window;                 /* Open a window onto the message.                  */

      Do offset  = 0 to 1023;           /* Read and print the contents of the message       */
       Call Print (subrange(offset));   /* using the mapped subrange and the AP's native    */
      End;                              /* instruction. Assume Print is a system routine.   */

      Call Close_window;                /* Close the window.                                */

      Call Return_print_message;        /* Send the message to the Print Reply Port.        */
    End;
  End
Printer_task;
```

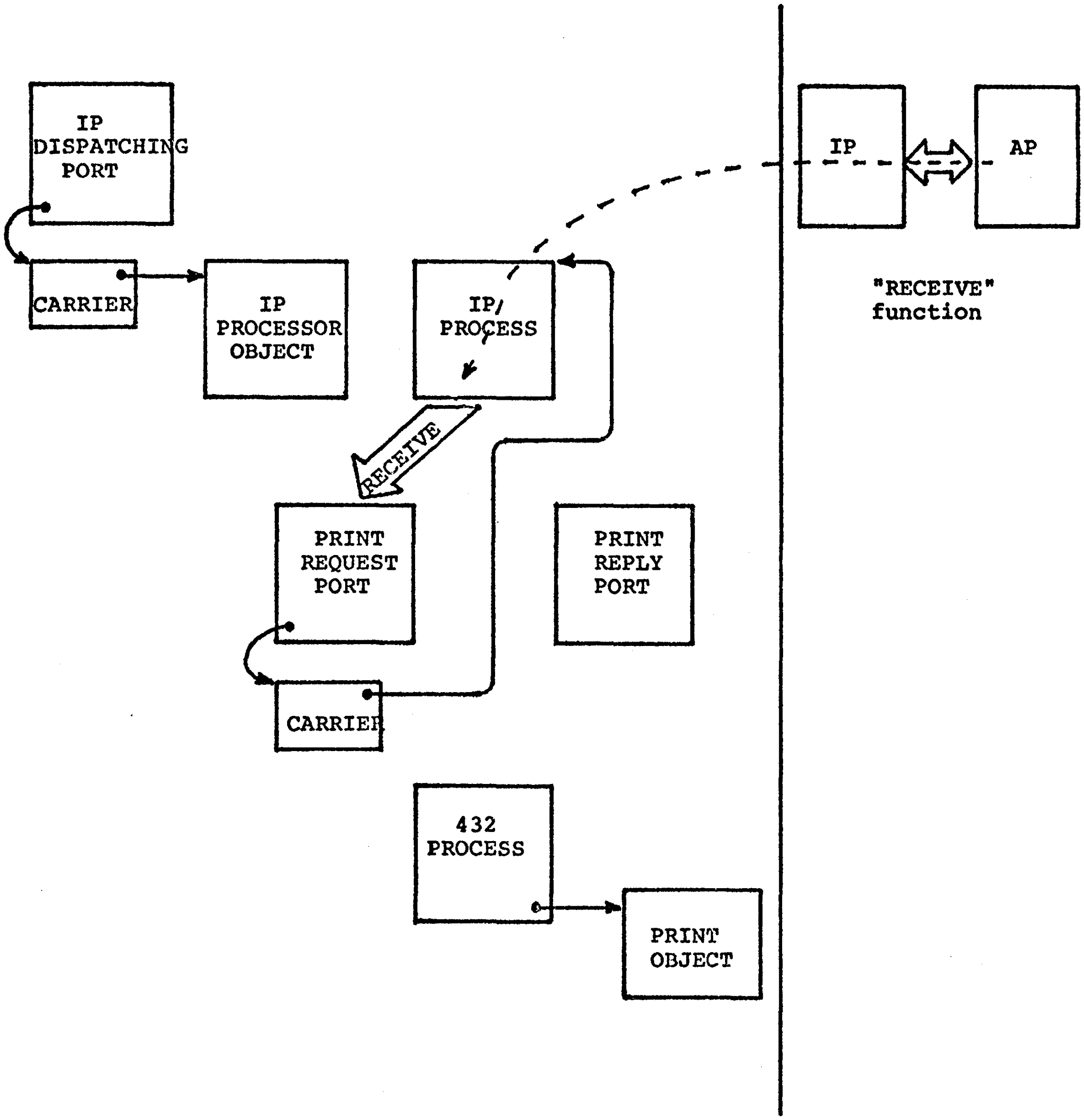


Figure F-2 IP Performs Blocking Receive

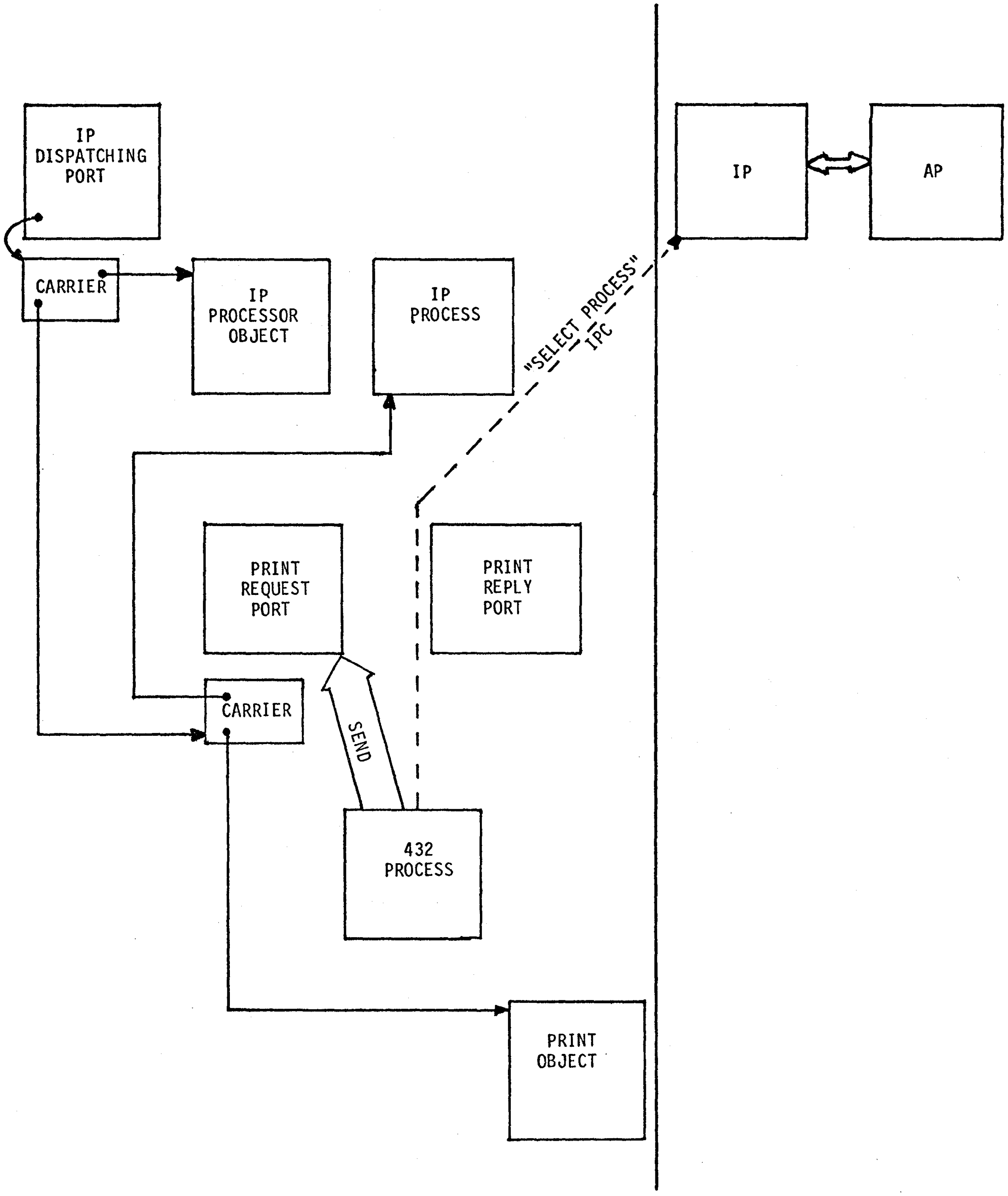


Figure F-3 GDP Executes SEND and Unblocks RECEIVE

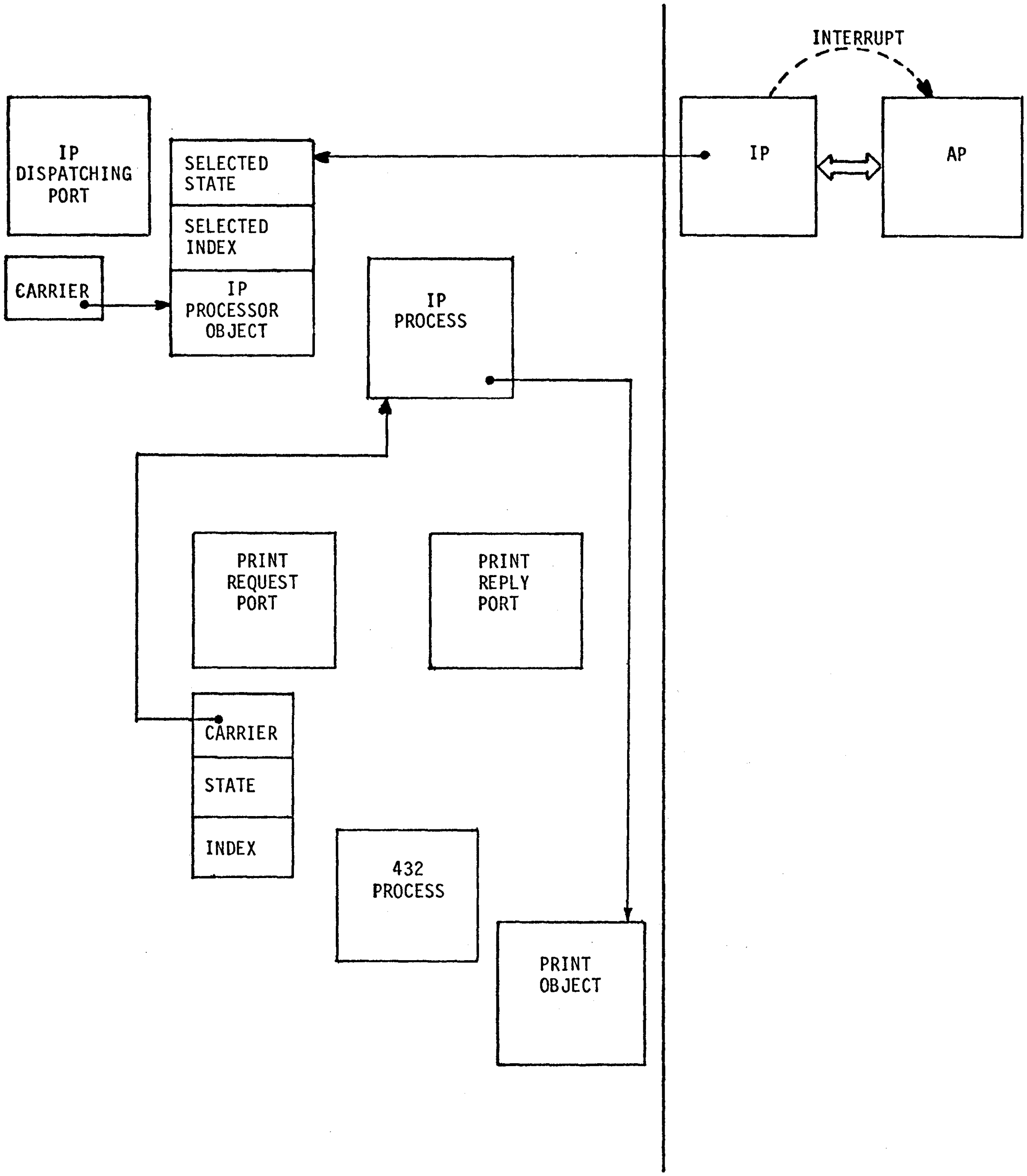


Figure F-4 IP Responds to IPC

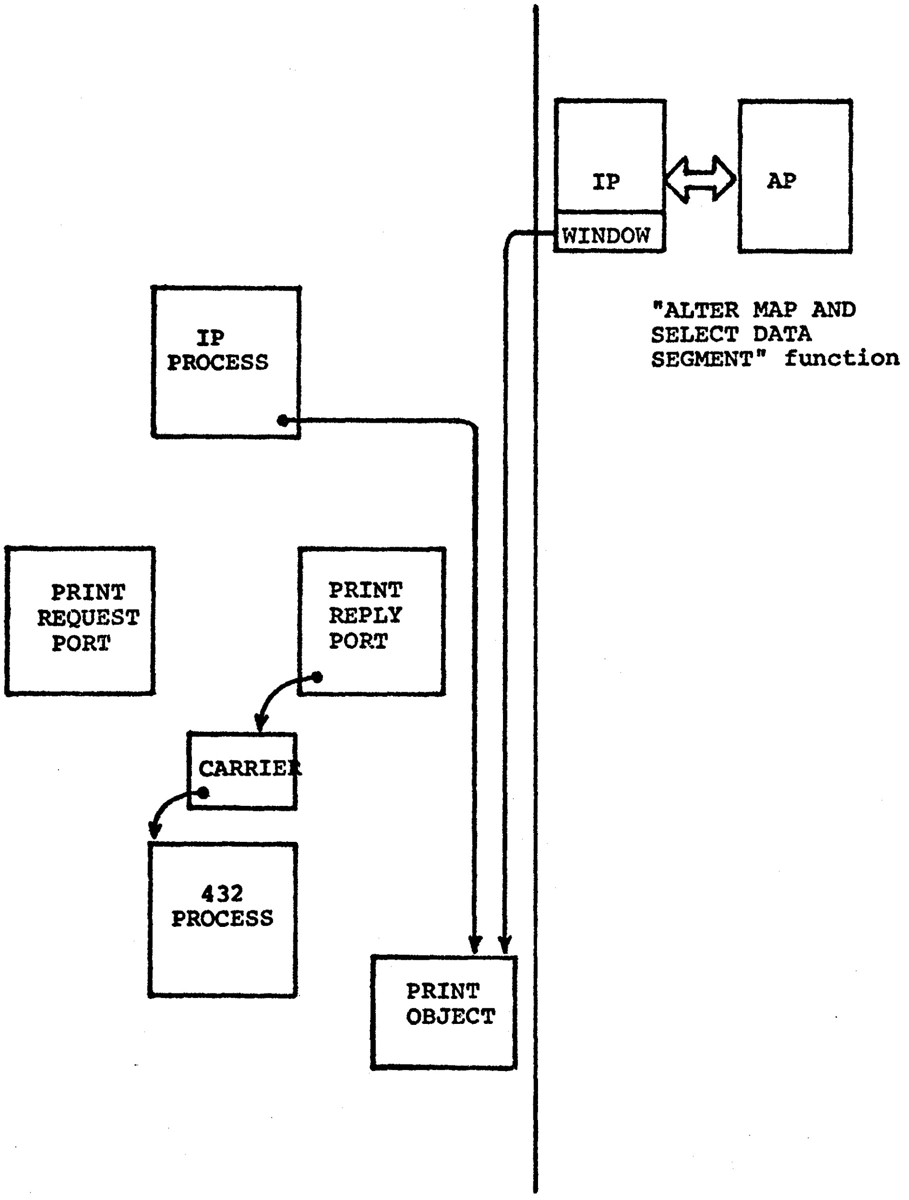


Figure F-5 Window Manipulation

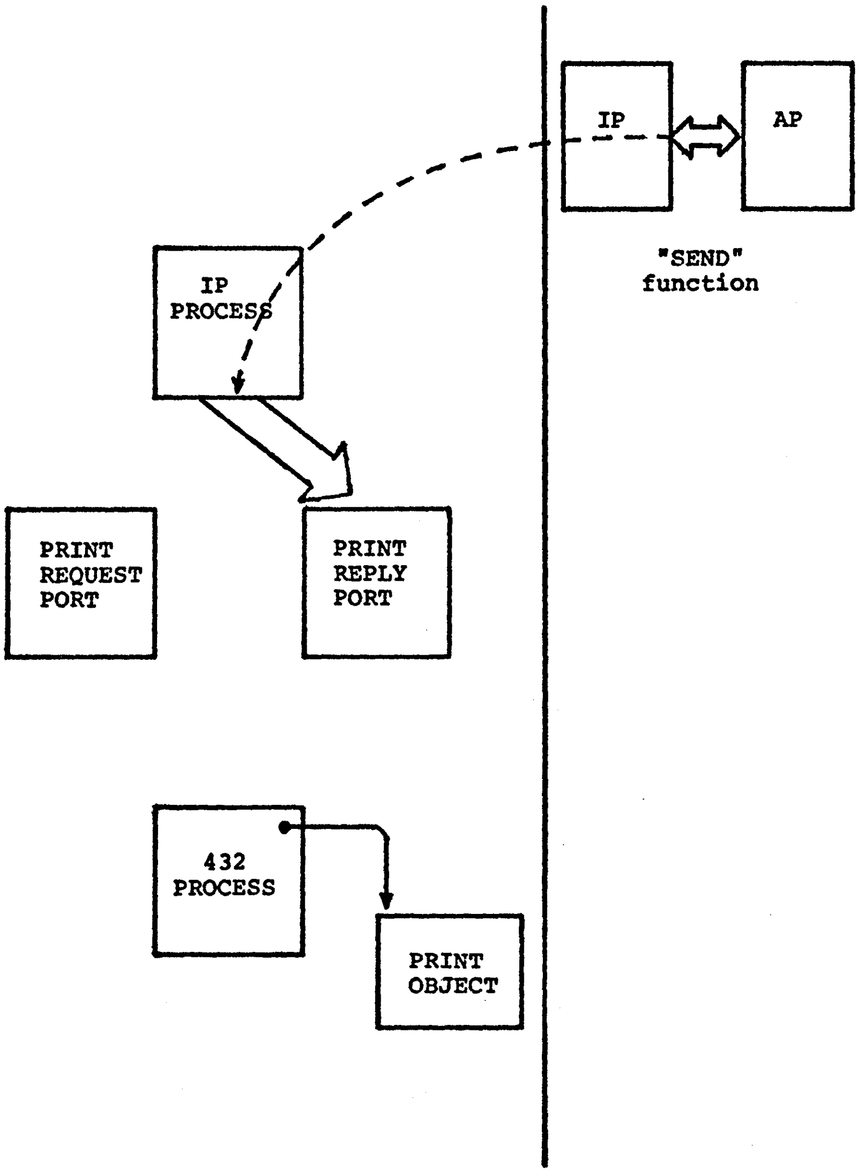


Figure F-6 Print Reply

iAPX 432 Interface Processor
Architecture Reference Manual
171863-001

REQUEST FOR READER'S COMMENTS

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

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

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

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

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

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

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

NAME ______________________________ DATE __________
TITLE ______________________________
COMPANY NAME/DEPARTMENT ______________________________
ADDRESS ______________________________
CITY ____________________ STATE ______ ZIP CODE __________

Please check here if you require a written reply. □

E'D LIKE YOUR COMMENTS . . .

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

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

AL3-2-485

intel®

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

Printed in U.S.A./Y63/1K/0781/AP