iAPX 43201/iAPX 43202
FAULT TOLERANT GENERAL DATA PROCESSOR

- **Range of Performance**
  - **— Adding Processors Increases Performance**
  - **— No Software Changes Required**
- **Large Address Space:**
  - **— 16 Megabytes Physical Memory**
  - **— 1 Terabyte Virtual Memory**
- **Capability-Based Addressing for Maximum Dependability**
  - **— Most Software Faults Trapped Before Damage Occurs**
  - **— Debugging Time Reduced**
  - **— Leads to Highly-Reliable, Robust Systems**
- **Memory Management Support On-Chip**
- **Symmetrical Instruction Set for all 8-, 16-, and 32-bit Data Types**
- **IEEE Standard 32-, 64-, and 80-Bit Floating-Point Operations**
- **Master/Checker Pairs Detect Hardware Errors Automatically**
- **Quad Modular Redundancy Ensures Immediate Recovery From Hardware Faults**

The iAPX 432 Micromainframe is a 32-bit multiprocessor specifically designed for those critical applications which demand absolute software reliability or hardware fault tolerance. At the heart of the system is the iAPX 432 General Data Processor (GDP) consisting of two VLSI components, the iAPX 43201 and iAPX 43202. Together with the other members of the iAPX 432 component family (i.e., the 43203 Interface Processor, the 43204 Bus Interface Unit, and the 43205 Memory Control Unit) the GDP can be used to build a fault-tolerant computer system that sustain any single-point failure and yet continue to operate correctly, without interruption.

Intel 432 systems offer a range of performance: by adding or removing GDPs, the throughput of a 432-based product can be increased to support more users or reduced to save hardware cost with no change to software. Thus, a family of end products with differing levels of performance can be developed from identical hardware modules using the same software.

The iAPX 43201 and iAPX 43202 are fabricated with Intel's highly reliable +5-Volt, depletion load, N-channel, silicon gate HMOS technology and each is housed in a 68-pin JEDEC Type A package. See Figures 1 and 2.

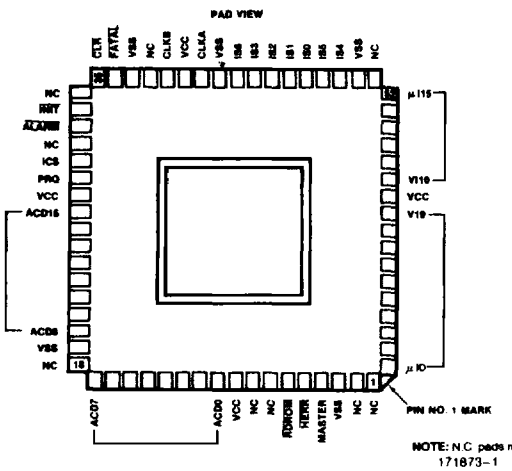


Figure 1. 43201 Pin Assignment, Instruction Decoder/Microinstruction Sequencer

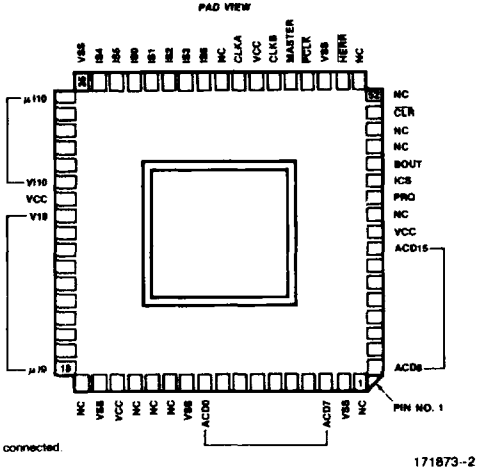


Figure 2. 43202 Pin Assignment, Execution Unit

November 1983
Order Number: 171873-002

Table 1 summarizes all signal groups, signal names and their active states, and whether or not they are monitored by the Hardware Error Detection circuitry.

Table 1. Pin Description

Symbol	Type	Name and Function
Processor Packet Bus Group		
ACD_{15} ACD_0	I/O	**Address/Control/Data Lines:** These 16 bidirectional signals carry physical memory addresses, control information (access length and type), and data to and from the memory bus. When the GDP is in checker mode, the ACD pins are monitored by the hardware error detection logic and are in the high impedance state.
PRQ	O	**Processor Packet Bus Request:** is issued to gain access to the bus. Normally low, the PRQ pin is brought high during the same cycle as when the first double-byte of address information appears on the ACD pins. PRQ remains high for only one cycle during the access, unless an address development fault occurs. In that case, the GDP leaves PRQ high for a second cycle to indicate it has detected an addressing or segments rights fault in completing the address generation. PRQ is checked by the hardware detection logic and remains in a high impedance mode when the GDP is in checker mode.
ICS	I	**Interconnect Status:** carries information on errors, data synchronization, and interprocessor communication. The interpretation of this signal depends on the current cycle of the bus transaction. See page 39 for a complete description.
B_{OUT}	O	**Enable Buffers for Output:** controls the direction of external buffers, if any are used. When B_{OUT} is asserted, it indicates that the buffers must be directed to carry information outbound from the GDP.
System Group		
V_{CC}	I	**Power:** These four pins supply 5-volt power to the GDP, and all must be connected; the pins are not connected together within the GDP.
V_{SS}	I	**Ground:** These five pins provide the ground reference for the GDP, and all must be connected; the pins are not connected together within the GDP.
$\overline{ALARM}$	I	**$\overline{Alarm}$:** monitors the condition of an unusual, system-wide condition such as power failure. Alarm is sampled on the rising edge of CLK_A.
$\overline{FATAL}$	O	**$\overline{Fatal}$:** is asserted by the GDP under microcode control when the GDP is unable to continue due to various error or fault conditions. Once $\overline{FATAL}$ is asserted, it can only be reset by the assertion of $\overline{INIT}$.
$\overline{PCLK}$	I	**Processor Clock:** The assertion of $\overline{PCLK}$ for one cycle causes the system timer within the GDP to decrement. Assertion of $\overline{PCLK}$ for two or more cycles causes the system timer to be reset. $\overline{PCLK}$ must remain unasserted for at least 10 clock cycles before being asserted again. The GDP samples $\overline{PCLK}$ on the rising edge of CLK_A.

Table 1. Pin Description (Continued)

Symbol	Type	Name and Function
System Group (Continued)		
$\overline{CLR}$	I	**Clear:** Assertion of $\overline{CLR}$ results in a microprogram trap causing the GDP to immediately terminate any bus transactions or internal operations in progress. The GDP resets to a known state, asserts $\overline{FATAL}$, and awaits an IPC for initialization. The IPC is not serviced for at least four clock cycles following the assertion of $\overline{CLR}$. Response to $\overline{CLR}$ is disabled by the first $\overline{CLR}$ assertion and is reenabled when the GDP receives the first IPC (or $\overline{INIT}$ assertion). The GDP samples $\overline{CLR}$ on the rising edge of CLK_A.
$\overline{INIT}$	I	**Initialize:** Assertion of $\overline{INIT}$ resets the internal state of the GDP and starts execution of the initialization microcode. $\overline{INIT}$ must be asserted for a minimum of 10 clock cycles. After the $\overline{INIT}$ pin returns to its nonasserted state, the GDP initializes all of its internal registers and windows, and waits for a local IPC. $\overline{INIT}$ is sampled on the rising edge of CLK_A.
MASTER	I	**Master:** This signal determines whether the 43202 is to function as a master or a checker. In master mode, the 43202 functions normally and drives all of its outputs. In checker mode, ACD_{15}-ACD_0 and PRQ enter a high impedance state and B_{OUT} is unconditionally low. A 43202, whether master or checker, monitors the ACD_{15}-ACD_0 and PRQ lines and compares the data on them to its internally generated result, signaling disagreement on the $\overline{HERR}$ line. For normal operation, MASTER should be left unconnected or tied high.
$\overline{HERR}$	O	**Hardware Error:** This signal is asserted by the GDP to indicate disagreement between data appearing on the ACD_{15}-ACD_0 and PRQ pins and the internally generated result of the GDP. $\overline{HERR}$ is valid during CLK_A and can normally be asserted by a GDP and every clock cycle, except immediately following $\overline{CLR}$. $\overline{HERR}$ requires an external 2.2 kΩ nominal pullup resistor.
CLK_A	I	**Clock A:** is a square-wave clock which must operate continuously to preserve the operating state of the GDP.
CLK_B	I	**Clock B:** is a square-wave clock which operates at the same frequency as CLK_A, but lags it by 90 degrees. CLK_B must operate continuously to preserve the operating state of the GDP.
Intra-GDP Group		
IS_6–IS_0	N/A	**Interchip Status Lines:** carry microprogram information from the 43202 back to the 43201. The lines are not checked by the hardware detection logic.
μI_0–μI_{15}	N/A	**Microinstruction Bus Lines:** carry microinstructions from the 43201 to the 43202. They are not checked by hardware detection logic.
$\overline{RDROM}$	I	**Read ROM:** This signal is used to read the microprogram from the 43201 ROM. If $\overline{RDROM}$ is held low when $\overline{INIT}$ goes high, the 43201 enters a diagnostic mode, and the microinstruction sequencer steps through the microprogram ROM, sequentially displaying (but not executing) the microinstructions on the μI_{15}-μI_0 lines. While $\overline{RDROM}$ is useful for testing, it should be tied to V_{CC} for normal operation.

FUNCTIONAL DESCRIPTION

Introduction

The iAPX 432 Micromainframe is a 32-bit multiprocessor especially designed for those critical applications which demand absolute software reliability or hardware fault tolerance. By developing the 432, Intel has broken with three decades of tradition that have defined how computers operate and redrawn the line separating functions of hardware and software. Many operations that 432 processors perform automatically would be done by the operating system in conventional machines. The development of the 432 was driven by two major objectives: to reduce the cost of software over the life cycle of the product, and to develop a computer with unprecedented reliability. From any perspective, the 432 is an uncommon machine.

Similar to many mainframe computers, processing in the 432 is divided between a central system, which handles data processing, and one or more peripheral subsystems, which transfer data to and from I/O devices. There are two types of processors, General Data Processors (GDPs) and Interface Processors (IPs), and two types of support components, Bus Interface Units (BIUs) and Memory Control Units (MCUs). Together, these VLSI components can be used to build a fault-tolerant computer system that is able to sustain any single-point failure of a component or bus, and yet continue to operate correctly, without program interruption and without software intervention.

This concern for reliability in the 432's design is not limited to automatic recovery from hardware faults, but extends to software as well. The 432 processors can detect hundreds of different software fault conditions from an attempt to divide by zero or execute data, to complex faults involving several independent processes. While most computers do not detect these faults at all, the 432 is able to trap and identify most faults before serious damage can occur. As a consequence, 432-based systems are easier to debug, and systems shipped to end-users will prove substantially more reliable.

Another important advantage of the 432 is the ability to tailor the throughput of the system to meet the price/performance requirements of each application or end-user. A family of products, for example, can be developed using the same hardware modules, simply adding or removing boards as the application requires. The end-user, for example, could buy an entry-level system with only two processors. The system would run more slowly than the high-end system, but it would also cost much less. Later, when the user's needs grew, additional General Data Processors could be installed to handle the heavier load on the system. No need to change software; all the programs that the user had developed would still be compatible. In fact, if at any time, one of the processors failed, the user could remove it, and the remaining processors in the system would continue to execute programs correctly while a replacement was sought.

This unprecedented flexibility is possible only because the 432 takes a unique approach to architecture, one closely tied to the structure of programs. The processors are no longer passive entities, responding only to software, but they can execute many functions automatically to keep the system working efficiently and reliably.

ARCHITECTURE

This section describes the architecture of the iAPX 432; that is, the machine-level programmer's view of the computer. As a rule, only compiler writers actually deal with the 432 at this level; however, many application programmers—who typically code in Ada—will find this discussion valuable for getting a "feel" for the operation of the underlying machine. Bear in mind that this discussion does not cover all programming facilities and some of the concepts have been simplified for the sake of clarity and space; a complete description of the architecture can be found in the **iAPX 432 General Data Processor Architecture Reference Manual.**

Since the 432 is a multiprocessor system, with specialized types of processors optimized for different kinds of work, the architecture of the GDP is different from the Interface Processor. At the same time, these diverse processors (and their associated software) must cooperate with each other to accomplish the overall task of the system. Therefore, the designs of the GDP and IP share an architectural foundation, the 432 common base architecture. The overall arrangement is illustrated in Figure 3.

Common Base Architecture

The common base architecture of the 432 is the glue that binds multiple processors, of the same and different types, together in a coherent system. Therefore, the common base architecture defines "global" facilities used by all processors, software, and support components (i.e., iAPX 43204 Bus Interface Unit and iAPX 43205 Memory Control Unit). These facilities include addressing, protection, object management, communication, timekeeping, and exception processing.

Objects

Memory protection systems can be considered in terms of boxes into which information can be locked and keys which can open the boxes. The 432's boxes and keys correspond to the logical units found in many high-level programming languages (e.g., Ada, Pascal); in other words, programming units and protection units match. Boxes are variable in size, each box containing one **object**; the object may be an array, a record, a list element, the text of a procedure—in fact, any logical entity which should be uniquely identified and protected.

Keys in the 432 are called access descriptors and are manifested in high level languages as pointers (instances of access variables in Ada). Whenever storage for a new object is allocated, whether by the linker or program statements, the processor automatically boxes the object (in its own segment), manufactures a unique key (access descriptor), and returns the key to the creating procedure. A brief introduction to objects can be found in Table 2.

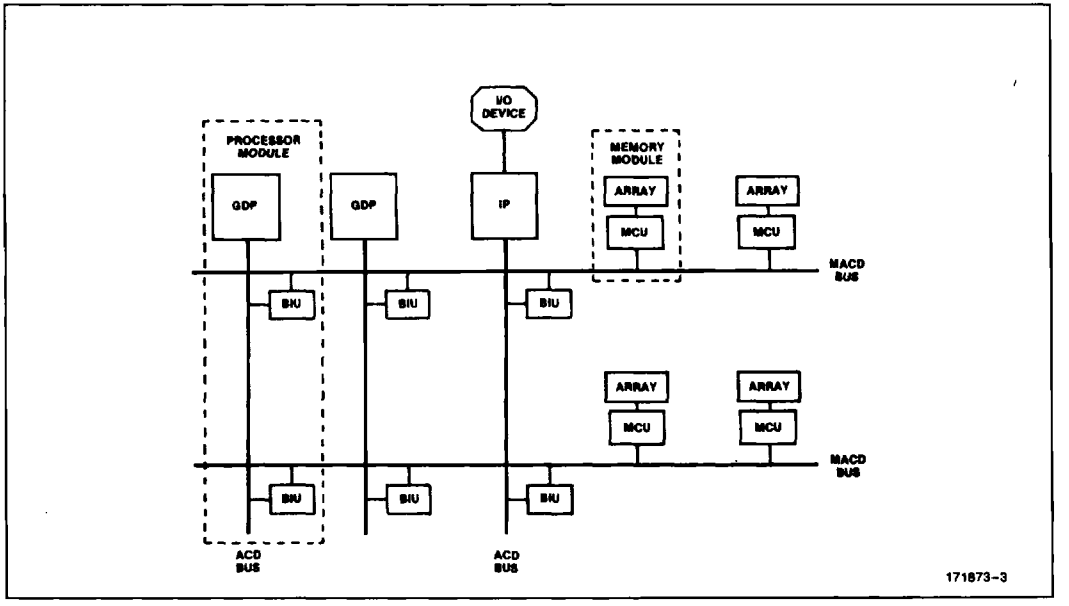


Figure 3. A typical 432 system with two memory buses, two General Data Processors, and a single peripheral subsystem for I/O

Table 2. Introducing Objects

1. **All information in an iAPX 432 system is contained in objects.**
   Even the instruction pointer, status flags, and other information used by the GDP are contained in objects.

2. **Each object can have two parts, a data part and an access part.**
   The data part can contain any information except accesses. Data in the data part can be added, assigned, manipulated as bit fields, or used for any purpose other than accessing an object. The access part can contain only **access descriptors** (ADs). ADs are used for referencing objects and can only be modified in carefully controlled ways.

3. **Objects can be created with different lengths.**
   An object can have from 0 to 65,536 bytes in its data part, and from 0 to 16,384 ADs in its access part. Any reference to an object is automatically checked to ensure that it falls within the bounds of the object.

Table 2. Introducing Objects (Continued)

4. **Each object has a fixed type.**
   The type of an object is determined when the object is created. An object's type can be used to define the operations allowed on the object. Software can define new object types at run-time.

5. **Objects can be local to a program or subprogram call.**
   Each object is created at a particular **level** that specifies whether the object is global or limited in scope to a particular program or subprogram activation.

6. **Objects can only be read or written via access descriptors.**
   To access data in an object, you must specify an AD that references the object and also specify the offset within the object's data part to the field accessed.

7. **A procedure call can only access those objects it has ADs for.**
   Each activation of a program or procedure is itself represented by a **context object**. The instructions executed by the context can only access those objects for which the context has ADs or can obtain ADs.

8. **Access descriptors can provide restricted access to objects.**
   Each AD specifies several **rights** bits, including read rights and write rights. To read from an object requires read rights set on the AD used; to write to an object requires write rights on the AD used. Different module activations can have ADs for the same object, but with different rights.

Storage

Address Spaces There are several distinct address spaces in a iAPX 432 system. Each peripheral subsystem, for example, has a local memory space, and typically, a local I/O space. A portion of each peripheral subsystem memory space is mapped by an Interface Processor into the central system. Processors (and DMA controllers) in a peripheral subsystem can gain access to central system data by reading and writing these local mapped address spaces.

The central system is divided into two 16-Megabyte physical address spaces, the **memory space** and the **interconnect space**. For the most part, the interconnect space consists of hardware registers used by the Bus Interface and Memory Control Units to maintain fault-tolerant systems. The MCU, for example, logs the number of memory errors it has detected and corrected in a register that processors are able to read by addressing the interconnect space.

GDPs use the instruction MOVE FROM INTERCONNECT SPACE to read these registers and MOVE TO INTERCONNECT SPACE to write to them. Peripheral subsystem software can gain access to the interconnect space through Window 1. All other 432 instructions and commands reference the memory space.

Logical addressing To operate on a data item, a GDP instruction (or IP command) presents a logical address as shown in Figure 4. For a GDP, a data item is an integer, character, or real number. In the case of an IP data transfer, such data items are simply bytes or double bytes. The logical address of a data item consists of an access descriptor and a displacement (offset) into that object. The several ways of specifying these components, particularly the displacement, give rise to the GDP's addressing modes, which are described in a later section.

Physical Translation GDPs translate logical addresses into 24-bit physical address; programs have no way to generate physical addresses directly. As shown in Figure 5, the essence of the translation is finding the physical base address of the target object and adding the displacement component to it.

As you'll recall, associated with each 432 procedure (or function) is an object reference list (set of keys) called access descriptors; this array of access descriptors defines the objects that are currently addressable by the procedure. The access descriptor in turn selects an entry in an object table. This entry, called an object descriptor, contains the base address of the target object.

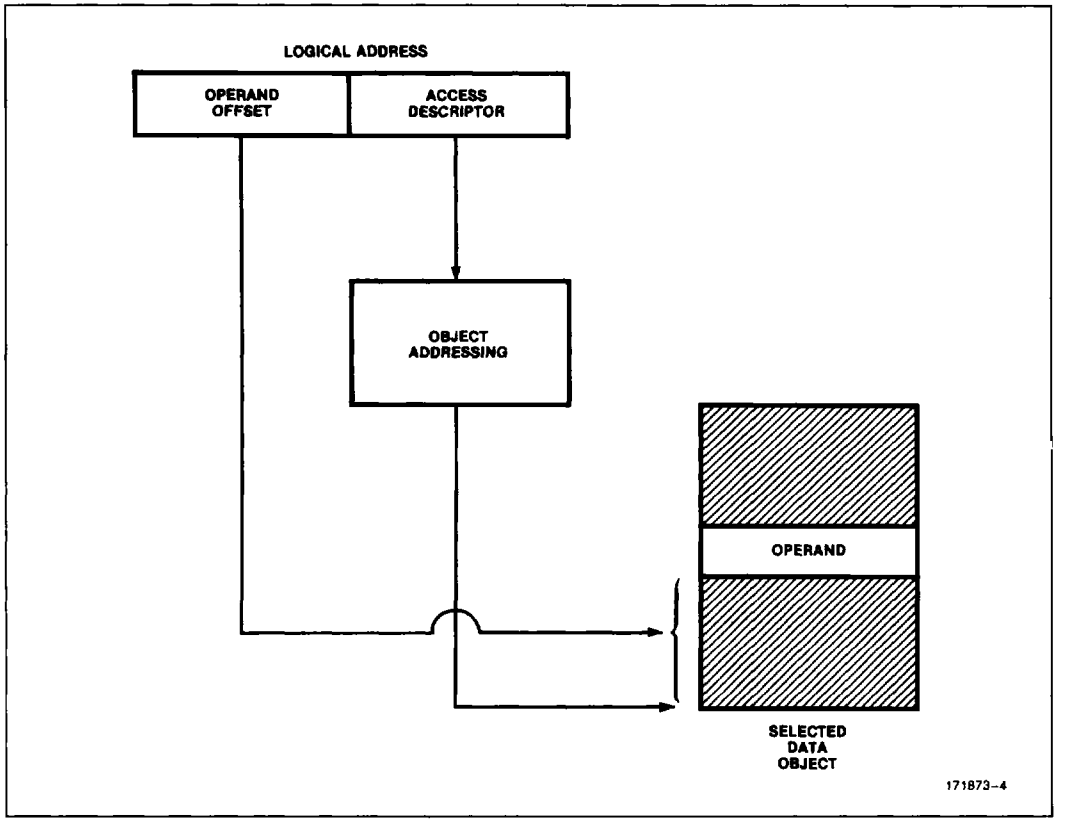


Figure 4. Simplified View of Logical Addressing

The focal point of the translation procedure is the object table. (Naturally, only the operating system's memory manager has a reference to the object table itself.) Every object in a system is represented by a single object descriptor, which contains its base address and other descriptive information. Conversely, there may be many access descriptors for a single object scattered among the access descriptor lists of all programs that have access to that object. Moving an object (to compact physical memory, for example), only requires updating its object descriptor, regardless of how many programs hold references to it.

Note that to improve performance, 432 processors automatically maintain two groups of physical addresses on chip. The addresses of frequently used system objects, such as the object table, are always immediately available. In addition, the processors cache exploits the tendency of most programs to cluster their references to a few objects at a time.

Virtual Memory Support Three fields in object descriptors aid a 432 executive's virtual memory manager. The **allocated** bit indicates whether real memory is associated with the object. When the virtual memory manager swaps an object out to external storage, it clears this bit first. The hardware checks the allocated bit during address translation; if it is clear, the hardware faults, and control passes to the memory manager. The memory manager can then swap the object back into physical memory, set the allocated bit, and return control to the instruction that faulted.

The **accessed** bit is set by the hardware when an object is referenced by an instruction. By periodically checking and clearing this bit in all object descriptors, the virtual memory manager can gain insight into the frequency with which each object is being used. This information can then be used to select objects to be swapped out, for example, on a least-recently-used basis.

The **altered** bit is set by hardware when an object is updated. If a virtual memory manager decides to swap an object that has not been altered, and it knows that a copy already exists in external storage, it can save time by discarding the memory-resident copy without swapping it. Both the accessed and altered bits are also cached on-chip to keep object table references to a minimum.

Protection and Access Control The services provided by modern operating systems typically take the form of procedures that can be called by user processes. Since most computers do not provide procedure-level protection as a matter of course, a special arrangement is necessary to protect operating system procedures from their callers (an OS procedure runs in the same process as its caller).

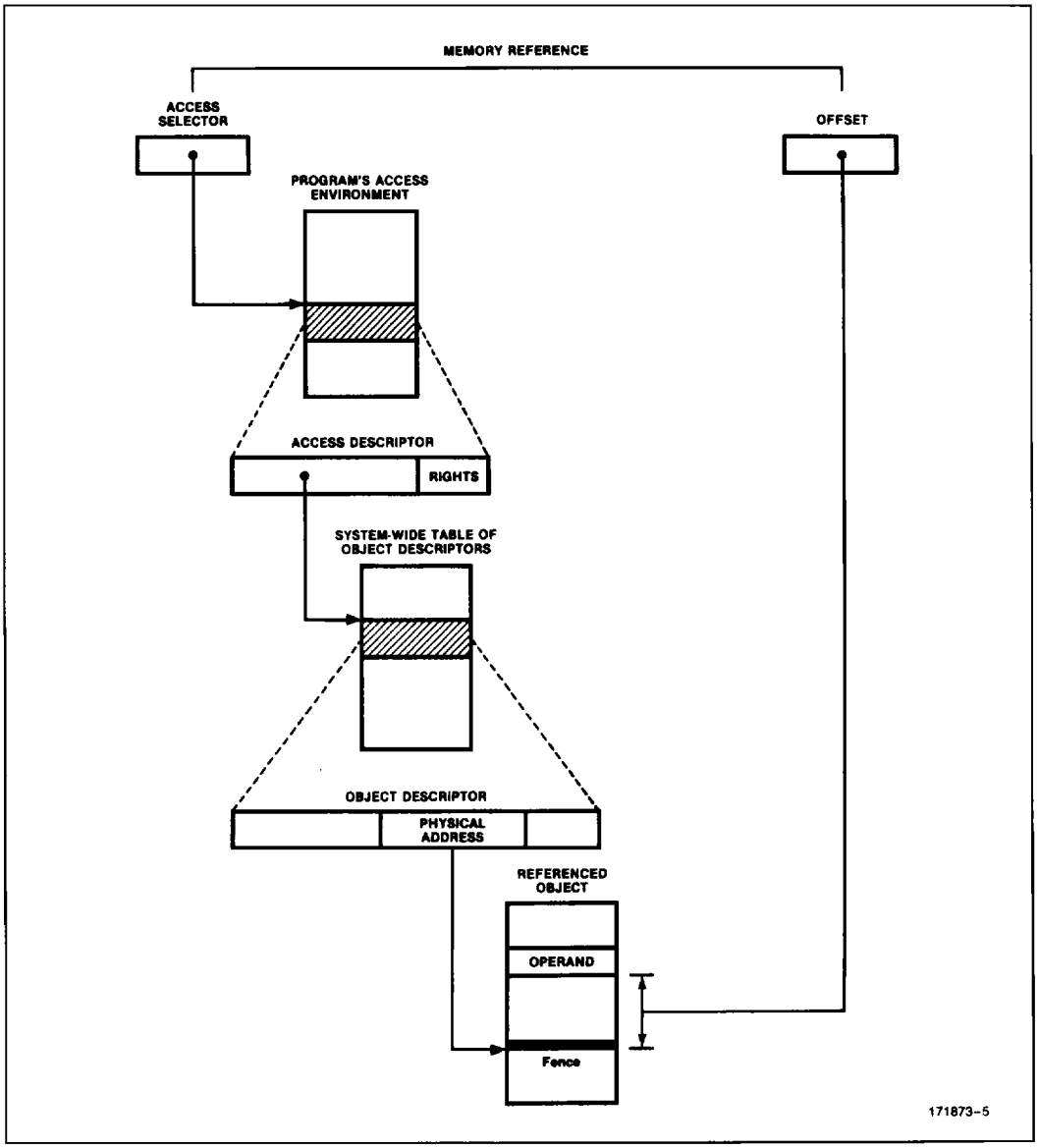


Figure 5. Two-Level Address Mapping

The usual approach is to confer **privilege** upon OS procedures while denying it to user processes. In effect, privilege is a second lock that operating system procedures can apply to boxes containing their information. If a computer has one privilege level (e.g., privileged supervisor, unprivileged user) then one key opens any privilege lock. A 4-level privilege scheme (e.g. kernel, executive, supervisor, user) provides more locks, one for each level. Each privileged key, however, not only opens all locks of the same level, but is effectively a master key for all lower privilege levels.

Privilege thus provides an asymmetric form of protection: information stored at a given level is protected from lower-level procedures only, not from procedures at the same or higher levels; in fact a procedure at the lowest level runs on an unprotected machine.

The 432's protection system, in contrast, is symmetric: every procedure—part of the operating system or not—is equally protected from every other because every procedure has access only to those objects it absolutely needs. The 432's run-time protection system is based on limiting the distribution of information and controlling the manner in which accessible information is referenced. Each object is protected as a unit, and each executing procedure has a unique access environment that specifies exactly which objects it may use, and in what manner it may use them. At the same time, the system is flexible enough to permit straightforward sharing of information between cooperating processes.

Limited Access Environments At any given point in time, a procedure's access list constitutes its current access environment. Since addresses can only be constructed from the access descriptors on this list, it is impossible for a procedure to gain access to any object for which it does not hold an access descriptor.

A procedure creates an initial access environment (called a context) for a procedure as part of its execution of the CALL instruction that invokes the procedure. (Actually, the entire context is not created each time a procedure is called, rather, to increase performance, each process is given a set of inactive contexts after compilation. When a procedure is called the context is "filled in" with the parameters for that activation.) This initial environment includes references for all objects that could be ascertained at compile-time: objects containing constants and statically allocated variables, and any procedures called by this procedure. In addition, the processor creates a reference to any parameters passed by the caller, to free storage from which new objects may be allocated, and to an operand stack for use in evaluating expressions in this activation.

As the procedure executes, it may modify its own access environment. For example, it may create a new object, it may receive an object reference from another process, or it may delete an object reference that is no longer needed. Notice that while the procedure may create new data at will, the only way for it to obtain access to data outside its initial environment is through the explicit cooperation of another process (i.e., through the interprocess communication facility called ports).

By entering a new context for each procedure activation, the 432 provides inherent support for reentrant and recursive programming. The complete change in access environments caused by the procedure invocation ensures that the called procedure inherits no part of its caller's environment, except for the parameters that have been passed to it. When the called procedure returns, the processor switches back to the caller's environment.

Integral Sharing Sharing is the other side of protection. To be able to share pieces of memory among processes in a controlled way is an important capability for a computer. Popular procedures and programs (especially editors and compilers) are often executed by many processes simultaneously; sharing a single copy of the code among all processes not only saves memory, but reduces virtual memory I/O traffic and disk space requirements.

The ability to share data is as important as sharing instructions; some applications, especially control-oriented ones, are most easily implemented as multiple processes sharing key data structures. The alternatives to memory sharing (maintaining separate copies for each process or sharing a file instead of memory) are often too wasteful, too complicated, or too slow to be practical in applications where the processes must have fast access to up-to-date information.

Unfortunately, most computer addressing schemes favor either protection or sharing. To protect processes from each other, many computers map them into separate address spaces; sharing an item then requires construction of a bridge between the two address spaces, which—when done at all—provides an awkward and incomplete facility. Other computers run processes in the same address space, enabling sharing but sacrificing protection (i.e., processes share everything).

The 432 has a single address space; every object in this space is potentially sharable by every process in the system. Protection is obtained by restricting the object each procedure can address to the subset it "needs to know" to perform its function; at any time the subset is defined by the procedures list of access descriptors.

An object is shared if procedures in more than one process has references for it (conversely, an object is private if there is only a single reference for it). The references can be provided by the compiler or can be obtained at run-time. (Note that interprocess sharing is a requirement of Ada and other languages that support concurrent programming.)

Typing Given that a procedure has access to an object (has a reference for it), the protection system insures the basic integrity of that object. That is, instruction references to the object are checked to make certain that they make sense for that object. By **typing** each object, the 432 hardware can positively determine what an object is; this, in turn, implies which operations (i.e., instructions) are valid for the object and which are not. In order to maintain good performance, however, the individual data items (e.g., characters, integers, etc.) within an object are typed and typed-checked at compile-time, not run-time.

Types of Objects There are five classes of objects that are used in a 432 system:

- System objects
- Generic objects
- Dynamic-type objects
- Refinements of any of these objects
- Interconnect objects

System objects have specific uses and formats recognized by 432 processors, and serve as the backbone of the architecture and the operating system. The 432 provides a very high-level of support for operating system functions because the processors are able to recognize and manipulate system objects. Processors operate on system objects in two ways: by executing high-level instructions on behalf of the operating system, and by executing certain functions on their own initiative, without software intervention.

The structure of a high-level program is reflected in the type and function of system objects. An Ada package, for example, is represented within the 432 as a **domain object.** The domain object contains a list of procedures, functions, and data associated with the single package; and like a package, is divided into **private** and **public** parts. Procedures in the public part of a domain correspond to procedures which are declared in the specification part of a package, and which are accessible to outside users. The private part of the domain, in contrast, corresponds to those data structures, functions, and procedures within the implementation section (body). No outside user can operate on, or directly call, these structures.

Instruction and data objects are good examples of the protection that an object-based architecture provides. Instructions contained in an instruction object cannot be manipulated as data, and likewise, the entries in a data object cannot be fetched by a processor for execution. Further, both the instruction and data objects are hidden from users in the sense that access to them is provided only through a procedure call.

The type managers for system objects are combinations of hardware and operating system software, so application programmers have no need to understand the internal organizations (representations) of these objects. A few of them, however, are cornerstones of the 432 architecture and it is worthwhile to examine them from a functional point of view. **Instruction objects,** from which processors fetch a procedure's machine instructions, have already been mentioned. Later in this section **storage resource** and **port** objects will be discussed.

Processor and **process objects** are used, naturally, to manage processors and processes. There is one processor object for each GDP and IP in the system. At initialization time, each processor obtains a reference for its own processor object, which it holds on-chip indefinitely. There is also one process object for each process (task) in the system. A processor object contains an access descriptor for a process object. Dispatching a new process to run on a processor amounts to changing this access descriptor. The entire set of system objects is described briefly in Table 3 and illustrated in Figure 6.

Generic objects An object whose type is **generic** is one that has no special significance to the hardware; its system type is effectively "none." Creating generic objects is faster than creating other types of objects and requires no special privilege. In languages such as Ada or Pascal, executing a NEW operation creates a record referenced by a pointer. In the 432, the operation creates a generic object and an access descriptor to reference it.

Dynamic-Typed Objects OEM-developed software added to a computer needs protection for exactly the same reasons that the operating system does. Coexisting with unknown programs that have been written by fallible (and sometimes malicious) end-users, the OEM software must keep running correctly and must prevent the disclosure or alteration of sensitive information belonging to either the OEM or the end-user. The principal weakness of conventional protection systems is that the hardware vendor monopolizes the protection facilities of the machine, leaving only interprocess protection for the OEM or end-user. In the 432, full protection can be extended to every new facility, whether added by Intel, the OEM, a software vendor, or the end-user.

Table 3. iAPX 432 System Objects

Instruction Object
contains GDP instructions; the GDP will fetch instructions only from instruction objects.

Domain
represents a program module (package) and references subprograms (instruction objects) and data objects in the module.

Context
represents a program or subprogram activation (call) and defines the access environment of the call, i.e., the set of objects that the activation can reference.

Type Definition Object (TDO)
represents a software-defined object type, and can contain attributes of the type (e.g., the type name).

Type Control Object (TCO)
represents type-specific privileges, such as the right to create objects of a particular type or to gain access to objects of a particular type.

Object Table
contains the object descriptors used in object addressing and memory management.

Storage Resource Object (SRO)
represents a free storage pool used to create new objects; references an object table that will contain the new object's descriptor, a physical storage object from which the new segment will be allocated, and a storage claim object that limits allocation from this SRO.

Physical Storage Object (PSO)
specifies free storage blocks in memory.

Storage Claim Object (SCO)
limits the number of bytes that can be allocated from a set of SROs that reference this SCO.

Process
represents a program or subprogram activation that can execute concurrently (in parallel) with other processes.

Port
provides communication between concurrent activities. A port includes a queue of messages sent to the port but not yet received, and a queue of blocked activities waiting to receive *messages (at an empty port) or to send messages (at a port with a full message queue).*

Carrier
represents an activity in communication with other concurrent activities via ports. Carriers carry messages to and from ports.

Processor Object
contains attributes and state information for an iAPX 432 processor (e.g., a GDP). Because programs in an iAPX 432 system can only manipulate information in objects, all information about a processor that must be visible to software must be contained in an object.

Processor Communication Object
used by the iAPX 432 interprocessor communication mechanism to transfer messages between processors.

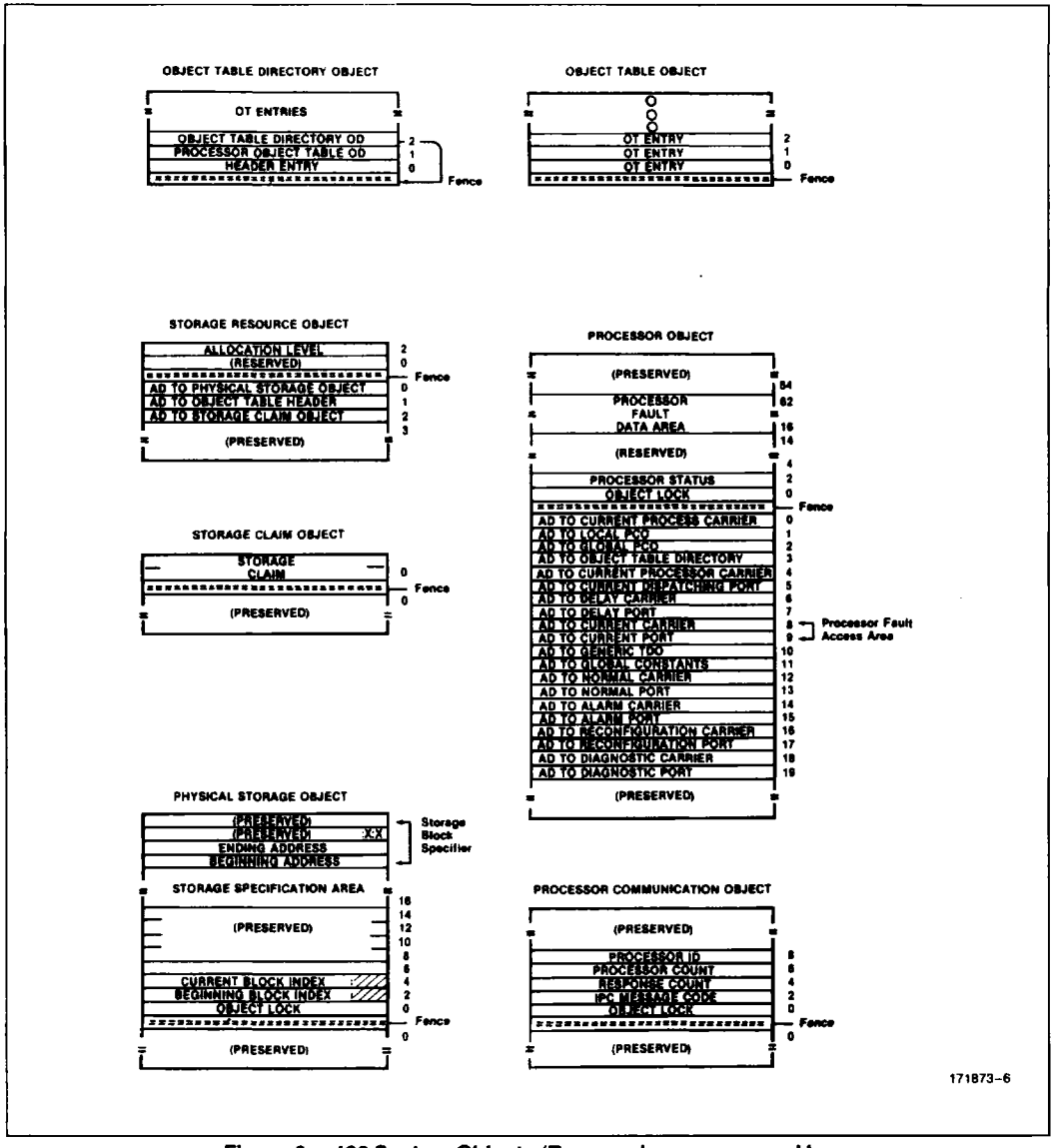


Figure 6. 432 System Objects (Reserved areas are used by processors; preserved areas are available to system software)

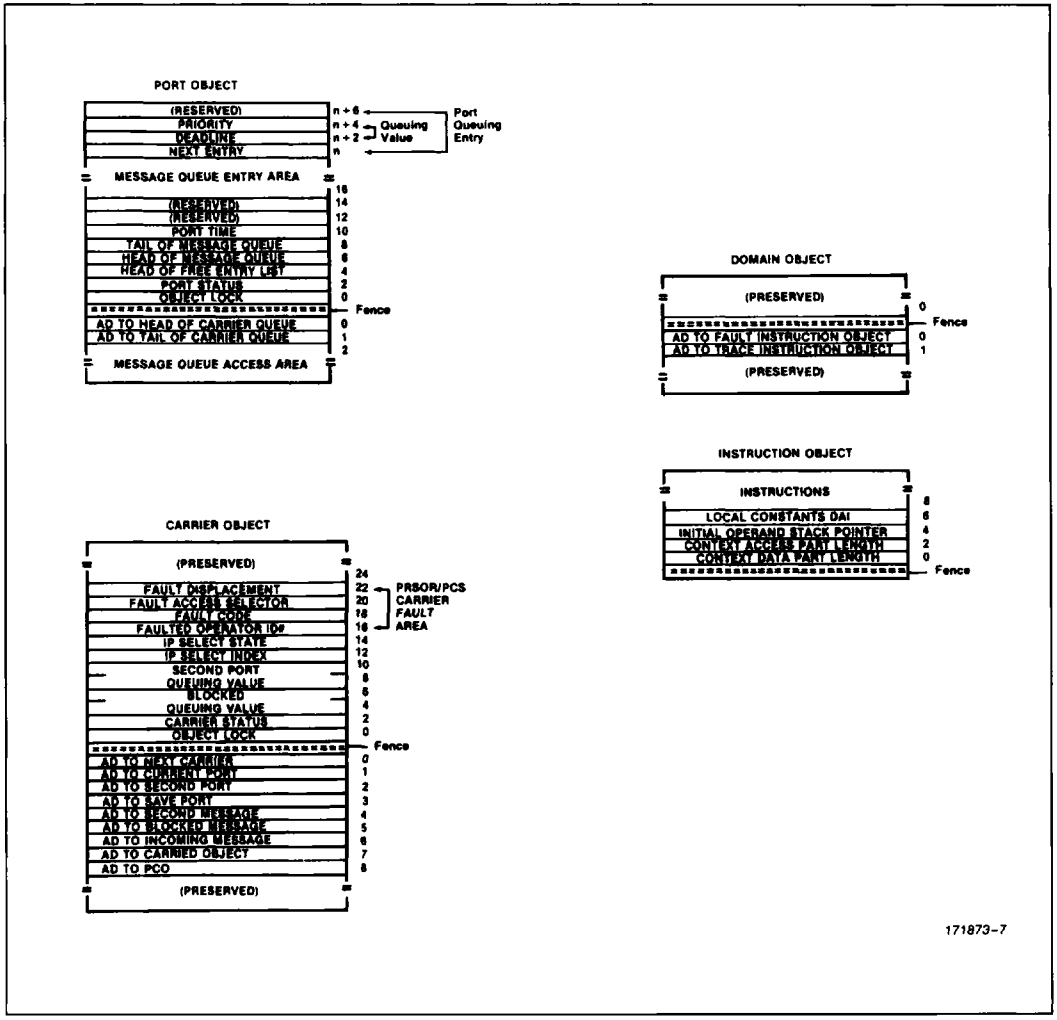


Figure 6. 432 System Objects (Reserved areas are used by processors; preserved areas are available to system software) (Continued)

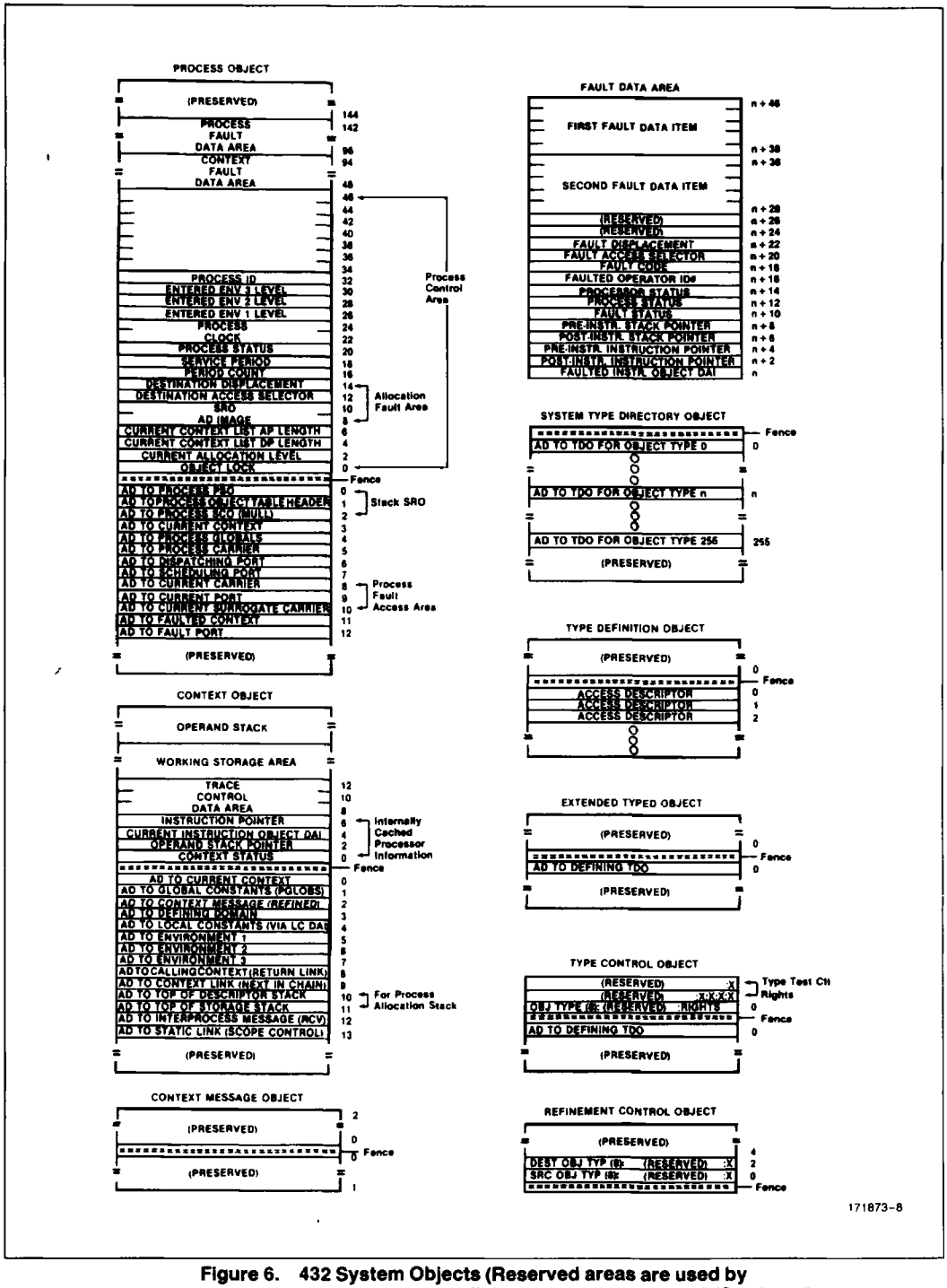


Figure 6. 432 System Objects (Reserved areas are used by processors; preserved areas are available to system software) (Continued)

If base typing protects the 432's addressing mechanism, and system typing protects the 432's system objects, dynamic-typing can be viewed as protecting users from one another. In essence, dynamic-typing enables a programmer to define a specific type of object including the operations that are valid for it, and then have the hardware enforce the definition, just as it would for any of the system objects.

When a type manager creates a dynamic-type object for a user, it returns an access descriptor for that object. When the user subsequently wishes to manipulate the object, it passes the access descriptor as a parameter in a call to one of the type manager's operations. Dynamic-typing prevents the user from manipulating the object's representation directly; the user holds a reference for the object, but the object is effectively "sealed" from access. Only the type manager can "unseal" the object; any other attempt to operate on the object's contents is aborted by the hardware.

Refinement Occasionally, it is convenient to define a new object that is a subset of an existing object. A "personnel record," for example, might contain "public" information such as name and department, and "private" information, such as salary. A process with access to the complete record may want to send only the public part to another process. It may do so by creating what is called a **refinement** of the object (see Figure 7).

Interconnect Objects Groups of hardware registers within the interconnect space are represented as Interconnect Objects, and can be accessed by the GDP only through the MOVE TO INTERCONNECT and MOVE FROM INTERCONNECT operators (Note: an Interface Processor gains access by opening Window 1 onto an Interconnect Object). Interconnect objects have no access descriptors.

Rights and Bounds Each access descriptor contains a set of rights which further defines how the holder of the reference may use the referenced object (see Figure 8). These **base** and **system** rights are set when the holder is given the reference. Base rights define whether the object may be read-only, write-only, either or neither. A user with a reference to a system object, for example, will be unable to read or write the object. System rights are system-type-specific; for example, the system rights for a port object indicate whether the holder may send a message to the port, or receive a message from the port.

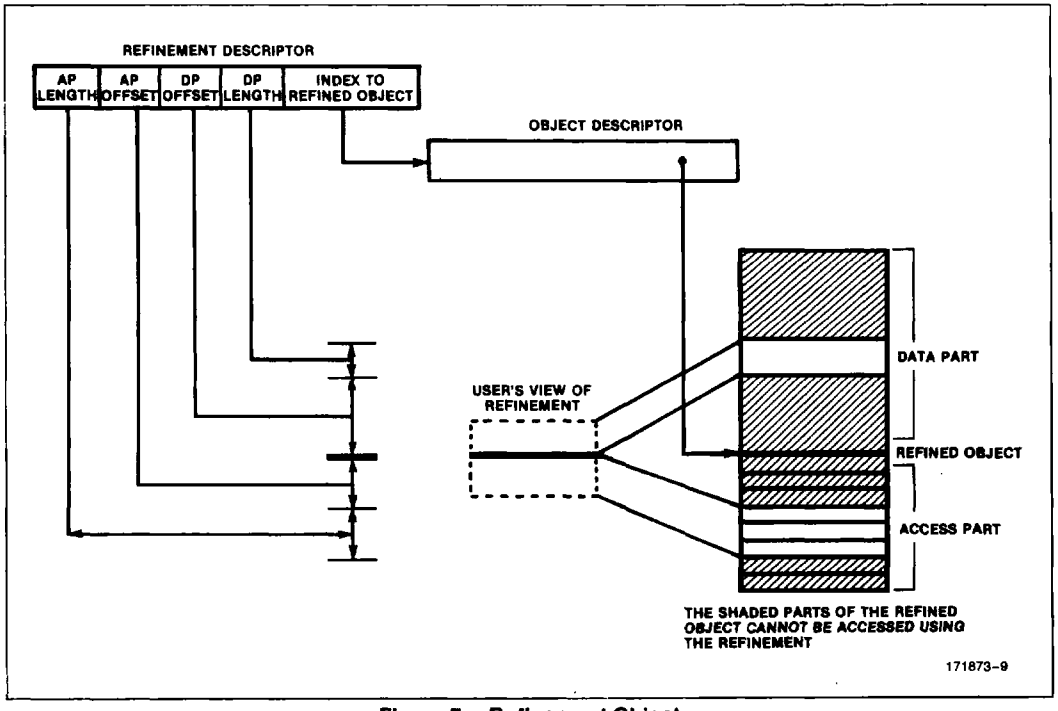


Figure 7. Refinement Object

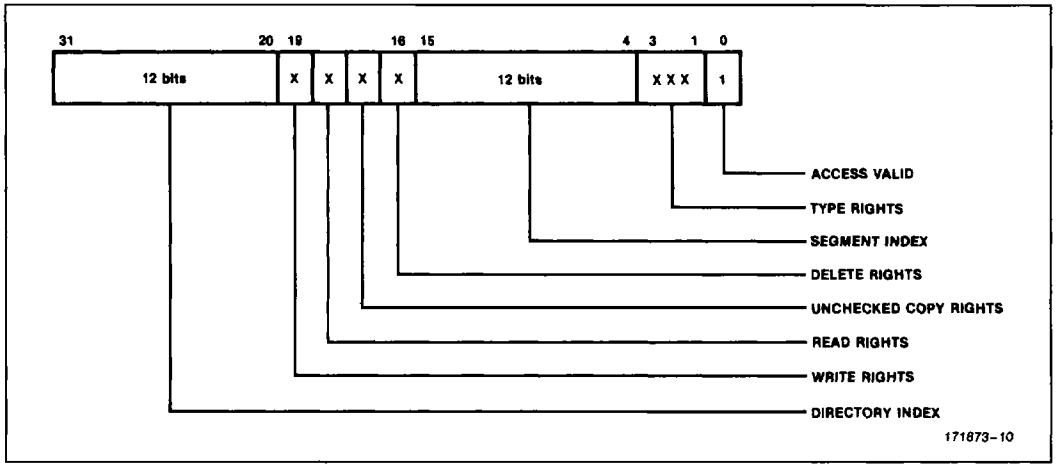


Figure 8. Access Descriptor Format

A type manager may define up to five rights for the dynamic-typed object it manages. It may selectively grant rights to a user when it creates an object and returns the reference for it. When a type manager receives a reference for an object, it may compare the rights in the reference to the operation requested by the user. The manager of a "bank account" object, for example, may permit all users to credit and debit bank accounts, but only a few may be granted the right to close an account.

Finally, the hardware performs a bounds check to insure that the displacement component of a logical address in fact falls within the target object. This is done by simply comparing the displacement to the object's length (contained in the object descriptor).

For improved performance, the hardware holds length and rights information for frequently-used objects, as well as their physical base addresses, and virtual memory control information, on chip.

Object Management

Reference Manipulation A procedure sometimes needs to manipulate the entries in its access descriptor list (as opposed to the objects the list references). The common base architecture defines operations that permits copying an access descriptor from one "slot" to another, and for nulling an access descriptor. This last operation is used to delete the reference to an object that is no longer needed. Another operation permits a procedure to inspect an access descriptor, for example, to examine the rights bits. A procedure may similarly inspect the object table entry indexed by an access descriptor to see, for example, if the object is dynamic-typed (i.e., the entry is for a type control object, rather than an object descriptor).

Object Creation and Deletion Free storage in central system memory is accounted for in system objects called **storage resource objects,** or SROs. SROs are lists of unallocated blocks of memory. Given a reference to an SRO, a procedure can create a new object dynamically; the instructions that create new objects automatically update the SRO from which storage is obtained.

Objects have "lifetimes": they come into being and occupy storage, and they also disappear, giving up their storage. The 432 common base architecture distinguished between short-term and long-term objects. A short-term object exists for the lifetime of the procedure that creates it; that is, it is allocated when the procedure is called and it is deallocated when the procedure returns. An operand stack, for example, is automatically created when a procedure is called, and a procedure may create an object to pass as a parameter to another procedure. A long-term object exists after its creating procedure returns; in fact, it lives indefinitely, until there are no object references left for it. A type manager "create" operation will create a long-term object.

As mentioned, short-term objects are deallocated automatically by the hardware when the creating procedure returns to its caller. Long-term objects are deallocated by a software routine called a **garbage collector.** This operating system routine sweeps through memory looking for objects that no longer have references left to them. When such an object is found, the garbage collector reclaims the storage occupied by the object by removing the object's descriptor from the object table and returning the storage block(s) occupied by the object to the SRO from which it was allocated.

Garbage collection is a complex operation; most conventional systems let garbage accumulate until a request for memory allocation cannot be satisfied. Then they halt normal execution, collect garbage, and resume operation. To avoid this suspension of service, the 432 architecture defines a **reclamation bit** in each object descriptor. By setting the reclamation bit whenever it copies an object reference, the hardware permits garbage to be collected on-the-fly, in parallel with normal execution. The operating system performs garbage collection automatically, relieving programmers of much of the burden of storage management.

Mutual Exclusion It is perfectly possible for two processes to hold references for the same object (see Figure 9, for example). If both processes only read the object they need not coordinate their operations. Consider, however, an object that accumulates the total number of transactions handled by several processes. Every so often each of these processes adds the number of transactions it has handled in the preceding time period to a field in this object. Although the addition is performed in a single machine instruction, at least three memory accesses are required to complete the operation: 1) read the old total; 2) read the increment; 3) write the new total. The integrity of the total is in jeopardy when two processes update it at nearly the same time.

The 432 has three classes of mutual exclusion mechanisms: I/O locks, object locks, and indivisible operations. Each object's storage descriptor contains a bit called the I/O lock. When an Interface Processor opens a window on the object, it checks to see that the object is not already I/O-locked, and then it locks it for the duration of the I/O transfer. Software running on a GDP may check for an I/O lock by inspecting an object's storage descriptor.

An **object lock** field may be defined for any data object. Processes that update such an object agree by convention not to update the object without first locking it, and further agree to unlock the object as soon as exclusion is no longer required. The LOCK OBJECT is conditional: it returns a value that indicates if the object was successfully locked (i.e., was not locked by another process). A process should refrain from accessing the object until it successfully locks it. In general, an object's type manager will take care of locking and unlocking the objects it manages, eliminating the need for object users to know anything about locks.

Note that a processor locks a 432 system object when it needs exclusive access to it during the execution of a high-level operation. This prevents another processor, or an executive routine, from interfering with a critical operation on the object.

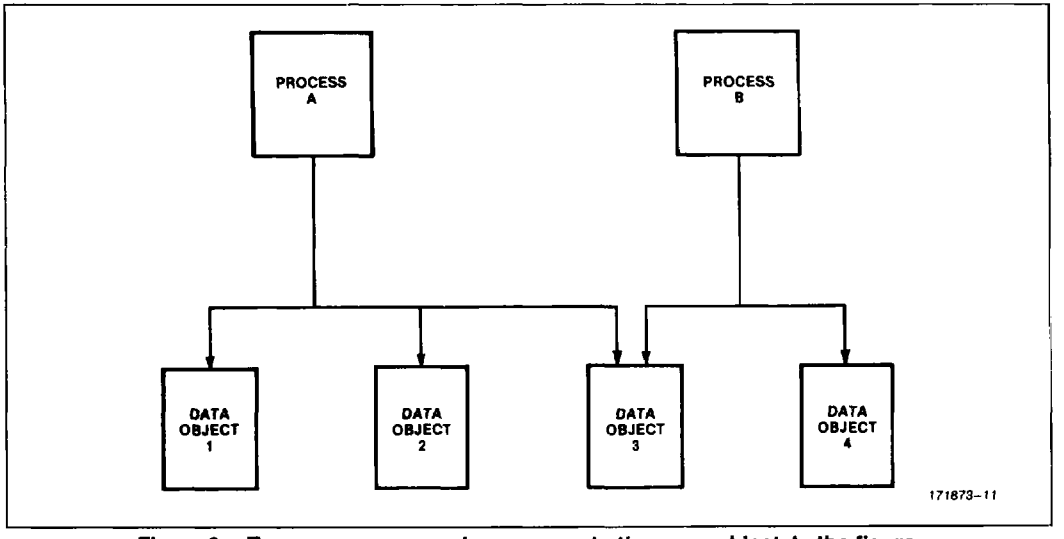


Figure 9. Two processes can share access to the same object. In the figure, both Processes A and B can access Data Object 3. In these cases, some mutual exclusion mechanism must be used to prevent inconsistent updating.

Often an object only needs to be locked for the duration of one instruction. The common base architecture defines operators that permit addition and bit field insertion (1 to 32 bits) to be performed **indivisibly.** When an indivisible operator is executed, the processor signals a read-modify-write bus cycle; the system memory controller must not permit a second RMW-write access to the target memory until the updated value has been written back into memory. Thus, the integrity of a shared value is guaranteed so long as programs that update it do so with indivisible operations.

While a refinement is effectively a new object that is a contiguous subset of an existing one, it actually requires only a new access descriptor, thereby saving storage and execution time. A process with a reference to a refinement has no knowledge of the "underlying" refined object. The process that created the refinement, however, can "retrieve" the refined object from a reference to the refinement.

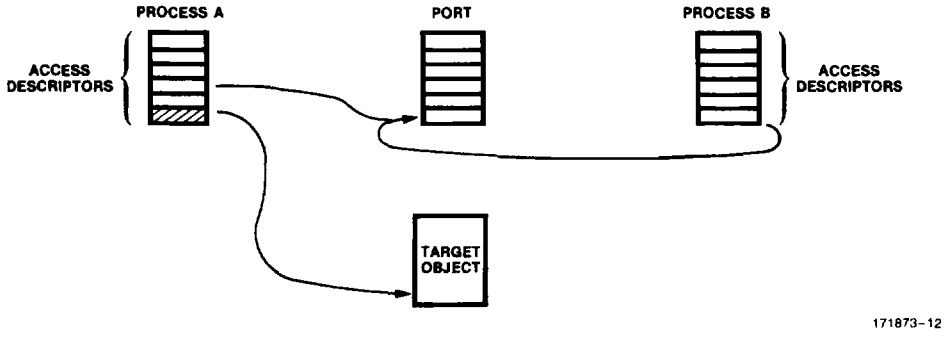
Process Communication

Except as they hold references to the same objects, 432 processes are completely independent of one another. Two processes may execute alternately on the same processor, or they may execute simultaneously on different processors. The 432 interprocess communication facility enables processes to communicate with each other by transmitting access descriptors (as messages) through memory during execution.

Since any object reference can be transmitted, process communication is an extremely efficient and versatile facility; it provides the basis for I/O operations and process dispatching in addition to more traditional message passing. Process communication can also be used to implement another form of mutual exclusion.

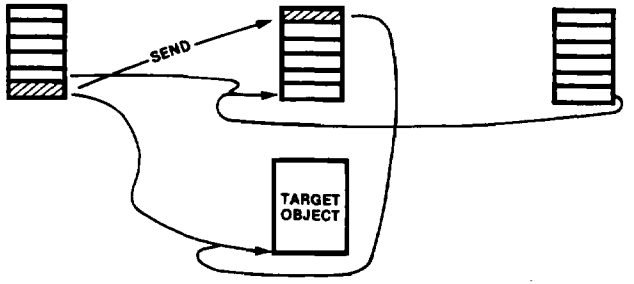
A complete transmission consists of a send and a corresponding receive. Since processes execute **asynchronously** with respect to each other, the time at which a process desires to send a message is unrelated to the time at which another process is ready to receive a message. Further, the rates at which processes send and receive are, for the most part, unpredictable. This is in contrast to the **synchronous** communication of procedures within the same process, which may pass object references as parameters in ordinary call and return operations. A call effectively suspends execution of the caller and starts execution of the called procedure; a return terminates the called procedure and resumes the caller. 432 interprocess communication, on the other hand, allows the communicating processes to run concurrently.

The 432 **port** object provides the synchronization and buffering needed for asynchronous process communication. Conceptually, a port is a queue; two processes with references to the same port have a channel over which they can communicate. A process wishing to transmit a message executes a SEND operator, which **copies** the access descriptor to the port (see Figure 10). A processes ready to obtain the message executes a RECEIVE operator, which moves the access descriptor at the head of the queue to the receiver's object reference list, thereby making the object accessible and deleting it from the queue.



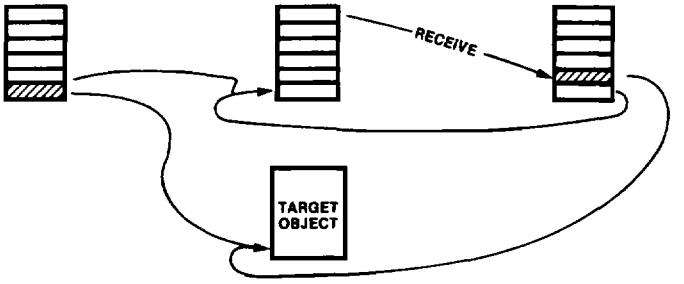
1) Both Process A and Process B have an access descriptor for the port, while A has an access descriptor for the target object.

Figure 10. Simple Message Transmission



171873-13

2) A sends message to port; an access descriptor for the target object is queued.



171873-14

3) B receives message from port; Processes A and B now have access to the target object.

Figure 10. Simple Message Transmission (Continued)

When a port is created, it is given a queuing discipline, which may be first-in-first-out (FIFO), or priority/deadline. A send to a FIFO port inserts the message at the back of the queue. A send to a priority/deadline port inserts the message according to priority and deadline parameters associated with it (and the messages already enqueued, if any). Insertion is done so that the highest priority messages are at the front of the queue; within a priority level, messages with the shortest (least) deadline are placed in front. A receive operation always takes the message at the front of the queue.

The simple SEND and RECEIVE operators are unconditional: they imply that the executing processes is willing to **block** if the port is full (or empty). A SEND to a full port or a RECEIVE at an empty port enqueues a reference to the sender's process object at the port; further execution of the process is blocked while it effectively waits at the port. When SENDs and RECEIVEs executed by other processes make it possible to complete the original operation, the process is automatically unblocked and execution may resume. A process waiting at a port due to a blocked RECEIVE, for example, is unblocked when another process SENDs a message to that port.

The CONDITIONAL SEND and CONDITIONAL RECEIVE operators, in contrast, never block: these operators return a value that indicates whether the operation was "successful." This signal gives the process the option of doing some other work and attempting the send or receive later. Other operators, called SURROGATE SEND and SURROGATE RECEIVE enable even more sophisticated forms of interprocess communication.

A port is also useful as a mutual exclusion mechanism. Processes that periodically require exclusive access to a shared object can designate a port to hold a single "key" to the object; the key can simply be a null access descriptor. When a process needs to obtain exclusive access, it does a receive operation on the port to pick up the key. (If the key is in use, the process blocks at the port.) When it actually receives the key, the process has exclusive access to the shared object. As soon as it has finished with the object, the process sends the key back to the port, making it available to other processes.

Interprocessor Communication

To coordinate the activities of multiple processors, all 432 processors can receive and respond to a set of predefined messages (see Table 4). For the most part, these messages are sent by processors on their own initiative. System software may also send a message to a particular processor, or may broadcast a message to all processors in the system.

When a system's memory manager moves an object or swaps it out of memory, for example, it will broadcast a "flush cache" message to all processors. This eliminates the possibility that a GDP will reference the object with an on-chip cache address which has been made invalid by the action of the memory manager. Sending a message to a processor requires a reference (with proper system rights) to that processor's process object; by controlling distribution of process object references, system software can likewise limit the ability to send processor messages.

Table 4 contains the Interprocessor Communication message codes in decimal along with a short description of the message:

Table 4. IPC Message Codes

0	Wakeup
1	Start
2	Stop
3	Accept Global IPCs
4	Ignore Global IPCs
5	Requalify Object Table Cache
6	Reset Processor
7	Requalify Processor
8	Requalify Process
9	Requalify Context
10	Requalify Data Object Cache
11	Enter Normal Mode
12	Enter Alarm Mode
13	Enter Reconfiguration Mode
14	Enter Diagnostic Mode

Timekeeping

The central system runs on a single time standard which is reflected in the on-chip clock of every 432 processor. The on-chip clock is a 16-bit accumulator which is driven by the PCLK signal. All 432 processors are tied to PCLK, which is independent of the component (CLK_A, CLK_B) clocks. When PCLK is asserted by an external timing source, all processor clocks increment ("tick") in unison.

PCLK's frequency, at thus the duration of one system unit, is set according to the resolution required by the application; 200 microseconds is a typical value. As discussed later, the system unit provides the basis for GDP processing and dispatching.

Each GDP process object contains a clock field that is automatically maintained by the hardware. A process clock indicates the number of system time units that the process has been bound to a processor (i.e, how long it has been executing). Using a 200 microsecond clock, a process clock can accumulate over 236 hours before turning over. Process clocks can be read by software and provide the basis for execution-time charging algorithms and for adaptive process scheduling.

Exceptions

Software Faults Most modern computers provide a facility for detecting errors during execution; for example, many CPUs detect arithmetic overflow or an application program's attempt to execute a "privileged" instruction. The 432 extends this concept into a comprehensive, structured software **fault** system (see Table 5).

A software fault is an exceptional condition uncovered by a processor during execution. It may be a simple computational error (e.g., square root of a negative number), an attempted protection violation, or a condition that requires off-line handling though not an error or violation. Whatever the source, the architecture recognizes that normal computation cannot continue until the exceptional condition is resolved.

The architecture defines software fault **detection** and fault **reporting;** that is, the notification that a fault has occurred and provision of information describing it. Software fault **handling,** which may include fault **recovery** in many cases, is the province of application or—more frequently—system software. A software fault may be detected at any time: during the execution of an instruction or command, while a processor is performing an operation on its own initiative, during an IP data transfer, and so forth. A processor reports a software fault by first recording descriptive information in the predefined **fault information area** of a system object. This information describes the nature of the fault and provide additional information that may assist software in recovering from it. The fault handler examines the fault information and takes "appropriate action," as defined by the application. This may vary considerably according to the application and the nature of the fault.

Table 5. Detectable Software Faults

General Fault Groups

Memory Reference Faults:
- Segment Overflow Fault
- Memory Overflow Fault
- Read Rights Fault
- Write Rights Fault
- Bus Error

Instruction Fetch Fault

Data Part Cache Qualification Faults
Data Part Access Faults:
- Access Descriptor Validity Fault
- Object Descriptor Type Fault

Object Table Qualification Faults:
- Object Descriptor Type Fault
- Object Type Fault

Access Environment Altered Faults:
- Access Descriptor Validity Fault
- Object Descriptor Fault

Data Operator Fault Groups

Domain Error Fault
Overflow Fault
Underflow Fault
Inexact Fault

Non-Instruction Interface Faults

Initialization:
- Object Qualification Faults (Processor)
- Object Qualification Faults (Object Table Directory)

IPC Faults
- Object Qualification Faults (PCO)
- PCO Response Count Fault
- PCO Lock Fault

Idle:
- Delay Port Service Faults

Process Binding:
- Object Qualification Faults (Carrier)
- Process Lock Faults
- Process Qualification Faults
- Port Operation Faults

Process Selection:
- Delay Port Service Faults
- Object Qualification Faults
- Port Operation Faults

Object Operator Faults

Branch
Branch True
Branch False:
- Instruction Pointer Overflow Fault
- Instruction Object Displacement Fault

Branch Indirect:
- Instruction Object Displacement Fault

Branch Intersegment
Branch Intersegment without Trace
Branch Intersegment and Link:
- Object Qualification Faults (Instructions)
- Instruction Object Displacement Fault

Copy Access Descriptor:
- Store Access Descriptor Faults

Null Access Descriptor:
- Destination Delete Rights Fault

Amplify Rights:
- TCO Type Rights Fault
- Object Qualification Faults (TCO)
- Type Fault
- Race Condition Fault

Retrieve Type Definition:
- Source AD Validity Faults
- Store Access Descriptor Faults

Create Refinement:
- Source AD Validity Fault
- Object Descriptor Type Fault
- Offset and Length Compatibility Fault
- Refinement Overflow Fault
- Level Fault

Create Typed Refinement:
- TCO Type Rights Fault
- Source AD Validity Fault
- Object Descriptor Type Fault
- Type Fault
- Offset and Length Compatibility Fault
- Refinement Overflow Fault
- Level Fault

Create Typed Object:
- Descriptor Allocation Faults
- Object Qualification Faults (TCO)
- TCO Type Rights Fault
- Level Fault
- Segment Allocation Faults
- Store Access Descriptor Faults

Table 5. Detectable Software Faults (Continued)

Inspect Object:
- Access Path Object Descriptor Fault

Lock Object:
- Source Representation Rights Fault

Unlock Object:
- Source Representation Rights Fault
- Object Lock ID/Type Fault

Call

Call through Domain:
- Object Qualification Faults (Domain)
- Domain Access Index Overflow Fault
- Instruction Object Type Rights Fault
- Object Qualification Faults (Instructions)
- Context Parameters Size Fault
- Context Type Rights Fault
- Object Qualification Faults (Context)
- Instruction Object Displacement Fault

Return:
- Context Type Rights Fault
- Context Qualification Faults
- Object Qualification Faults (PSO)
- Object Qualification Faults (Object Table)
- PSO Lock Fault
- Instruction Object Displacement Fault

Block Move:
- Offset Overflow

Return and Fault:
- Return Fault

Send

Receive

Conditional Send

Conditional Receive

Delay Process

Send Process:
- Port Type Rights Fault
- Level Fault

Surrogate Send

Surrogate Receive:
- Surrogate Carrier Validity Fault
- Surrogate Carrier Type Rights Fault
- Destination Port Type Rights Fault
- Port Type Rights Fault
- Level Fault

Set Process Mode:
- Process Object Type Rights Fault
- Process Object Access Mismatch Fault

Send to Processor:
- PCO Type Rights Fault
- Object Qualification Faults (PCO)

Move to Interconnect

Move from Interconnect:
- Odd Displacement Fault
- Odd Interconnect Descriptor Base Address Fault
- Object Qualification Faults (Interconnect)

The common base architecture recognizes that some faults are more serious than others; indeed certain faults, such as "not allocated" (i.e., an object needs to be swapped in from external storage) will be routine in many systems. Accordingly, software faults are divided into levels based on their impact on the system and the amount of information required to resolve them. This permits software to provide a response that is appropriate to the severity of the problem and to minimize the disruption that handling the fault may have on the rest of the system.

In general, the philosophy is to keep the unaffected parts of the system running while the fault is handled outside the normal flow of execution. Processors record fault information in the system object that corresponds to the fault-level; for example, information describing a process-level fault is recorded in the process object.

A context is a single instance of a procedure in execution. A **context-level** fault is one that can normally be handled within the process (i.e., by application code). Ada, for example, permits programmers to write exception handlers that will respond to GDP context-level faults.

A **process-level** fault prevents the current process from continuing until the fault is handled, but does not affect other processes. GDPs and IPs respond to this situation similarly. Generally, the procedure is to remove the offending process from the set of active processes by sending its process object to a **fault** port and then dispatching the next ready process.

Each process is associated with a fault port; a fault port is an ordinary port that queues messages that happen to be references to "broken" processes. An operating system **fault process** can receive these process objects and attempt to "repair" them, that is, recover from the fault. For example, if the fault process determines that the fault is "object not allocated," it can notify the system's virtual memory manager to swap in the needed object. When this has been done, the fault process can send the process off to its dispatching port (see "Scheduling and Dispatching").

A **processor-level** fault threatens (but might not absolutely prevent) continued execution by the processor. The GDP and IP respond to this situation differently, but the basic procedure is to run a processor diagnostic program.

At the final level, the processor cannot do anything, not even record fault information. It therefore halts and asserts its FATAL pin. Software on some other processors might monitor this pin; for example, it could be routed to an interconnect register and periodically sampled. A halted processor can be restarted by hardware (asserting its INIT pin) or software (sending a START IPC).

Interrupts. Each 432 processor has an ALARM pin which can be asserted to signal the occurence of an extremely high priority external event. A typical example is imminent power failure. In general, when ALARM has been asserted, the GDP will complete its current instruction and then invoke a designated software process (waiting at the ALARM port).

Data Types

The memory formats of the GDP's eight basic data types are illustrated in Figure 11. Any data type may be stored on any byte boundary (performance is improved, however, when data is aligned on physical memory boundaries). The types are divided into four classes: character, ordinal (unsigned integer), integer, and real. These data types correspond directly to the "primitive" types defined in most high level languages. Implementing the essential types in hardware, with a choice of storage requirements for each class, helps ensure that compiler-generated code sequences are both compact and fast.

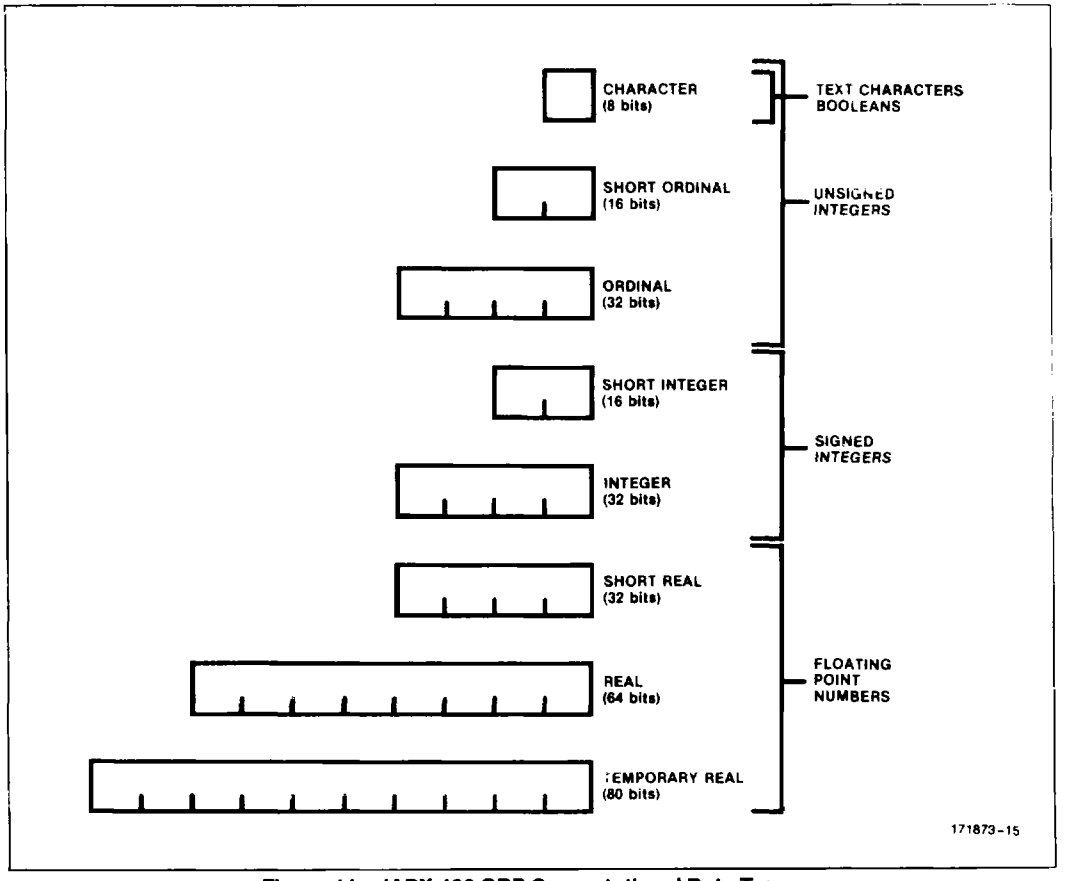


Figure 11. iAPX 432 GDP Computational Data Types

Table 6 gives the attributes of the numeric data types. Of particular note is the temporary-real data type. The extra range and precision of this type contributes to the production of consistently safe, reliable floating-point algorithms. As its name implies, the temporary-real type is intended for holding intermediate computational results. The inputs and outputs of a calculation should be defined as short-real or real, according to the range and accuracy of the available data. All intermediate results should be held in temporary-real, with the final conversion to the output at the end of the computation.

Extremely large and small values are most likely to occur in intermediate computations; using temporary-real for these makes overflow and underflow exceedingly unlikely in most applications. Temporary-real's extended precision also prevents round-off errors from accumulating during long computations; the only significant round-off occurs at the conversion from temporary-real to the real or short-real format of the final result.

Instruction Set

Table 7 shows the instructions the GDP provides for its data types. The symmetry of the instruction set with respect to data types simplifies compiler code generation.

In the data transfer group, the ZERO and ONE instructions write a constant into an operand. MOVE copies a variable, popping the stack if it is the source, pushing it if it is the destination. SAVE copies the stack top without popping the stack; it can be used to duplicate the stack top.

The logical instructions perform the customary operations. XNOR is the complement of XOR (exclusive OR): where XOR returns 1-bits when corresponding bits are unequal, XNOR returns 1-bits when corresponding bits are equal. It is thus the Boolean equivalent of "equals."

In the arithmetic group, the REMAINDER instruction performs **exact** modulo division; it is very useful for reducing an argument to a periodic transcendental function (e.g. tangent) to the range accepted by the function without introducing round-off error. SQUARE ROOT executes in about the same time as ordinary division; programmers need not contort algorithms to eliminate time-consuming square roots.

Table 6. Numeric Data Types

Data Type	Significant Digits*	Approximate Range
Character	2	$0 \le x \le 255$
Short-Ordinal	4	$0 \le x \le 65{,}535$
Ordinal	9	$0 \le x \le 4{,}294{,}967{,}295$
Short-Integer	4	$-32{,}768 \le x \le 32{,}767$
Integer	9	$-2{,}147{,}483{,}648 \le x \le 2{,}147{,}483{,}647$
Short-Real	6–7	$8.43 \times 10^{-37} \le \lvert x \rvert \le 3.37 \times 10^{38}$
Real	15–17	$4.19 \times 10^{-307} \le \lvert x \rvert \le 1.67 \times 10^{308}$
Temporary-Real	19	$3.4 \times 10^{-4932} \le \lvert x \rvert \le 1.2 \times 10^{4932}$

* Decimal equivalent

Table 7. iAPX 432 Operators and Computational Data Types

	Operators	Data Types							
		Char.	Short Ordinal	Ordinal	Short Integer	Integer	Short Real	Real	Temp. Real
MOVE OPERATORS	MOVE	X	X	X	X	X	X	X	X
	SAVE	X	X	X	X	X	X	X	X
	ZERO	X	X	X	X	X	X	X	X
	ONE	X	X	X	X	X	–	–	–
LOGICAL OPERATORS	AND	X	X	X	–	–	–	–	–
	INCLUSIVE OR	X	X	X	–	–	–	–	–
	EXCLUSIVE OR	X	X	X	–	–	–	–	–
	EQUIVALENCE	X	X	X	–	–	–	–	–
	NOT	X	X	X	–	–	–	–	–
ARITHMETIC OPERATORS *	ADD	X	X	X	X	X	*	*	X
	SUBTRACT	X	X	X	X	X	*	*	X
	MULTIPLY		X	X	X	X	*	*	X
	DIVIDE		X	X	X	X	*	*	X
	REMAINDER		X	X	X	X	–	–	X
	INCREMENT	X	X	X	X	X	–	–	–
	DECREMENT	X	X	X	X	X	–	–	–
	NEGATE	–	–	–	X	X	X	X	X
	ABSOLUTE VALUE	–	–	–			X	X	X
	SQUARE ROOT								X
	INDEX	–	–	X	–		–	–	–
BIT-FIELD INSERT	EXTRACT		X	X	–	–	–	–	–
	INSERT		X	X	–	–	–	–	–
	SIGNIFICANT BIT		X	X	–	–	–	–	–
RELATIONAL OPERATORS	EQUAL	X	X	X	X	X	X	X	X
	NOT EQUAL	X	X	X	X	X			
	EQUAL ZERO	X	X	X	X	X	X	X	X
	NOT EQUAL ZERO	X	X	X	X	X			
	LESS THAN	X	X	X	X	X	X	X	X
	LESS THAN OR EQUAL	X	X	X	X	X	X	X	X
	POSITIVE	–	–	–	X	X	X	X	X
	NEGATIVE	–	–	–	X	X	X	X	X
	MOVE IN RANGE				X	X			
CONVERSION OPERATORS	TO CHARACTER	–				X			
	TO SHORT ORDINAL	X	–			X			
	TO ORDINAL			–		X			X
	TO SHORT INTEGER				–	X			
	TO INTEGER	X	X	X	X	–			X
	TO SHORT REAL						–		X
	TO REAL							–	X
	TO TEMPORARY REAL		X	X	X	X	X	X	–

WHERE:
X — Means the operator is available for the given data type.
* — Means the operator is available for the given data type and for instructions in which one of the operands is a temporary real.
– — Means the operator is not available and would be of little or no use if it were.
(BLANK) — Means the operator is not available.

The bit field instructions, EXTRACT and INSERT BIT FIELD, make the manipulation of packed bit field records simple and rapid. The SIGNIFICANT BIT instruction returns the position of the "leftmost" 1-bit in an ordinal or short-ordinal. Note that the INSERT BIT FIELD is an indivisible operation; once the instruction starts to run, no other processor can perform an indivisible operation on the field until its new value has been written into memory.

The instructions in the comparison group assert a condition existing in a single variable (e.g., EQUAL ZERO) or between two variables (e.g., GREATER THAN). These instructions return a Boolean value TRUE or FALSE according to the truth of the assertion. Conditional branching is effected by following a comparison with BRANCH TRUE or BRANCH FALSE instruction.

The GDP has the full complement of 432 common base instructions plus addition data processing operations. Some of these permit changing the flow of control in a program by conditional and unconditional, and by calling a procedure. Others facilitate the manipulation of composite objects (objects made up of other objects), access to data declared global to all procedures in a process, and setting precision and rounding modes for real number computations. Finally, two of the instructions give a GDP program access to the interconnect space.

Instruction Formats

The 432's instruction codes have been designed to minimize the space the instructions occupy in memory and still allow for efficient encoding. In order to achieve the best efficiency in storage, the instructions are encoded without regard for byte, word, or other artificial boundaries. The instructions may be viewed as a linear sequence of bits in memory, with each instruction occupying exactly the number of bits required for its complete specification.

Processors view these instructions as composed of fields of varying numbers of bits that are organized to present information to the Instruction Decoder in the sequence required for decoding. A unified form for all instructions allows instruction decoding of all instructions to proceed in the same manner.

In general, GDP instructions consist of four main fields. These fields are called the class field, the format field, the reference field, and the opcode field. The reference field, in turn, may contain several other fields, depending upon the number and the complexity of the operand references in the instruction. The fields of a GDP instruction are stored in memory in the following format.

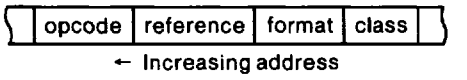
The class field is either 4-or 6-bits long, depending on its encoding. The class field specifies the number of operands required by the instruction and the primitive types of the operands. The class field may indicate 0, 1, 2, or 3 operands. If the class field indicates one or more references, a format field is required to specify whether the references are implicit or explicit and their uses.

In the case of explicit references, the format field can indicate whether or not the reference is direct or indirect. Further, the format field may indicate that a single operand plays more than one role in the execution of the instruction. As an example, consider an instruction to increment the value of an integer in memory. The instruction begins with a class field specifying that the operator is of order two and that the two operands occupy a word of storage; next, the format field indicates that a single reference specifies a logical address to be used both for fetching the source operand and for storing the result; it is followed by an explicit data reference to the integer to be incremented; and finally the instruction ends with an opcode field for the order-two operator INCREMENT INTEGER.

It is possible for a format field to indicate that an instruction contains fewer explicit data references than are indicated by the instruction's class field. In this case, the other data references are implicit, and the corresponding source or result operands are obtained from (or returned to) the top of the operand stack. Consider the following statement:

$$A = A + B*C$$

The instruction fragment for this statement consists of two instructions and has the following form:



171873-16

Assume that A, B, and C are integer operands. The first class field (the rightmost field shown above) specifies that the operator requires three references and that all three references are to word operands.

The first format field contains a code specifying two explicit data references supplying only two source operands. The destination is referenced implicitly so that the result of the multiplication is pushed on the operand stack. The second class field is identical to the first and specifies three required references by the operator, all to word operands. The second format field specifies one explicit data reference to be used for both the first source operand and the destination. The second source operand is referenced implicitly and popped from the operand stack when the instruction is executed.

The reference fields themselves can be of various lengths and can appear in varying numbers (consistent, of course, with the specifications in the class and format fields. If implicit references are specified, reference fields for them will not appear. Direct references will require more bits to specify than indirect references.

Following the class, format, and reference fields, the opcode field appears. The opcode field specifies the operator to be applied to the operands specified in the preceding fields.

Addressing Modes

The operands (data items) that a GDP instruction is to operate on are encoded in the instruction as data references consisting of two parts: a **base** part and an **Index** part. The entire data reference can therefore be viewed as having three components: an access selection component, which selects an object; a base part of the operand offset, which provides a byte displacement to the base of the area of memory within the selected object; and an index part of the operand offset, which specifies a particular operand within that area.

The addressing is very flexible since each part of the operand offset can be specified directly or indirectly. A direct base or direct index has its value specified directly in the data reference encoding. When indirection is used, however, the value of the base or index is given by a short-ordinal value located within the currently accessible object.

There are four possible combinations of direct and indirect base and index parts, and each combination results in a different mode of reference (see Figure 12). Each of the four combinations has been used to name a data reference mode indicating the kind of data structure for which the reference would usually be used. The scalar, record, static array, and dynamic array modes correspond roughly to the direct, base, indexed, and base-plus-index modes found in many computers; all four modes are independently available for any operand specified in an instruction.

As shown in Figure 13, the displacement component may be encoded directly in the instruction or may come from base and index variables in memory (including the stack), or may consist of one direct and one indirect value. Choosing between direct and indirect specifications primarily depends on whether information is fixed at compile-time and what may be computed during execution.

Note that an indirect index value (used to select an array element) is expressed naturally as the element number to be accessed. The hardware automatically **scales** the index according to the data type being manipulated by the instruction to calculate the actual byte displacement. For example, to address the third element of a vector, the indirect index variable would contain the value 3 for any type of vector—character, integer, real, etc.

A fifth addressing mode is implicitly specified when a data reference is expected (according to the "number of references" field), but none is encoded in the instruction. The data reference in this case is the operand on top of the stack. If the operand is the source, it is automatically popped from the stack; if the operand is the destination, the result of the operation is pushed onto the stack.

Large Array Indexing

The maximum size of the data part of an object is 65,636 (64K) bytes, but of course, some applications require arrays that are larger. The INDEX ORDINAL operator is used to access these large arrays.

The large array is mapped (at compile-time) into a series of objects, each with data parts that are 2,048 bytes (2K) long. All these objects are directly accessible in the current logical access environment. The INDEX ORDINAL operator works as follows:

Given:

- The size of each element in the array (i.e., a scale factor)
- The access selector for the base segment of the array
- The ordinal index for the desired array element

The operator computes:

- The access selector for the appropriate 2K data object that contains the indexed array element
- The displacement into the data part of that object in the array element

The resulting short-ordinal values can then be used with the indirect access selection mode and the record, static array, or dynamic array data reference modes to access the array element. Of course, this whole process is invisible to the typical 432 programmer who uses a high-level language and leaves the choice of machine instructions to the compiler.

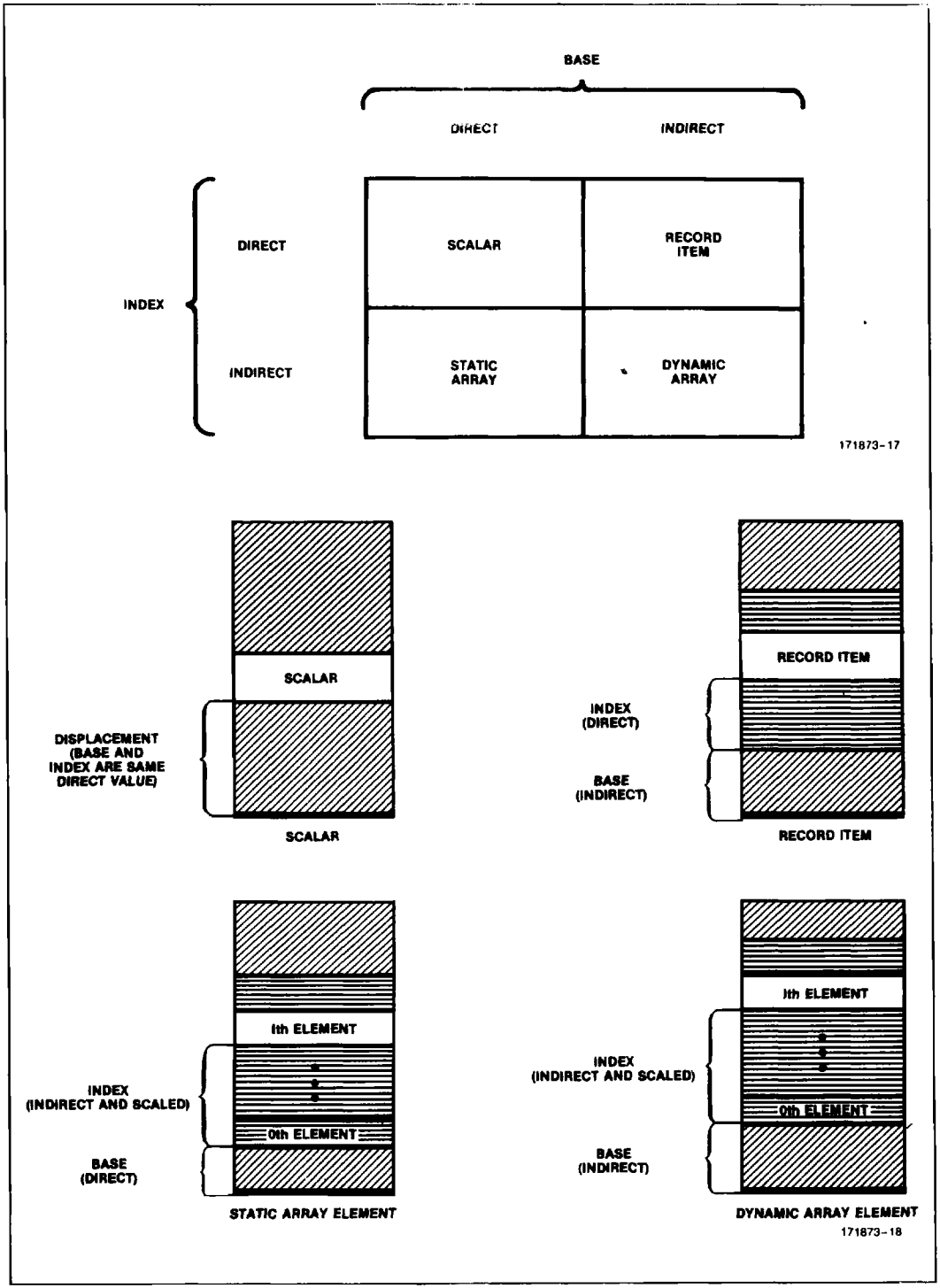
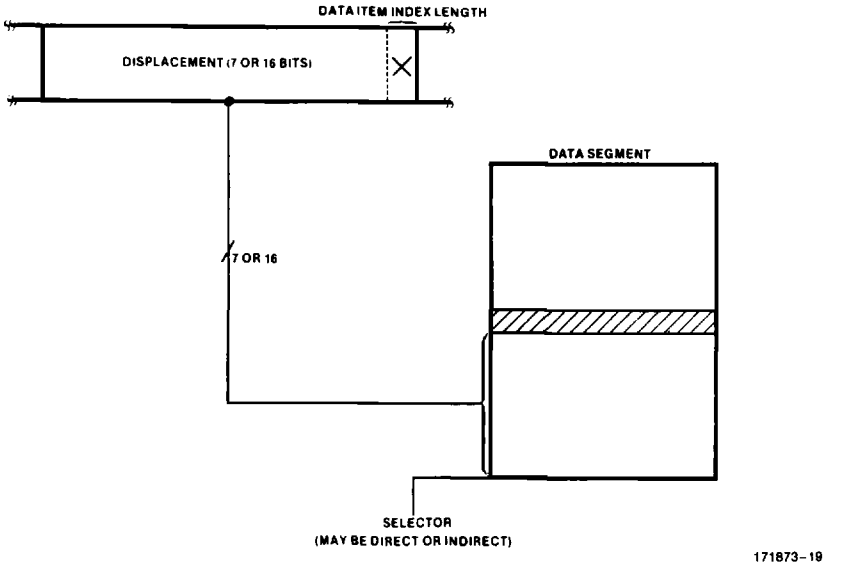
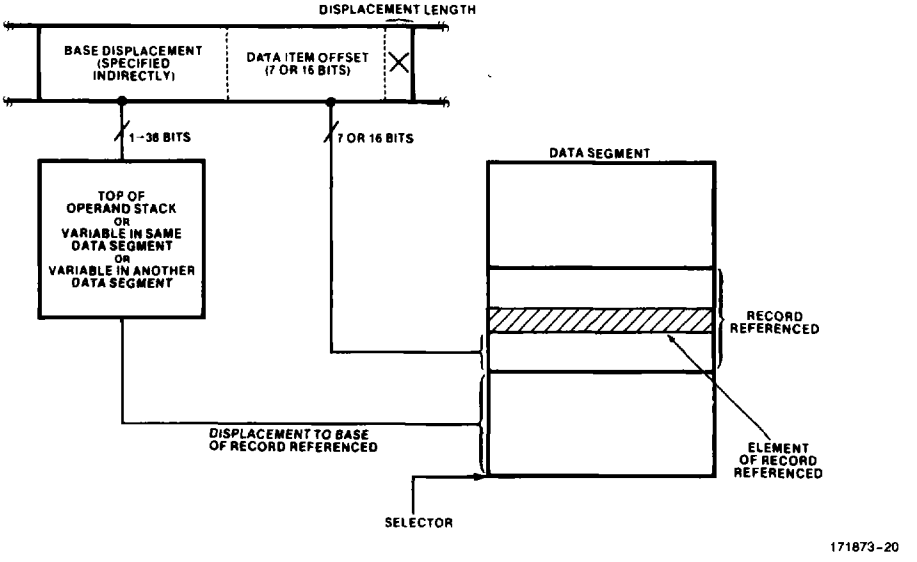


Figure 12. Addressing Modes

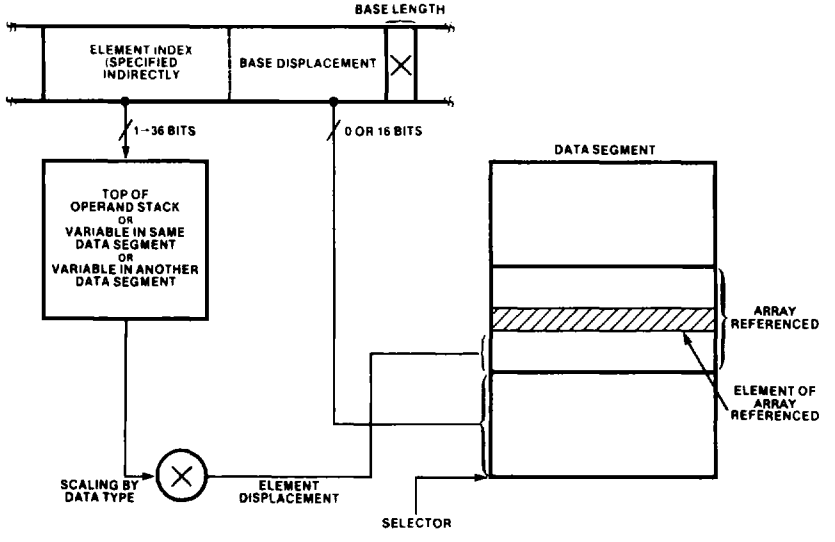


A. SCALAR DATA REFERENCE MODE

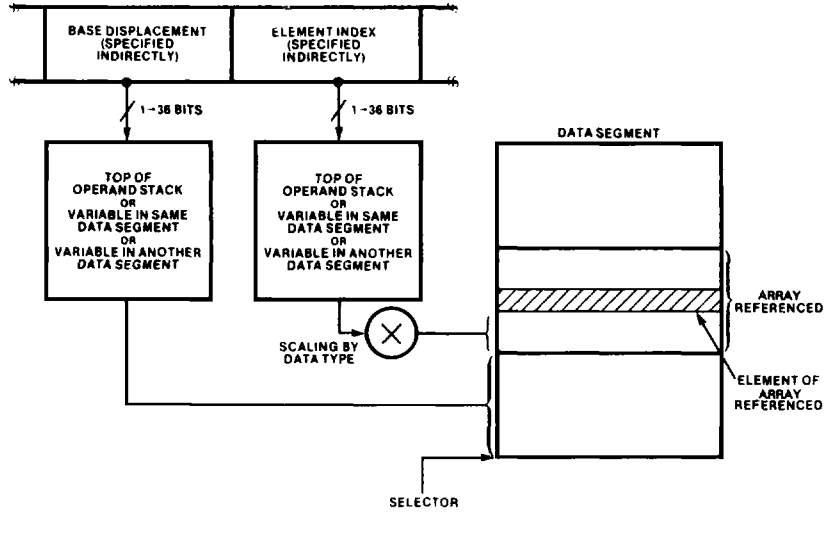


B. RECORD ITEM REFERENCE MODE

Figure 13. Modes of Displacement Generation



C. STATIC ARRAY ELEMENT REFERENCE MODE



D. DYNAMIC ARRAY ELEMENT REFERENCE MODE

Figure 13. Modes of Displacement Generation (Continued)

Scheduling and Dispatching

In most systems there will be more processes to run than there are processors. The procedure by which processes "take turns" running on GDPs is called dispatching and scheduling. Each processor is assigned to a **dispatching port,** from which it obtains its work, that is, the processes it executes. A dispatching port is an ordinary port object; it so happens that the access descriptors queued there are for process objects and processor objects. The assignment of processors to dispatching ports is defined by the application; usually all processors share one port, but each may have its own, or a processor's dispatching port may be changed by operating system software during execution.

Scheduling and dispatching are performed in two loops as shown in Figure 14. To maximize processor utilization, low-level scheduling and dispatching are performed automatically by the processor with no software intervention. Every process has four scheduling parameters; these are initially set by the operating system when a process is created. The parameters are.

1) **priority,** the relative urgency of the process;
2) **deadline,** the amount of time that may pass before the process must have a turn on the processor;
3) **service period,** the duration of one turn;
4) **period count,** the number of turns the process should be given before examining its scheduling parameters.

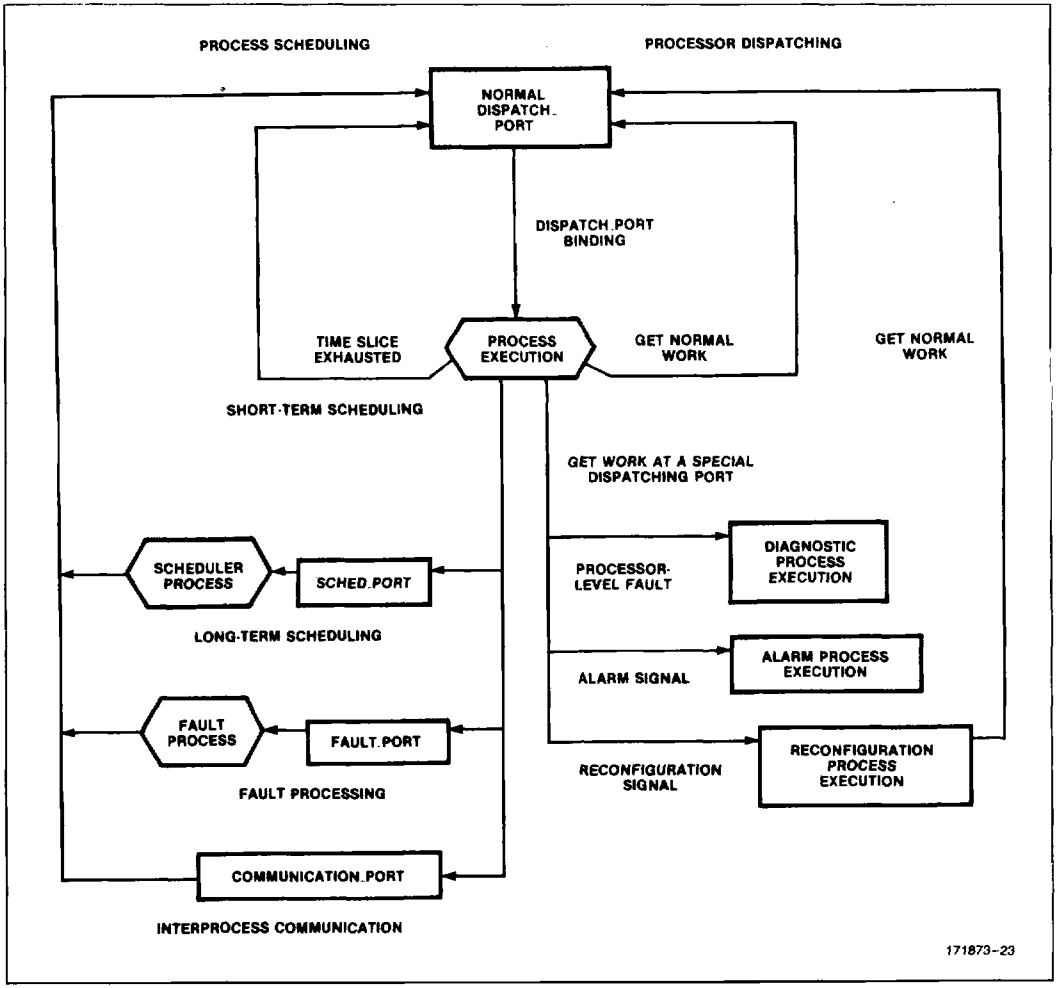


Figure 14. Process scheduling and processor dispatching. The left half of the diagram describes possible states of a process, while the right half describes possible states of a processor.

All time values are based on the system time unit. These parameters give operating system software great flexibility in setting system scheduling policy (or policies), and even altering a policy during execution.

Like the messages at any priority/deadline port, processes waiting for service at a dispatching port are ordered by deadline within priority; the highest-priority-least-deadline process is at the front of its queue. A GDP dispatching operation consists simply of "receiving" this process. The processor loads its on-chip service timer with the process's service period value and runs the processor for one service period. (Assertion of the PCLK pin increments the processor clock and decrements the service period as well.) At the end of the service period, the GDP decrements the process's period count, updates the process's process clock with the number of time units given to it, and schedules the process for another turn. If the period count has not yet expired, this is done by sending the process to its dispatching port. The send operation inserts the process into the queue according to its scheduling parameters.

If the process blocks before its period expires (before the service timer goes to zero), the period count and the process clock are also updated (with the number of actual units received), but the process is sent to a communication port instead of a dispatching port.

High-level scheduling is performed by the operating system; it gives the executive the opportunity to examine the system's performance and perhaps adjust its scheduling algorithms. When a process has exhausted all its service periods, the processor sends the process to a **scheduling port** instead of a dispatching port. The operating system scheduler receives the process, sets its scheduling and service parameters again, and sends the process back to the dispatching port, where the low-level cycle begins again.

When a GDP attempts to dispatch a process and none is available, the processor queues itself (that is, its processor object) at the dispatching port and "sleeps" until a process arrives. (A sleeping processor is almost completely idle; in a multiprocessor configuration this helps to reduce contention for use of the memory bus.) The processor that sends a process to the dispatching port also "wakes up" the sleeping processor (by means of an IPC); the awakened processor then dispatches the newly-arrived process. Operating system software may periodically check dispatching ports for idle processors and reassign them to dispatching ports that are more heavily loaded.

Designing Fault-Tolerant Systems

When used together, the five components in the iAPX 432 family provide all the logic necessary to build a system that will tolerate the failure of any single component or bus, yet continue to execute programs without error and without interruption. No software intervention is required: fault detection, isolation, and reconfiguration of the system is performed entirely by the hardware.

Each GDP is able to detect hardware errors automatically because of a capability known as Functional Redundancy Checking (FRC), so called because a second or redundant GDP checks the operations of the first or master GDP. Functional Redundancy Checking provides the low-level hardware support upon which hardware fault-tolerant modules are constructed.

During initialization, each GDP is assigned to operate as either a master or a checker (see Figure 15). While a master operates in a conventional manner, a checker places all output pins that are being checked into a high-impedance state. Those pins which are to be checked on a master and checker are parallel-connected, pin for pin, such that the checker is able to compare its master's output pin values with its own. If on any cycle, the values differ, the checker asserts HERR and the faulty components can be immediately disabled. Thus, any hardware errors can be detected as they occur and before they have had the opportunity to corrupt the operation of other components in the system.

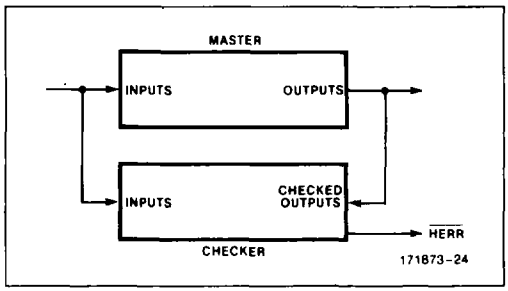


Figure 15. Function redundancy checking detects hardware errors automatically.

While FRC can be used alone to provide automatic error detection, a completely fault-tolerant system must also be able to reconfigure itself, replacing the set of failed components with another pair that is still working. In order to do so, the 432's architecture enables two pairs of master/checker components to be combined to form primary and shadow processors in a configuration known as Quad Modular Redundancy (QMR). See Figure 16.

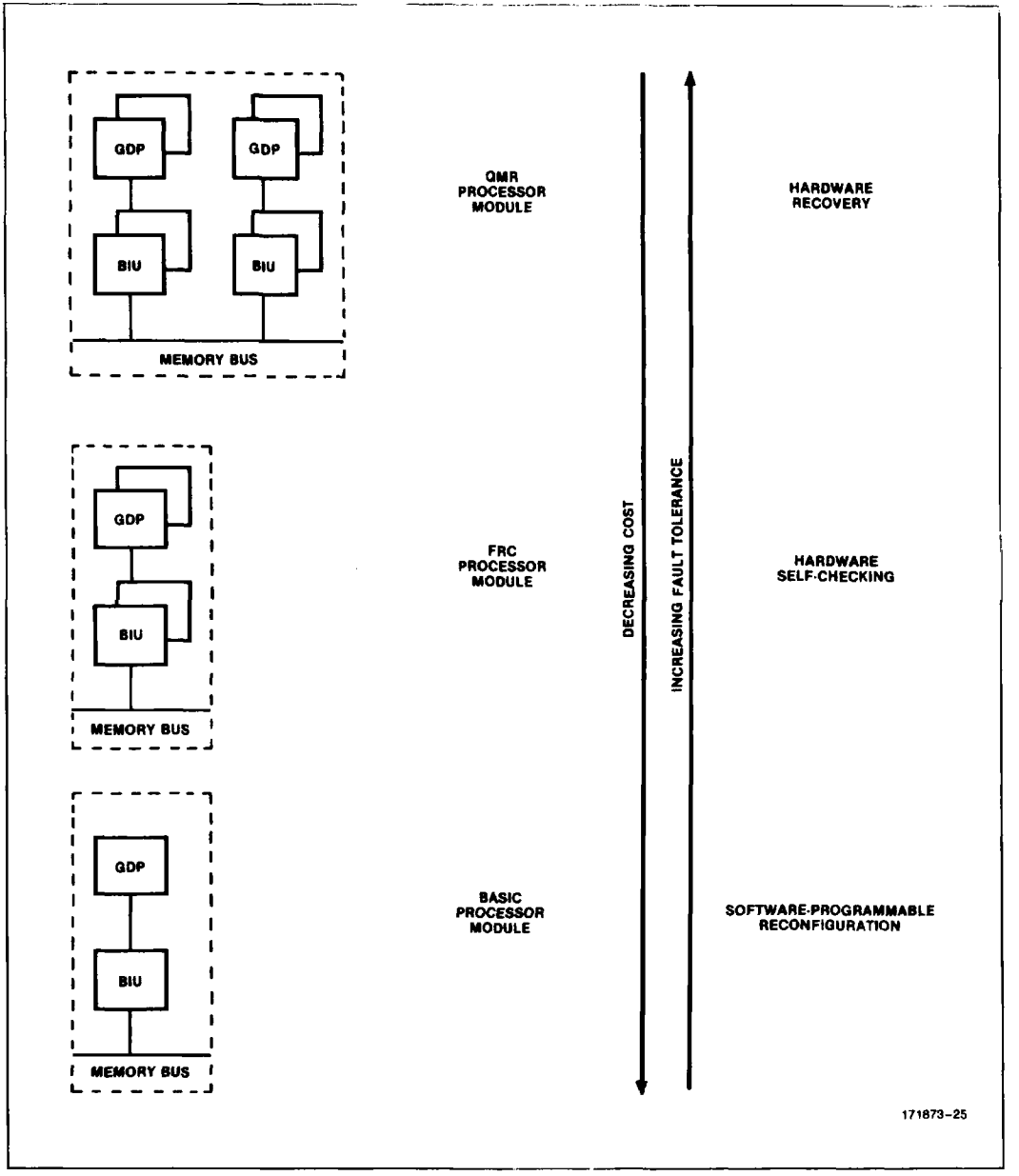


Figure 16. Fault Tolerant Alternatives

Every module in a QMR system is paired with another self-checking module of the same type. The pair of self-checking modules operates in lock step and provides a complete and current backup for all state information in the module. The mechanism is known as module shadowing because a shadow is ready to fill in if the primary fails (or vice versa). Fault detection and recovery occurs transparently to both application and system software. When a fault is detected, the faulty pair is automatically disabled, and the remaining pair takes over. Only then is system software notified that a failure has occured.

A more complete discussion of the fault-tolerant capabilities of the iAPX 432 can be found in the **iAPX 43204-iAPX 43205 Fault Tolerant Bus Interface and Memory Control Units** data sheet (Order Number 210963).

HARDWARE IMPLEMENTATION

The iAPX 432 General Data Processor is organized as a three-stage microprogram-controlled pipeline. The first stage is the Instruction Decoder, the second the Microinstruction Sequencer, and the third the Execution Unit. The first two stages of the pipeline are physically located on the iAPX 43201 with the third stage on the iAPX 43202. Each stage, however, can be considered an independent subprocessor that operates until the pipeline is full, and then halts and waits for more work to do.

Instruction Decoder

The general task facing the Instruction Decoder (see Figure 17) is to interpret the macroinstruction stream both to extract logical addresses and to determine the next microinstruction sequence to be initiated. In doing so, it performs the following functions:

- Receives macroinstructions
- Processes variable length fields
- Extracts logical addresses
- Generates starting addresses for the micro-instruction procedures
- Generates microinstructions for simple operations

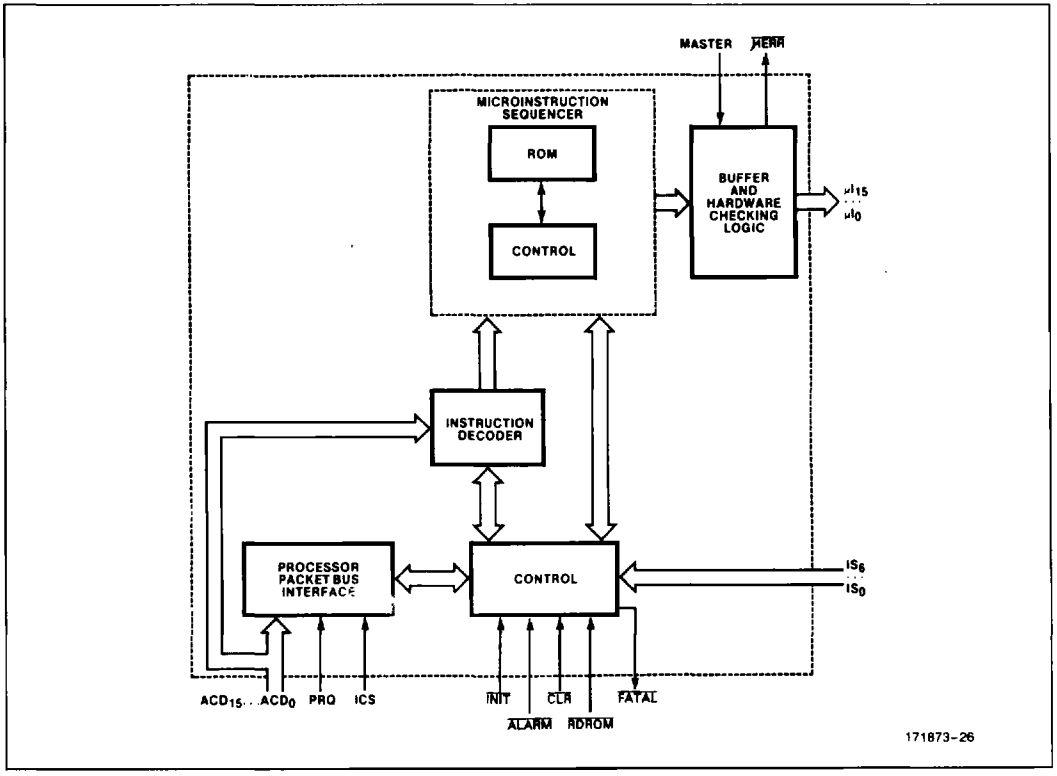


Figure 17. 43201 Block Diagram

The Instruction Decoder requests words from memory as they are needed, from one to ten bytes in a single access. Depending upon the complexity of the instruction, a 432 instruction may range from a few bits long to several hundred bits long, extending over many words.

A GDP instruction is composed of a variable number of fields and each field may contain a variable number of bits. In most cases, the encoding of a field specifies its length. The ID determines when an instruction boudary has been reached so it can properly begin decoding the next instruction.

In some cases, the interpretation of one field may depend upon the value of some previous filed. The intrepretation of the opcode (the last field in an instruction), for instance, depends on the value of the class field (the first field) in the instruction. The ID therefore saves enough information about each instruction to properly interpret each filed.

Since a GDP instruction may contain an explicit reference to some location in memory, the logical address information must be transfered to the Reference Generation Unit in order to generate the correct physical address of the operand. As with all fields in of a GDP instruction, the length of logical address fields is variable. Consequently, the ID formats the logical address and stores it until needed by the Reference Generation Unit.

Since branch instructions occur frequently, it is important to minimize the startup time for the GDP after a branch has occurred. Since an instruction may begin on any bit, the GDP is able to begin decoding at any point in a segment.

Microinstruction Sequencer

The Microinstruction Sequencer (MS) decides which microinstruction should be sent to the Execution Unit (EU) for each cycle. In doing so, it performs the following functions:

- Executes microcode sequences out of an on-chip, 4k by 16-bit ROM
- Responds to bus control signals
- Invokes macroinstruction fetches
- Issues microinstructions to the EU
- Initiates interprocessor communication and fault handling sequences

The MS chooses from two sources of microinstructions: they may come from either the ID or from the ROM in the MS. After issuing one microinstruction, the MS then computes the address in ROM (if any) for the next microinstruction. Since the EU may require differing lengths of time to complete some microinstructions, the MS waits for the requested operation to be completed before issuing the next one.

Execution Unit

The iAPX 43202 contains the third stage of the GDP pipeline—the Execution Unit (see Figure 18). The EU receives microinstructions from the 43201 and routes them to one of the two independent subpro-

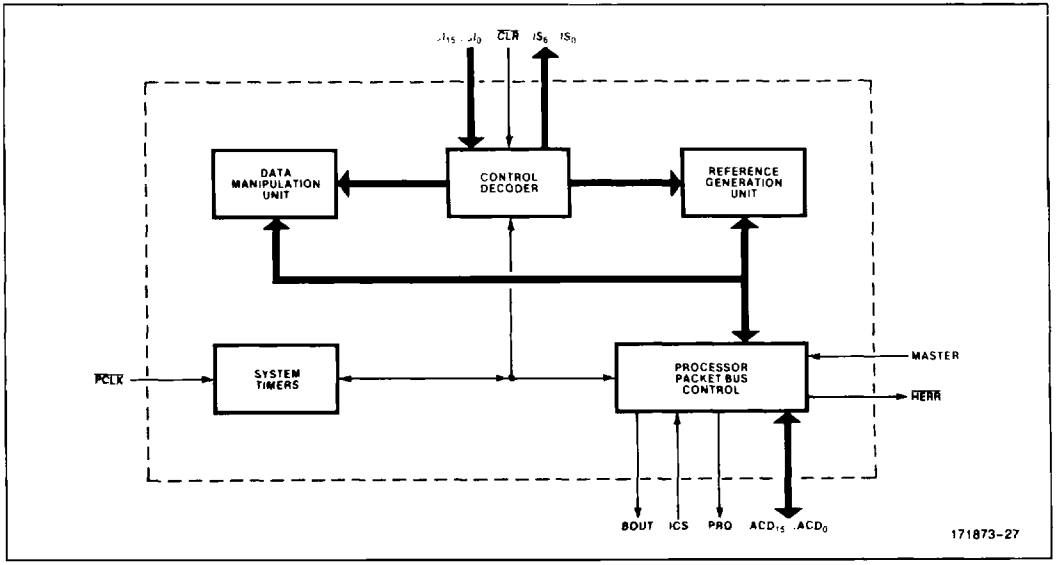


Figure 18. 43202 Block Diagram

cessors that comprise it: the Data Manipulation Unit (DMU) and the Reference Generation Unit (RGU).

While the EU executes most microinstructions in one clock cycle, each of the subprocessors has an associated sequencer that may run for many cycles in response to certain microinstructions. These sequencers are invoked, for example, for floating operations in the DMU and Processor Packet bus transactions in the RGU.

The DMU contains the registers and arithmetic logic to perform the following functions:

- Hardware recognition of nine data types
- 16- and 32-bit multiply, divide, and remainder through a built-in state machine
- Control functions for 32-, 64-, and 80-bit floating point arithmetic.

The RGU performs the following functions:

- Translates 40-bit virtual addresses into 24-bit physical addresses
- Enforces the capability-based protection system
- Sequences 8-, 16-, 32-, 64-, and 80-bit memory accesses
- Controls on-chip top-of-stack register

When a reference to a given memory segment has been translated from its logical representation to a physical address, a cache in the RGU maintains the physical base address as well as the length of the segment. Further references to the same segment reuse this information for additional address translations. A least-recently-used algorithm is implemented in hardware to determine which segment base-length pair to replace when a new segment is referenced. To further increase performance, the top 16-bit element in the operand stack is cached in the DMU.

In enforcing capability-based addressing, every memory reference is checked by the RGU to see if it is within the length of its segment, and the type of access (read, write, etc.) is verified to make certain that the object has the proper rights to perform the operation.

The iAPX 43201 and iAPX 43202 components together form a GDP. Figure 19 shows a logical representation with both units interfacing to the Processor Packet bus as a single processor. Figure 20, in turn, shows the physical layout.

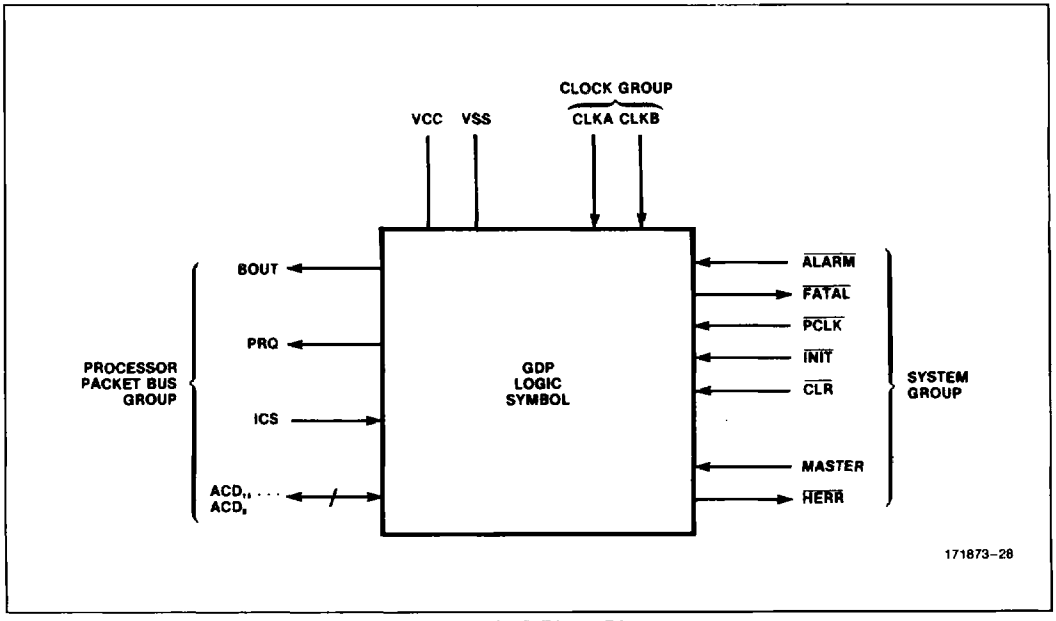


Figure 19. GDP Block Diagram

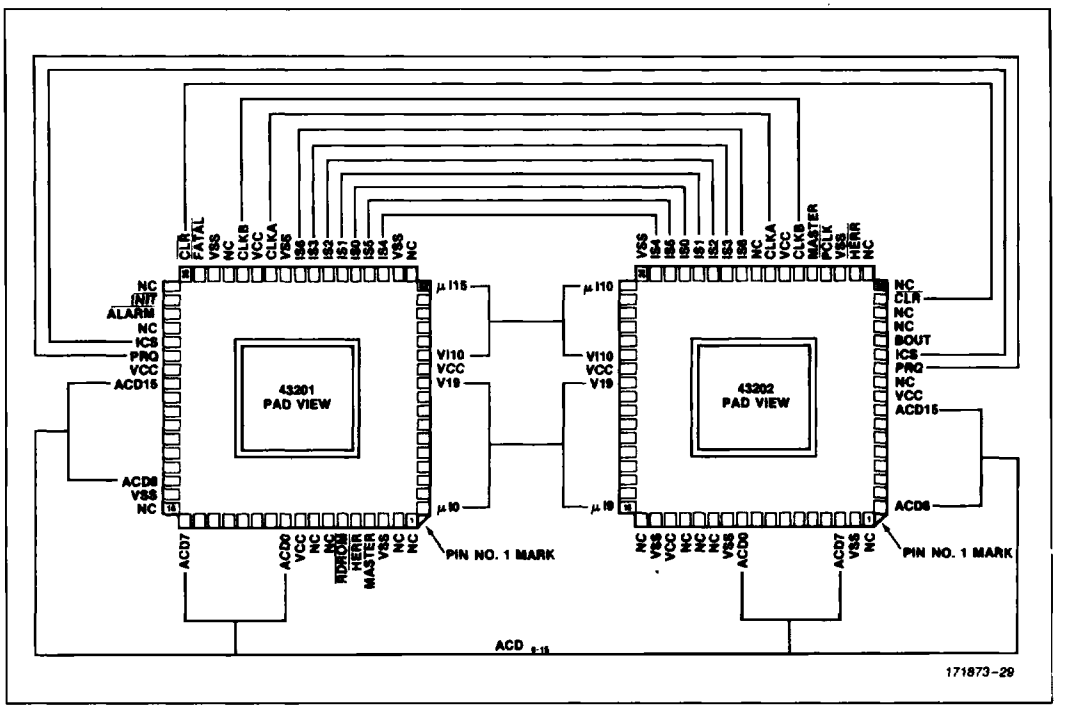


Figure 20. GDP Interconnect

PROCESSOR PACKET BUS DEFINITION

Processors sharing the same memory must contend for access to that memory over one or more system buses. Therefore, efficient bus utilization is essential in a multiprocessing system. A simple and efficient approach to building a 432 interconnect system is to use the iAPX 43204 Bus Interface Unit; the VLSI component provides the necessary circuitry to interconnect 432 processors with from one to eight memory buses. Some system designers, however, may prefer to take other approaches to the interconnect design to optimize the cost/performance ratio of the hardware for their specific application. With that requirement in mind, Intel formulated an iAPX 432 packet bus protocol which supports a wide range of system bus architectures.

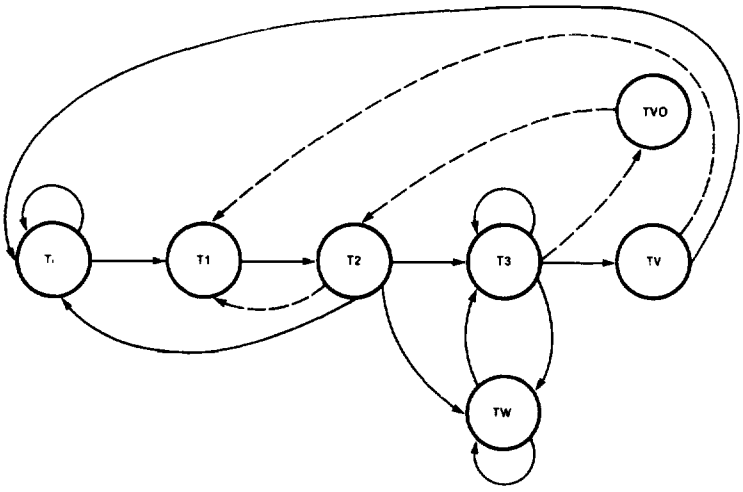
To reduce bus occupancy and increase the performance range of 432 systems, the packet bus protocol separates processor requests and replies into separate packets. A processor can issue a request packet and leave the system bus free until the reply packet is returned from memory.

As a second method of maximizing the efficiency of bus utilization, the packet bus protocol allows varaible length packets of data. If a processor wishes to read a 64-bit operand, it can be done with a single request and reply packet. Thus, fewer individual storage requests are required to process long operands. This aspect of the protocol enables processors to interface easily to 16-bit, 32-bit, or even 64-bit system buses.

This section describes the 19 signal lines that compose the Processor Packet bus and their timing relationships. While this section defines all valid bus activities, the processors do not necessarily perform all allowed activities; nevertheless, slaves to the Processor Packet bus must support all state transitions to ensure compatibility (see Figure 21).

The Processor Packet bus consists of three control lines:

- PRQ (Processor Packet bus request)
- B_{OUT} (Enable Buffers for Output)
- ICS (Interconnect Status)



171873-30

* Note that the broken transitions in the GDP state diagram are not generated by the GDP component pair.

Initial State	Next State	Trigger
Ti	T1	Bus cycle desired
	Ti	No bus cycle desired
T1	T2	Unconditional
T2	T3	ICS high
	Tw	ICS low
	T1	Canceled, Access Pending
	Ti	Canceled, No Access Pending
T3	T3	Additional transfer required, ICS high
	Tw	Additional transfer required, ICS low
	Tv	All transfers completed, no overlapped access
	Tvo	Current write with overlapped access
Tv	Ti	No access pending
	T1	Access pending
Tvo	T2	Unconditional
Tw	Tw	ICS low
	T3	ICS high

Figure 21. Processor Packet Bus State Diagram

PRQ has two functions whose use depends upon the application; for example, PRQ either indicates the first cycle of a transaction on the bus or the cancellation of a transaction initiated during the previous cycle. Of the three control lines, B_{OUT} has the simplest function, serving as a direction control for buffers in larger systems which require more electrical drive than the processor components can provide. The ICS signal has three different interpretations depending on the state of the Processor Packet bus transaction. It may indicate whether or not:

- An interprocessor communication (IPC) is waiting,
- A slave requires more time to service the processor's request, or
- A bus error has occurred.

The bus also includes 16 three-state Address/Control/Data lines (ACD_{15}–ACD_0). These lines emit information to specify the type of cycle being initiated; transmit addresses, data to be written, and control information; and during a read operation, receive data returned to the processor. Details of the ACD operation are summarized below.

Address/Control/Data Lines

In the first cycle (T1 or Tvo) of a Processor Packet bus transaction (indicated by the rising edge of PRQ), the eight high-order ACD bits (ACD_{15}–ACD_8)specify the type of the current transaction. In this first cycle, the low-order ACD bits (ACD_7–ACD_0) contain the least significant eight bits of the 24-bit address.

During the next cycle (T2), the remainder of the address is presented on the ACD pins, aligned so that the most significant byte of the address is on ACD_{15}–ACD_8 while the mid-significant byte is on ACD_7–ACD_0. If PRQ is asserted during T2, the access is cancelled and the ACD lines are not defined.

During the third bus cycle (T3 or Tw) of a Processor Packet bus transaction, the processor presents a high impedance to the ACD lines for read transactions and asserts data for write transactions.

Once the bus has entered T3 or Tv, the sequence of state transactions depends on the type of cycle requested during the preceding T1 or Tvo. Accesses ranging in length from 1 to 10 bytes may be requested (see Table 8). If a transfer of more than one double byte has been requested, T3 must be entered for every double byte that is transferred. ICS dictates whether the processor simply enters T3 or first enters Tw to wait.

After all data is transferred, the processor enters either Tv or Tvo. Tvo can be entered only when the processor is prepared to accomplish an immediate write transfer (overlapped access). During Tvo, the ACD lines contain address and specification information aligned in the same fashion as T1. If the processor does not require an overlapped access, the bus state move to Tv (the ACD lines will be high impedance). After Tv, a new bus cycle can be initiated with T1, or the processor may enter the idle state (Ti).

Table 8. ACD Specification Encoding

ACD 15	ACD 14	ACD 13	ACD 12	ACD 11	ACD 10	ACD 9	ACD 8
Access	**Op**	**RMW**	**Length**			**Modifiers**	
0–Memory	0–Read	0–Normal	000– 1 Byte 001– 2 Bytes 010– 4 Bytes 011– 6 Bytes 100– 8 Bytes 101–10 Bytes 110–16 Bytes* 111–32 Bytes* *Not implemented			ACD15 = 0: 00–Inst Seg Access 01–Stack Seg Access 10–Context Ctl Seg Access 11–Other ACD15 = 1: 00–Reserved 01–Reserved 10–Reserved 11–Interconn Register	
1–Interconnect Space	1–Write	1–RMW					

Interconnect Status (ICS)

As discussed earlier, ICS has three possible interpretations depending on the current state of the bus transaction (see Table 9). Even so, under most conditions ICS indicates whether or not an IPC is pending; a valid low during any of these cycles with IPC significance signal the processor that an IPC has been received. While an iAPX 432 processor is only required to record and service one IPC or reconfiguration request at a time, logic in the interconnect system must record and sequence multiple (and possibly simultaneous) IPC occurrences and reconfiguration requests. Thus, the logic that implements ICS must accommodate global and local IPC arrivals and requests for reconfiguration as individual events:

Table 9. ICS Interpretation

State	Significance	Level	
		High	Low
Ti, T1, T2	IPC	No IPC Waiting	IPC Waiting
T3, Tw	Stretch	Don't Stretch	Stretch
Tv, Tvo	Err	Bus Error	No Error

1. Assert IPC significance on ICS for the arrival of an IPC or reconfiguration request.

2. When the iAPX 432 processor reads interconnect address register 2, it will respond to one of the status bits for the IPC or reconfiguration request signalled on ICS in the following order:

BIT 2 (1 = reconfigure, 0 = do not reconfigure)

BIT 1 (1 = global IPC pending, 0 = no global IPC)

BIT 0 (1 = local IPC pending, 0 = no local IPC)

3. The logic in the interconnect system must clear the highest order status bit that was serviced by the iAPX 432 processor, and if an additional IPC message has arrived, the interconnect logic must signal an additional IPC to the processor by setting ICS high for at least one cycle and then setting ICS low for at least one cycle, while ICS has IPC significance.

Processor Packet Bus Request (PRQ)

PRQ is normally low and goes high only during T1, T2, and Tvo. High levels during Tvo and T1 indicate the first cycle of an access. A high level during T2 indicates that the current cycle is to be cancelled. See Table 10.

Table 10. PRQ Interpretation

State	PRQ	Condition
Ti	0	Always
T1	1	Initiate access
T2	0	Continue access
	1	Cancel access
T3	0	Always
Tw	0	Always
Tv	0	Always
Tvo	1	Initiate overlapped access

Enable Buffers for Output (B_{OUT})

B_{OUT} is provided to control external buffers when they are present. Table 11 and Figures 22 through 27 show its state under various conditions.

Processor Packet Bus Timing

Each timing diagram shown on the following pages illustrates the timing relationships on the Processor Packet bus during various types of transactions. This approach to transfer timing allows maximum time for the transfer to occur and yet guarantees hold time.

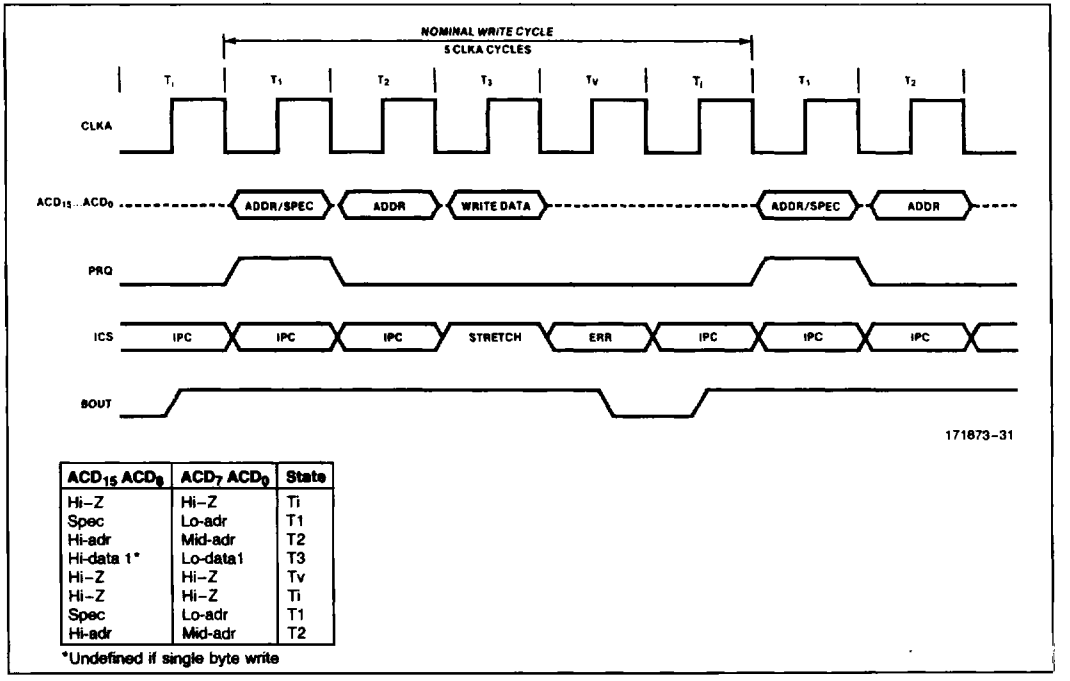
Any agent connected to the Processor Packet bus is recognized as either a processor (a GDP or IP) or a slave (e.g., the memory subsystem).

In all transfers between a processor and a slave, the data to be driven is clocked for three-quarters of a cycle before it is sampled. This allows adequate time for the transfer and ensures sufficient hold time after sampling. The B_{OUT} timing is unique because B_{OUT} functions as a direction control for external buffers.

Detailed set-up and hold times can be found in the AC Characteristics section.

Table 11. B_{OUT} Interpretation

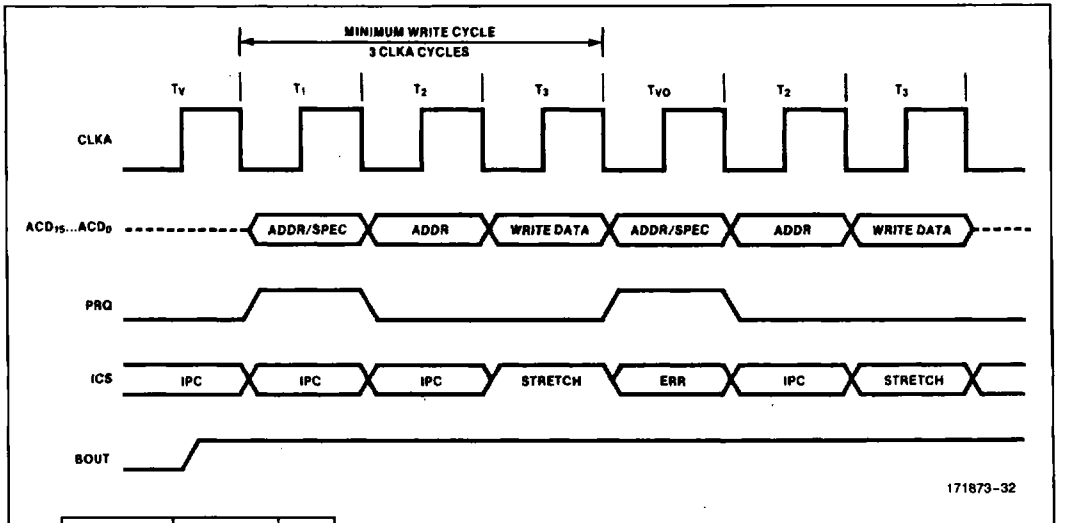
B_{OUT}	Always High	Low-to-High Transition or Low	High-to-Low Transition or Low	High-to-Low Transition or High
Write	T1, T2, T3, Tw, Tvo	Ti	None	Tv
Read	T1, T2	Ti, Tv	T3, Tw	None



ACD_{15} ACD_8	ACD_7 ACD_0	State
Hi-Z	Hi-Z	Ti
Spec	Lo-adr	T1
Hi-adr	Mid-adr	T2
Hi-data 1*	Lo-data1	T3
Hi-Z	Hi-Z	Tv
Hi-Z	Hi-Z	Ti
Spec	Lo-adr	T1
Hi-adr	Mid-adr	T2

*Undefined if single byte write

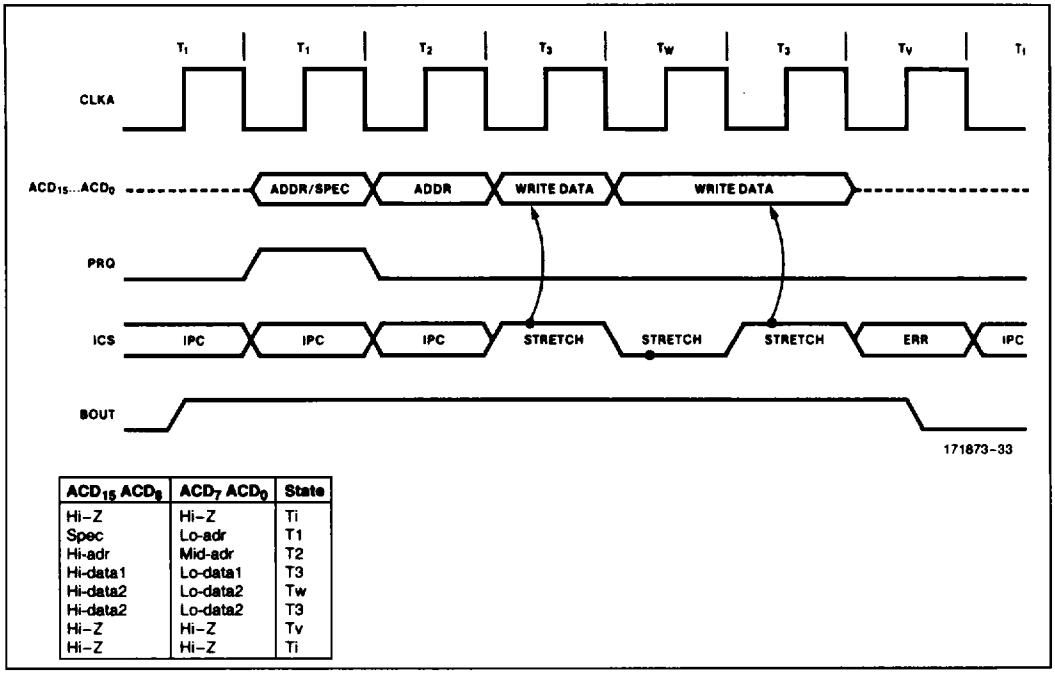
Figure 22. Nominal Write Cycle Timing



ACD_{15} ACD_8	ACD_7 ACD_0	State
Hi-Z	Hi-Z	Tv**
Spec	Lo-adr	T1
Hi-adr	Mid-adr	T2
Hi-data1*	Lo-data1	T3
Spec	Lo-adr	Tvo
Hi-adr	Mid-adr	T2
Hi-data1	Lo-data1	T3

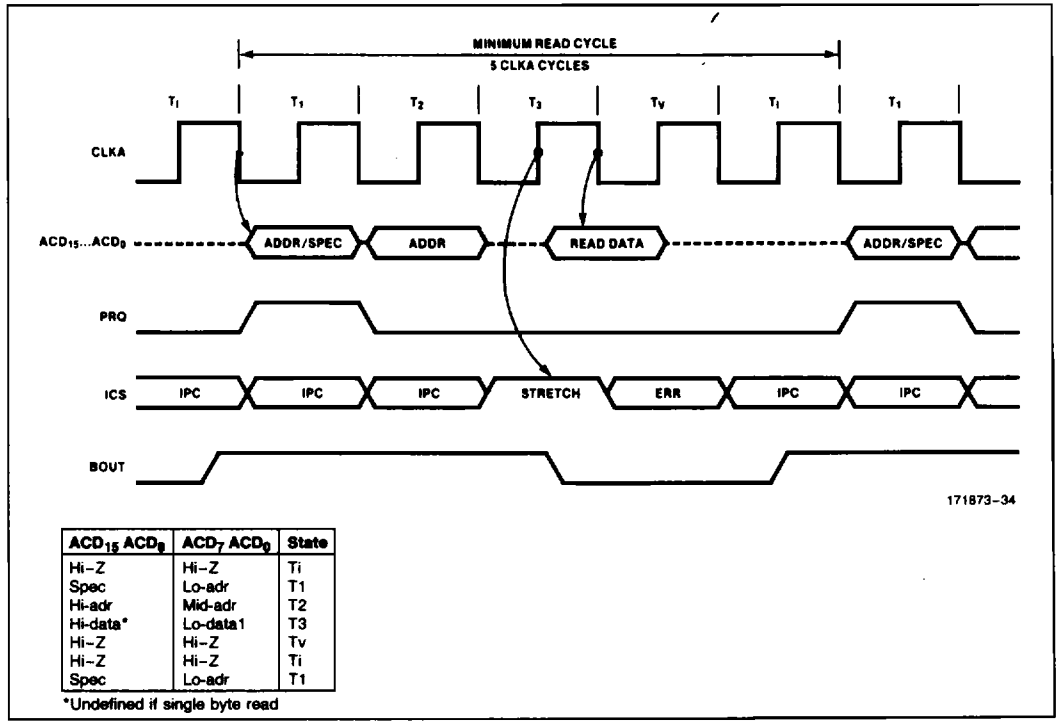
*Undefined if single byte write
**(Preceded by read cycle)

Figure 23. Minimum Write Cycle Timing



ACD_{15} ACD_8	ACD_7 ACD_0	State
Hi-Z	Hi-Z	Ti
Spec	Lo-adr	T1
Hi-adr	Mid-adr	T2
Hi-data1	Lo-data1	T3
Hi-data2	Lo-data2	Tw
Hi-data2	Lo-data2	T3
Hi-Z	Hi-Z	Tv
Hi-Z	Hi-Z	Ti

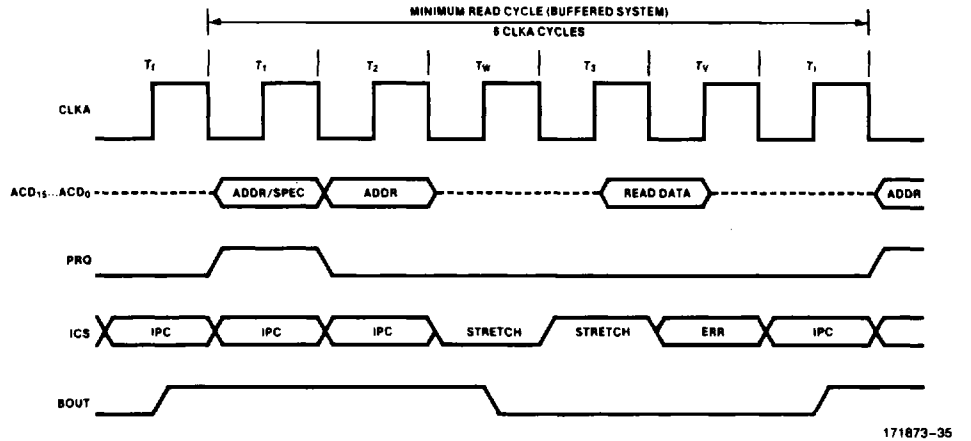
Figure 24. Stretched Write Cycle Timing



ACD_{15} ACD_8	ACD_7 ACD_0	State
Hi-Z	Hi-Z	Ti
Spec	Lo-adr	T1
Hi-adr	Mid-adr	T2
Hi-data*	Lo-data1	T3
Hi-Z	Hi-Z	Tv
Hi-Z	Hi-Z	Ti
Spec	Lo-adr	T1

*Undefined if single byte read

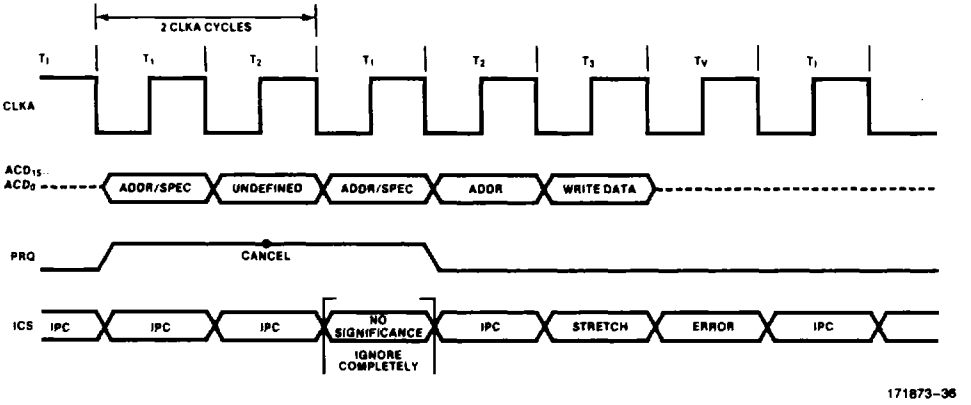
Figure 25. Minimum Read Cycle Timing



ACD_{15} ACD_8	ACD_7 ACD_0	State
Hi-Z	Hi-Z	Ti
Spec	Lo-adr	T1
Hi-adr	Mid-adr	T2
Hi-Z	Hi-Z	Tw
Hi-data1*	Lo-data1	T3
Hi-Z	Hi-Z	Tv
Hi-Z	Hi-Z	Ti

*Undefined if single byte read

Figure 26. Stretched Read Cycle Timing



ACD_{15} ACD_8	ACD_7 ACD_0	State
Hi-Z	Hi-Z	Ti
Spec	Lo-adr	T1
Undefined	Undefined	T2**
Spec	Lo-adr	T†
Hi-adr	Mid-adr	T2
Hi-data*	Lo-data	T3
Hi-Z	Hi-Z	Tv
Hi-Z	Hi-Z	Ti

*Undefined if single byte write
**Access Cancelled
†New Access Started (Slave must support this subsequent access even though all processors may not implement it.)

Figure 27. Minimum Faulted Access Cycle

Package

The iAPX 43201 and iAPX 43202 are both packaged in a 68-pin, leadless JEDEC type A hermetic chip carrier. Figure 28 illustrates the package, and Figures 1 and 2 show the pinouts.

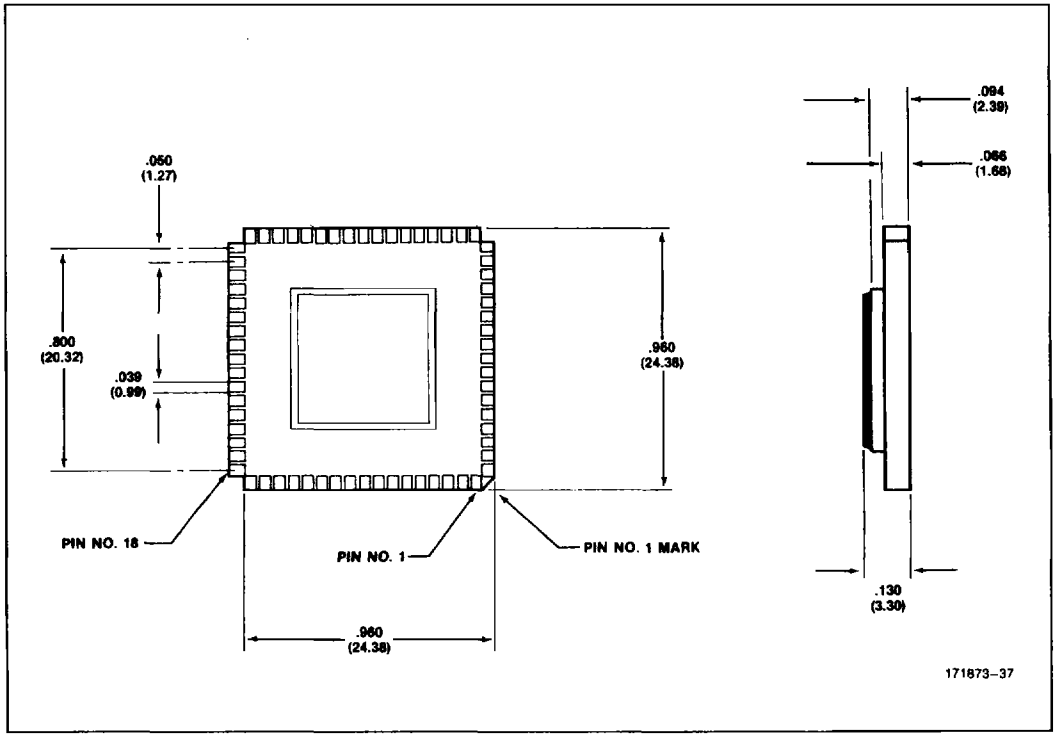


Figure 28. JEDEC Type A Package

ABSOLUTE MAXIMUM RATINGS*

Ambient Temperature Under Bias .....0°C to 70°C
Storage Temperature .....−65°C to +150°C
Voltage on Any Pin with
Respect to Ground .....−1 to + 7V
Power Dissipation.....2.5W

Notice: Stresses above those listed under "Absolute Maximum Ratings" may cause permanent damage to the device. This is a stress rating only and functional operation of the device at these or any other conditions above those indicated in the operational sections of this specification is not implied. Exposure to absolute maximum rating conditions for extended periods may affect device reliability.

DC ELECTRICAL CHARACTERISTICS (V_{SS} = 0V, V_{CC} = 5V ± 10%)

Symbol	Parameter	Min	Max	Units
V_{ILC}	Input Low Voltage Clocks	−0.3	+0.5	V
V_{IHC}*	Input High Voltage Clocks	3.5	V_{CC}+0.5	V
V_{ILI}	Input Low Voltage Intra-GDP Bus	−0.3	0.7	V
V_{IHI}	Input High Voltage Intra-GDP Bus	3.0	V_{CC}+0.5	V
V_{IL}	Input Low Voltage	−0.3	0.8	V
V_{IH}	Input High Voltage	2.0	V_{CC}+0.5	V
V_{OLI}	Output Low Voltage Intra-GDP Bus (I_{OLI} = 0.1 mA)	—	0.35	V
V_{OHI}	Output High Voltage Intra-GDP Bus (I_{OHI} = 0.1 mA)	3.25	V_{CC}	V
V_{OL}	Output Low Voltage (I_{OL}** = 2.0 mA)	—	0.45	V
V_{OH}	Output High Voltage (I_{OH} = −400 μA: 43201 −800 μA: 43202)	2.4	V_{CC}	V
I_{CC}	Power Supply Current (sum of all V_{CC} pins)	—	500	mA
I_{LI}	Input Leakage Current	—	±10	μA
O_{LI}	Output Leakage Current	—	±10	μA

*For operation at 5 MHz or slower, the GDP may be operated with V_{IHC} minimum of 2.7V.
**I_{OL} for $\overline{HERR}$ = 0.4 mA; for $\overline{FATAL}$ = 4 mA

iAPX 43201 AC CHARACTERISTICS (V_{CC} = 5V ± 10%, T_A = 0°C to 70°C)

Symbol	Description	5 MHz		7 MHz		8 MHz		Units
		Min	Max	Min	Max	Min	Max	
t_{CY}	Clock Cycle Time	200	500	143	500	125	500	ns
t_r, t_f	Clock Rise and Fall Times	0	10	0	10	0	10	ns
t1, t2, t3, t4	Clock Edge Delay Times	45	250	32	250	26	250	ns
t_{DC}	Input Signal to Clock Setup	5	—	5	—	5	—	ns
t_{DH}	Clock to Input Signal Hold Time	35	—	30	—	25	—	ns
t_{CD}	Clock to Output Signal Delay Time	—	85	—	65	—	55	ns
t_{OH}	Clock to Output Signal Hold Time	20	—	17	—	15	—	ns
T_{IE}	Input Enable Time	10	—	10	—	10	—	t_{CY}
t_{SI}	Input Signal to Init Setup Time	10	—	10	—	10	—	ns
t_{IS}	Init to Input Signal Hold Time	20	—	17	—	15	—	ns

The above specifications are subject to the following definitions and test conditions:

1. Note that tcy = t1 + t2 + t3 + t4 + 2*t_r + 2*t_f.

2. *Pins under consideration were subjected to the following purely capacitive loading:*
   C1 = 25pF on $\overline{HERR}$
   C1 = 50pF on ul15. . .ul10, IS6. . .IS0
   C1 = 70pF on all remaining pins.

3. All timings are measured with respect to the switching level of 1.5 Volts. The switching point of CLK_A and CLK_B is referenced to the 1.8 Volt Level.

4. CLK_A and CLK_B must be continuously applied for the 43201 to retain its state.

iAPX 43202 AC CHARACTERISTICS (V_{CC} = 5V ± 10%, T_A = 0°C to 70° C)

Symbol	Description	5 MHz		7 MHz		8 MHz		Units
		Min	Max	Min	Max	Min	Max	
t_{CY}	Clock Cycle Time ($t_{CY} = t_1 + t_2 + t_3 + t_4 + 2t_r + 2t_f$)	200	500	143	500	125	500	ns
t_r, t_f	Clock Rise and Fall Times	0	10	0	10	0	10	ns
t1, t2, t3, t4	Clock Pulse Widths	45	250	32	250	26	250	ns
t_{DC}	Signal to Clock Setup Time	5	—	5	—	5	—	ns
t_{DH}	Clock to Signal Hold Time	35	—	30	—	25	—	ns
t_{CD}	Clock to Signal Delay Time	—	85	—	65	—	55	ns
t_{OH}	Clock to Signal Output Time	20	—	17	—	15	—	ns
t_{DF}	Clock to Signal Data Float Time	—	75	—	75	—	55	ns

The timing characteristics given below assume the following loading on ouput pins. Loading is given in terms of a fixed capacitance plus a DC current load.

Pins	Loading
$\overline{HERR}$	90 pF Iol = 8 mA., Open Drain
B_{OUT}	70 pF Iol = 8 mA., Ioh = 800 μA
PRQ	70 pF Iol = 4 mA., Ioh = 800 μA
$IS_6 . . . IS_0$	50 pF MOS only
$ACD_{15} . . . ACD_0$	70 pF Iol = 4 mA., Ioh = 800 μA

All output delays are measured with respect to the falling edge of CLK_A except for B_{OUT}. B_{OUT} output delays are measured with respect to the rising edge of CLK_A.

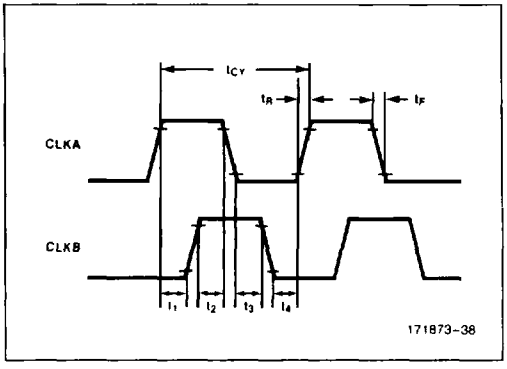
All timings are measured with respect to the switching level of 1.5 Volts. The switching point of CLK_A and CLK_B is referenced to the 1.8V level.

The 43202 is not capable of DC operation. For continuous data and logic state retention the CLK_A and CLK_B signals must be present.

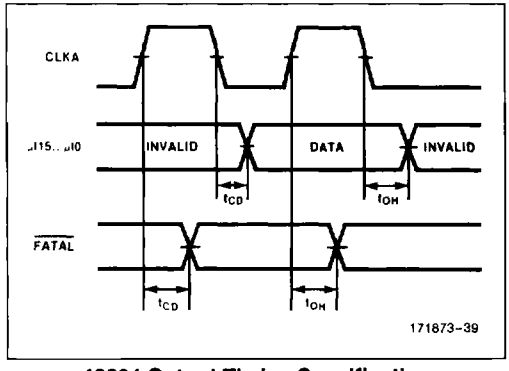
iAPX 43201/43202 Capacitance

Symbol	Parameter	Typical	Unit
C_{IN}	Input Capacitance	6	pF
C_{OUT}	Output Capacitance	12	pF
Conditions: fc = 1 MHz, V_{IN} = 0V, V_{CC} = 5V, T_A = 25°C Outputs in High Impedance State			

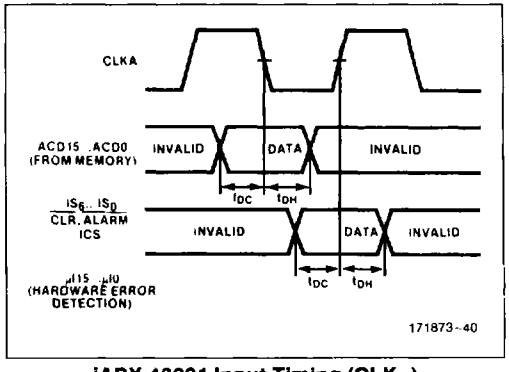
WAVEFORMS:



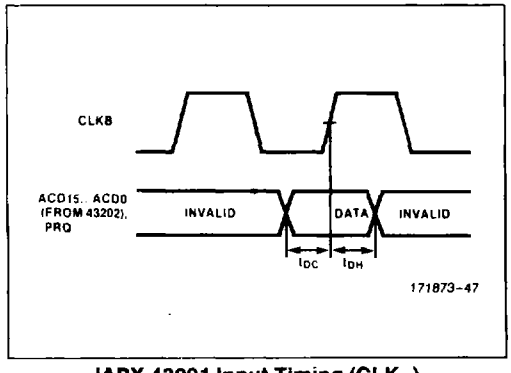
43201 Clock Input Specification



43201 Output Timing Specification

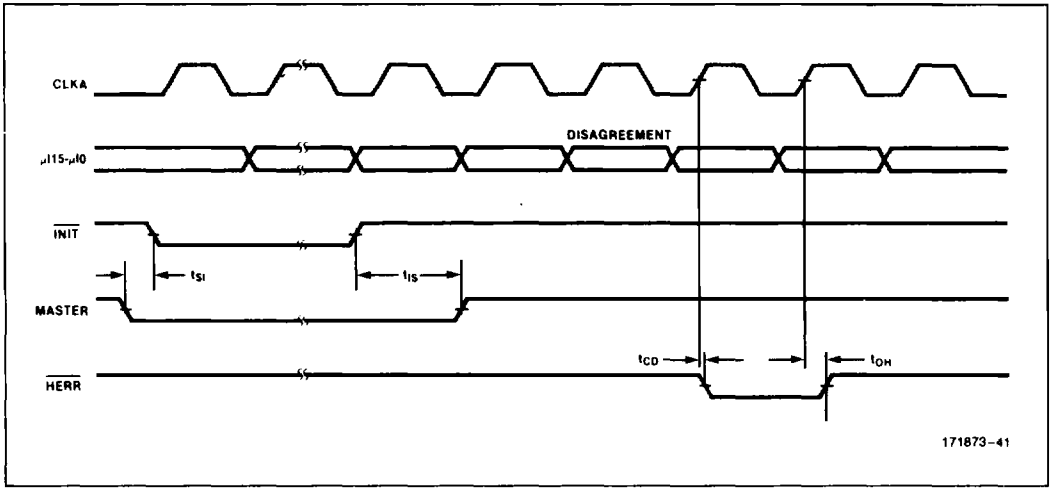


iAPX 43201 Input Timing (CLK_A)

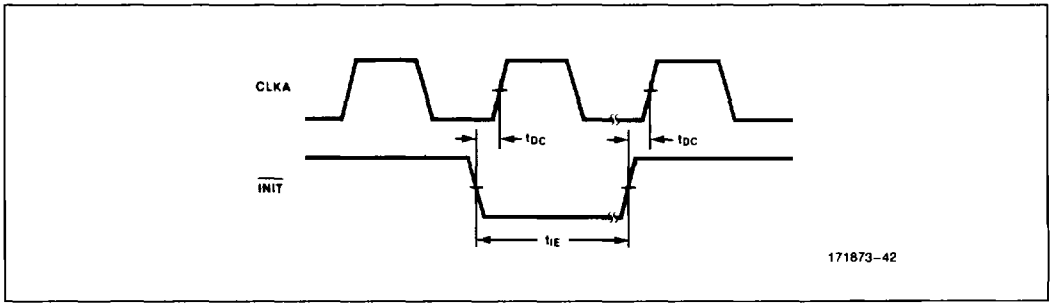


iAPX 43201 Input Timing (CLK_B)

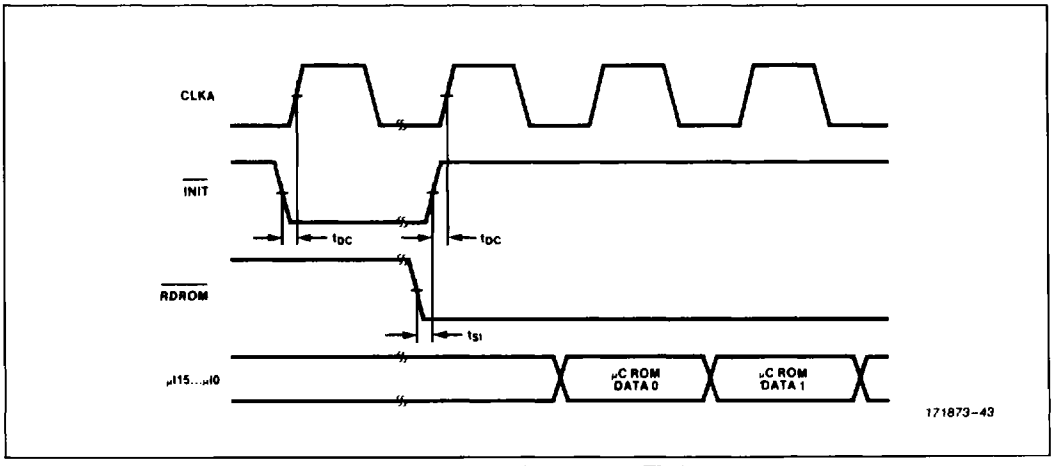
WAVEFORMS (Continued):



43201 Hardware Error Detection Timing

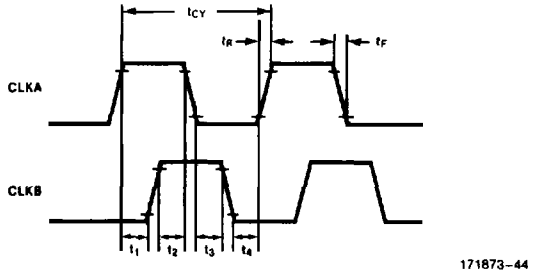


43201 Initialization Timing

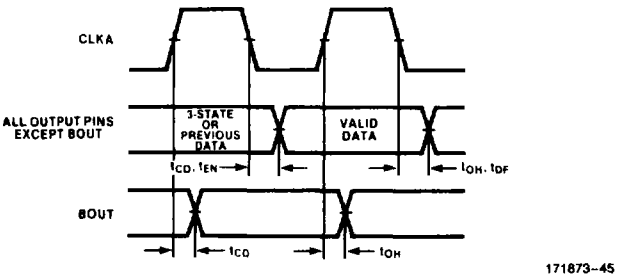


43201 Microcode Interrogate Timing

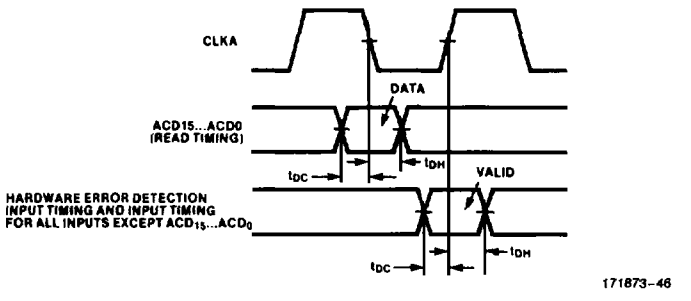
WAVEFORMS (Continued):



43202 Clock Input Specification



43202 Output Timing Specification



43202 Input Timing Specification

iAPX 432. General Data Processor Operator Set Summary

Character Operators	Short-Integer Operators	Integer Operators
Move Character	Move Short Integer	Move Integer
Zero Character	Zero Short Integer	Zero Integer
One Character	One Short Integer	One Integer
Save Character	Save Short Integer	Save Integer
AND Character	Add Short Integer	Add Integer
OR Character	Subtract Short Integer	Subtract Integer
XOR Character	Increment Short Integer	Increment Integer
XNOR Character	Decrement Short Integer	Decrement Integer
Complement Character	Negate Short Integer	Negate Integer
	Multiply Short Integer	Multiply Integer
Add Character	Divide Short Integer	Divide Integer
Subtract Character	Remainder Short Integer	Remainder Integer
Increment Character		
Decrement Character	Equal Short Integer	Equal Integer
	Not Equal Short Integer	Not Equal Integer
Equal Character	Equal Zero Short Integer	Equal Zero Integer
Not Equal Character	Not Equal Zero Short Integer	Not Equal Zero Integer
Equal Zero Character		Less Than Integer
Not Equal Zero Character	Less Than Short Integer	Less Than or Equal Integer
Less Than Character	Less Than or Equal Short Integer	Positive Integer
Less Than or Equal Character	Positive Short Integer	Negative Integer
Convert Character to Short Ordinal	Negative Short Integer	Move in Range Integer
	Move in Range Short Integer	Convert Integer to Short Integer
	Convert Short Integer to Integer	Convert Integer to Ordinal
		Convert Integer to Temporary Real
		Convert Integer to Character
		Convert Integer to Short Ordinal

Short-Ordinal Operators	Ordinal Operators	Short-Real Operators
Move Short Ordinal	Move Ordinal	Move Short Real
Zero Short Ordinal	Zero Ordinal	Zero Short Real
One Short Ordinal	One Ordinal	Save Short Real
Save Short Ordinal	Save Ordinal	
		Add Short Real—Short Real
AND Short Ordinal	AND Ordinal	Add Short Real—Temporary Real
OR Short Ordinal	OR Ordinal	Add Temporary Real—Short Real
XOR Short Ordinal	XOR Ordinal	Subtract Short Real—Short Real
XNOR Short Ordinal	XNOR Ordinal	Subtract Short Real—Temporary Real
Complement Short Ordinal	Complement Ordinal	Subtract Temporary Real—Short Real
		Multiply Short Real—Short Real
Extract Short Ordinal	Extract Ordinal	Multiply Short Real—Temporary Real
Insert Short Ordinal	Insert Ordinal	Multiply Temporary Real—Short Real
Significant Bit Short Ordinal	Significant Bit Ordinal	Divide Short Real—Short Real
		Divide Short Real—Temporary Real
Add Short Ordinal	Add Ordinal	Divide Temporary Real—Short Real
Subtract Short Ordinal	Subtract Ordinal	Negate Short Real
Increment Short Ordinal	Increment Ordinal	Absolute Value Short Real
Decrement Short Ordinal	Decrement Ordinal	
Multiply Short Ordinal	Multiply Ordinal	
Divide Short Ordinal	Divide Ordinal	
Remainder Short Ordinal	Remainder Ordinal	
	Index Ordinal	
Equal Short Ordinal	Equal Ordinal	
Not Equal Short Ordinal	Not Equal Ordinal	
Equal Zero Short Ordinal	Equal Zero Ordinal	
Not Equal Zero Short Ordinal	Not Equal Zero Ordinal	
Greater Than Short Ordinal	Less Than Ordinal	
Greater Than or Equal Short Ordinal	Less Than or Equal Ordinal	
Convert Short Ordinal to Integer	Convert Ordinal to Integer	
	Convert Ordinal to Temporary Real	

iAPX 432. General Data Processor Operator Set Summary (Continued)

Short-Real Operators	Real Operators	Temporary-Real Operators
Equal Short Real Equal Zero Short Real Less Than Short Real Less Than or Equal Short Real Positive Short Real Negative Short Real Convert Short Real to Temporary Real	Move Real Zero Real Save Real Add Real—Real Add Real—Temporary Real Add Temporary Real—Real Subtract Real—Real Subtract Real—Temporary Real Subtract Temporary Real—Real Multiply Real—Real Multiply Real—Temporary Real Multiply Temporary Real—Real Divide Real—Real Divide Real—Temporary Real Divide Temporary Real—Real Negate Real Absolute Value Real Equal Real Equal Zero Real Less Than Real Less Than or Equal Real Positive Real Negative Real Convert Real to Temporary Real	Move Temporary Real Zero Temporary Real Save Temporary Real Add Temporary Real Subtract Temporary Real Multiply Temporary Real Divide Temporary Real Remainder Temporary Real Negate Temporary Real Square Root Temporary Real Absolute Value Temporary Real Equal Temporary Real Equal Zero Temporary Real Greater Than Temporary Real Greater Than or Equal Temporary Real Positive Temporary Real Negative Temporary Real Convert Temporary Real to Ordinal Convert Temporary Real to Integer Convert Temporary Real to Short Real Convert Temporary Real to Real
Access Descriptor Movement Operators	**Type and Rights Manipulation Operators**	
Copy Access Descriptor Null Access Descriptor	Amplify Rights Restrict Rights Retrieve Type Definition	
Refinement Operators	**Object Creation Operators**	**Access Path Inspection Operators**
Create Generic Refinement Create Typed Refinement	Create Object Create Typed Object	Inspect Access Descriptor Inspect Object Equal Access Move to Embedded Data Value Move from Embedded Data Value
Access Interlock Operators	**Branch Operators**	**Interconnect Operators**
Lock Object Unlock Object Indivisibly Add Short Ordinal Indivisibly Add Ordinal Indivisibly Insert Short Ordinal Indivisibly Insert Ordinal	Branch Branch True Branch False Branch Indirect Branch Intersegment Branch Intersegment without Trace Branch Intersegment and Link Breakpoint	Move to Interconnect Move from Interconnect
Process Communication Operators	**Processor Communication Operators**	**Context Operators**
Send Receive Conditional Send Conditional Receive Surrogate Send Surrogate Receive Delay Process Read Process Clock Send Process Set Process Mode	Read Processor Status and Clock Send to Processor	Enter Environment Copy Process Globals Set Context Mode Call Call through Domain Return Return and Fault Adjust Stack Pointer Block Move

Additional Information

More information about the iAPX 432 Micromainframe architecture can be found in the following publications:

- iAPX 432 General Data Processor Architecture *Reference* Manual (Order Number 171860)
- iAPX 432 Interface Processor Architecture Reference Manual (Order Number 171863)
- iAPX 432 Interconnect Architecture Reference Manual (Order Number 172487)

Information on the electrical characteristics of other 432 components can be found in the following publications:

- iAPX 43203 Interface Processor Data Sheet (Order Number 171874)
- iAPX 43204/43205 Fault Tolerant Bus Interface and Memory Control Units (Order Number 210963)
- iAPX 43204/43205 BIU/MCU Electrical Specifications (Order Number 172867)