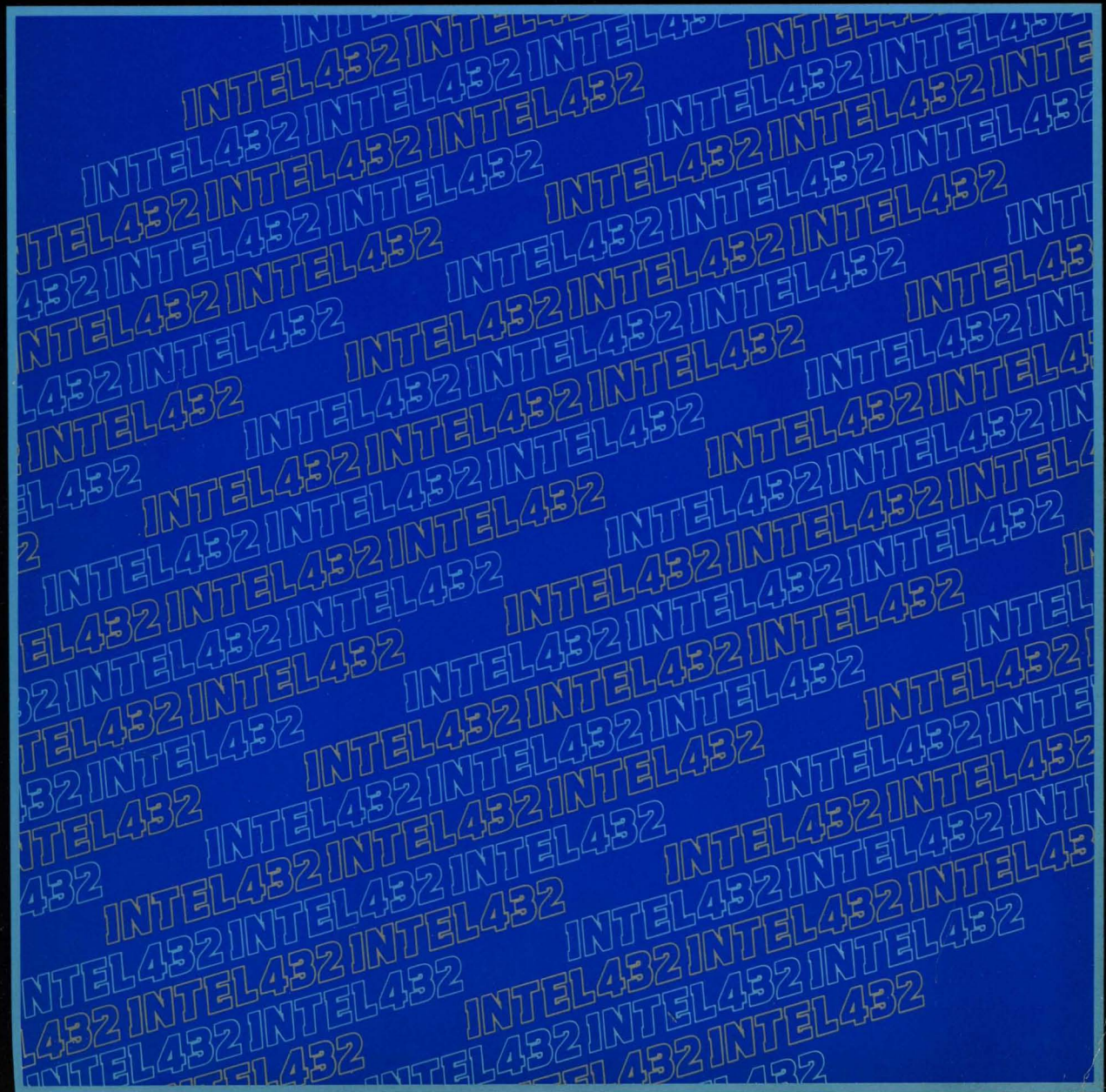
intel®

iNTEL432 SYSTEM SUMMARY: MANAGER'S PERSPECTIVE



INTEL 432

SYSTEM SUMMARY:

MANAGER'S PERSPECTIVE

Manual Order Number: 171867-001

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

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

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

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

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

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

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

BXP	Intelevision	Micromap
CREDIT	Intellec	Multibus
i	iRMX	Multimodule
ICE	iSBC	Plug-A-Bubble
iCS	iSBX	PROMPT
im	Library Manager	Promware
INSITE	MCS	RMX/80
Intel	Megachassis	System 2000
Intel	Micromainframe	UPI
		μScope

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

PREFACE

The *Intel 432 System Summary* is a series of booklets that introduces project managers and technical staff to the Intel 432 Micromainframe family. Compared to the 432's reference manuals, this material is both broader in scope and shallower in depth. On the other hand, it is not intended to teach any particular subject but to *introduce* an array of related topics. In general, the *System Summary* is the first publication anyone interested in the 432 should read.

Although the *System Summary* is introductory in nature, the presentation does assume a good general background in computer hardware and/or software. Ideally, the reader has previously developed a microcomputer-based product.

Recognizing that readers have different backgrounds and interests, the *System Summary* is published in three volumes:

- *Intel 432 System Summary: Manager's Perspective,* Order No. 171867;
- *Intel 432 System Summary: Software Engineer's Perspective,* Order No. 172177;
- *Intel 432 System Summary: Hardware Engineer's Perspective,* Order No. 172178.

The volumes are to be published in the order given above; consult your Intel sales office for information on the availability of any book in the series.

Each volume begins with an introduction that rapidly surveys the principal members of the product family. The *Manager's Perspective* links the major innovations of the 432 to the application issues that inspired them. This material is valuable to engineers as well as managers. The *Software Engineer's Perspective* begins with a description of a few key technical concepts that are central to the design of both the 432 and applications that are based upon it. It covers the 432 architecture (the compiler writer's view of the processors), Ada* (the 432's first programming language), and iMAX (the 432's Multifunction

* Ada is a trademark of the U. S. Department of Defense.

Applications Executive). The *Hardware Engineer's Perspective* is devoted to the components (chips) and to the System 432/600, a collection of board-level building blocks.

The Intel 432 is a comprehensive and evolving set of hardware and software products; the *System Summary* introduces only the cornerstones. For detailed and up-to-date information on the complete product line, consult your local Intel sales office.

CONTENTS

TABLES

ILLUSTRATIONS

INTRODUCTION

A NEW COMPUTER TECHNOLOGY

The Intel 432 is a microcomputer family. Original equipment manufacturers (OEMs) will use the 432 as one element of the larger products they build for other parties. These end products may be office automation work stations, factory information systems, large PABXs or telephone office switches, transaction processing systems, families of general-purpose computers and so on. The 432 "engine" at the heart of such a product will often be imperceptible to an end user.

While — as the above list suggests — the 432 can support a diverse array of end products, it is at the same time aimed at a distinctive class of applications. Products well-suited to the 432 exhibit some or all of the following attributes (as seen by the OEM, not the end user):

- a *range of performance* (potentially extending up to the level of a midrange mainframe) is required to span a family of related products or to provide headroom for future growth;
- *maximum dependability* (data integrity and uptime) of both hardware and software is critical;
- *software dominates* development cost and time to market;
- *concurrent execution* of many independent and cooperative activities characterizes the run-time environment;
- *growth and evolution* of services over time make software revision as important as initial development.

Compared to "traditional" microcomputer applications, these applications are larger (as measured by consumption of computer as well as staff and financial resources) and are far more complex. These factors place demanding requirements on the computer system selected to support the application.

One requirement of large applications is abundant computer resources; accordingly, the 432 family is based on a 32-bit architecture. Compared to a 16-bit machine, the 432 provides a very large actual address space, an enormous virtual address space, and a rich variety of data

types, instructions and addressing modes. While these capabilities are undeniably important, a 32-bit computer is essentially a "bigger hammer." To meet the demands of extremely complex applications, 32-bit resources are at once *necessary* and *insufficient*. That is why, for the 432, its 32-bit architecture is more a *point of departure*, than a goal.

The goal of the 432 is to **significantly reduce the life-cycle costs of complex microcomputer applications.** Toward this end, the 432 introduces a *new computer technology*, an integrated system of hardware, software and methodology. The technology of the 432 preserves the traditional microprocessor virtues of low cost, small physical size and low power consumption. Like a mainframe family, it offers a 32-bit architecture and spans a range of performance. In other important ways, the 432 resembles no computer of the past. Considered as a whole, its new technology constitutes a breakthrough in computer system design. To emphasize that this technology is inadequately described by conventional computer "classifications," the Intel 432 is called the Micromainframe family.

SYSTEM ORGANIZATION

All 432-based systems share the overall organization depicted in figure 1. The boundary between the central system and the peripheral subsystems essentially divides responsibility for data processing from input/output processing. It also serves as a protective barrier: all information in central system memory is shielded (by 432 hardware) against unauthorized access; peripheral subsystems may or may not provide any sort of protection. Finally, processing required to satisfy a critical real-time constraint (usually related to an I/O device) is generally performed in a peripheral subsystem, close to the source of the constraint.

The central system is organized as a set of 432 processors that share access to a common pool of memory and to each other. General data processors (GDPs) perform

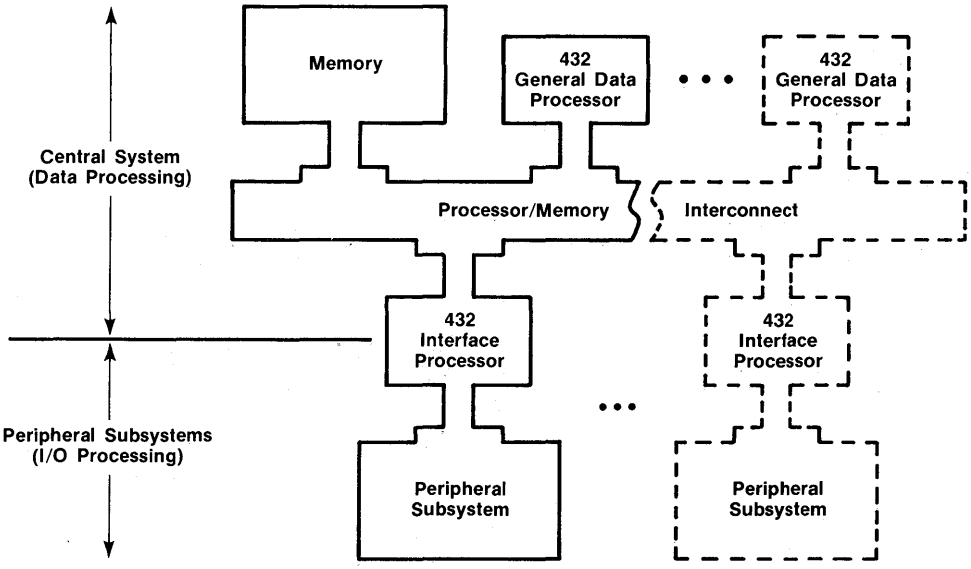


Figure 1.
General 432 System Organization

computational work, while interface processors (IPs) provide pathways for input/output to and from the central memory. The number and type of processors configured in a given system is a function of performance requirements, and can be varied independently of software. All 432 processors have built-in facilities for communicating with each other, both automatically and under software control. Additional communication facilities permit programs running on the same or different processors to exchange messages through memory.

The central system supports up to 2^{24} bytes (16 megabytes) of real memory, and a virtual memory space of 2^{40} (over a trillion) bytes. Enforced automatically by the processors, every data structure in the central memory is individually protected. It is important to note here that "data structure" means *any organized collection of information,* including such logical entities as operand stacks and sequences of code, as well as what are ordinarily considered data structures.

A multiprocessor design like the 432 permits widely differing systems to be built from a small collection of parts. No bus design could possibly satisfy the cost, size, flexibility and performance requirements of all possible system configurations. Therefore, the 432 defines a standard processor/memory communications *protocol* rather than a standard bus. Designed to minimize bus occupancy and exploit available bus width, the protocol is based on a variable-length (1 to 16 byte) *packet* of information. Processors transmit request packets to memory, and receive reply packets in response to read operations. The protocol defines interprocessor communication as well. Each application is free to design an interconnect structure that implements the protocol in conformance with local needs.

Independent decentralized I/O, along the lines of the mainframe channel concept, is inherent in the 432. Input/output operations — including all device control, interrupt handling and data buffering — are delegated to peripheral subsystems. These are autonomous satellite computers attached to the central system by means of 432 interface processors. The number and configuration of peripheral subsystems is a function of application needs and can evolve over time. Any computer that can communicate over a standard 8- or 16-bit bus, such as Intel's Multibus design (IEEE standard 769), can serve as a peripheral subsystem.

GENERAL DATA PROCESSOR

Fabricated in two 64-pin chips (see figure 2 and table 1), the 432 general data processor provides the 432's primary computational base. The GDP combines mainframe computer functionality — data types, addressing modes, basic instruction set — with the form factor, power requirements, and cost characteristics of a microprocessor.

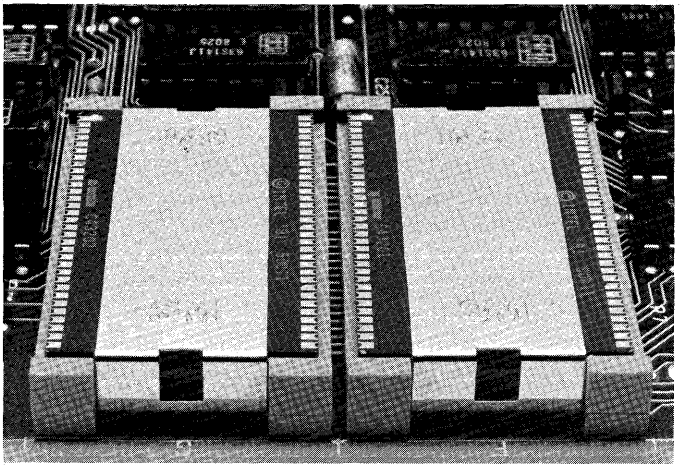


Figure 2.
General Data Processor

Table 1.
General Data Processor
Specification Summary

Addressability	2^{24} bytes physical, 2^{40} bytes virtual.
Data Types	character, 16/32 bit signed and unsigned integers, 32/64/80 bit floating point.
Addressing Modes	scalar, stack top, record element, static vector element, dynamic vector element.
Basic Instructions	
Data Transfer	move, save, zero, one.
Arithmetic	add, subtract, multiply, divide, remainder, square root, increment, decrement, negate, absolute value.
Logical	AND, OR, XOR, XNOR, complement.
Comparison	equal, not equal, equal zero, not equal zero, greater than, greater than or equal, positive, negative.
Conversion	(to any data type).
Bit Field (1-32 bits)	extract, insert, significant bit.
Control Flow	branch (conditional and unconditional), call, call with message, return.
High-level Instructions	
Communication	16 instructions (e.g., send, receive, broadcast to processors).
Storage Allocation	4 instructions (e.g., create data segment).
Mutual Exclusion	6 instructions (e.g., lock object).
Protection	14 instructions (e.g., restrict rights, inspect access).
Automatic Operations	Process dispatching and low-level scheduling, message synchronization and queuing.

(Table 1 continued on next page)

Protection	"Need to know" addressing at data structure level; attempted violations detected and reported by hardware. Automatic detection of processor hardware errors when processors are configured in self-checking pairs.	
Selected Timing Data (μsec at 8 MHz)		
Processor Cycle	0.125	
Memory Read/Write (word)	0.75	
High-level/Automatic Operations*		
Send Message	86.875	
Receive Message	96.975	
Create Segment (100,1000,10,000 bytes)	94.9, 308.375, 2417.75	
Suspend, Reschedule And Dispatch New Process	401.875	
Arithmetic*	(32-bit integers)	(80-bit floating point)
Add/Subtract	0.5	19.125
Multiply	6.375	27.875
Divide	10.625	48.25
Square Root	n.a.	55.625
Package	(Two) 64-pin quad in-line (QUIP)	
Power Requirement	+5V ±10%, 2.5 Watts	

*Preliminary figures, subject to change. Data reflect execution time only and do not include instruction fetch, operand fetch/store or pipeline refill, where applicable. Arithmetic figures are thus comparable to register-to-register operations on a register-based machine. High-level and automatic operations do include all memory accesses made during execution; in these cases the memory subsystem is assumed to provide the minimal response time, and no provision is made for delays due to arbitration or error correction.

Table 1.
General Data Processor Specification Summary (Cont.)

Compared to a conventional processor, the GDP absorbs into hardware many functions that are customarily performed by application and systems software. For example, the GDP provides hardware operations on floating point numbers in its basic instruction set. Clearly, implementing these in hardware improves performance. Less obvious effects are the simplification and improved reliability of software. Without effort, programmers reap the benefits of the very clean, thoroughly-considered algorithms of the proposed IEEE floating point standard.

Extending this concept, the GDP supplements its basic instruction set with additional high-level instructions. These instructions execute time-critical, frequently-used operations that are conventionally performed by operating system software. The operands of these systems programming instructions are not numbers or characters but data

structures that resemble conventional OS "control blocks."

Going one step further, the GDP also performs a number of functions automatically, on its own initiative, rather than in response to an instruction. For example, a GDP allocates itself among ready programs with no intervention from an operating system. Since they represent operating system software functions moved into silicon, the high-level instructions and automatic operations are collectively called the "silicon operating system."

INTERFACE PROCESSOR

The single-chip 432 interface processor (see figure 3 and table 2) functions at the central system/peripheral subsystem boundary. Acting as an "intelligent adaptor," the interface processor permits peripheral subsystem software to direct data transfers across the system boundary. The interface processor is wired into the memory space of the peripheral subsystem like a memory-mapped peripheral controller and is indistinguishable from a block of local memory. It may be addressed by PS software to execute commands and it may be addressed

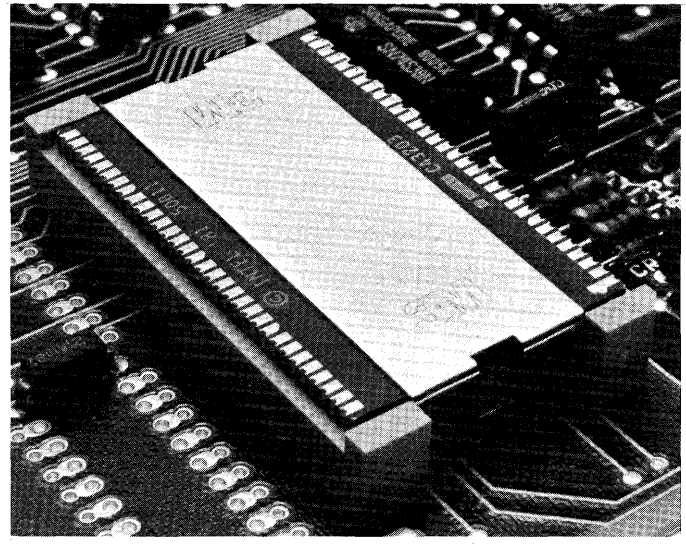


Figure 3.
Interface Processor

by any active agent (e.g., DMA controller) to transfer data. The IP's commands correspond to GDP high-level instructions; they permit peripheral subsystem software to operate within the central system environment. An important group of commands enables communication with both programs and processors in the central system.

Data transfers are performed by means of four IP data paths, called *windows*. Each window exposes one "data structure" in central system memory; peripheral subsystem software can switch a window to a different data structure by means of an IP command. To an agent in the peripheral subsystem, a window is just a range of memory addresses; writing into these addresses writes into the exposed data structure, and reading from these addresses obtains data from the data structure. Since all transfers pass through it, the IP is able to insure that a peripheral subsystem does not violate the protection standards of the central system.

Table 2.
Interface Processor Specification Summary

Addressability	2^{24} bytes physical, 2^{40} bytes virtual.
Central System Interface	Identical to GDP.
Peripheral Subsystem Interface	Appears to peripheral subsystem as memory mapped controller; 8/16 bit data bus multiplexed with 16 bit address bus; standard control and interrupt signals; compatible with Multibus architecture.
Transfer Units	Byte and double-byte.
Transfer Modes	Random: Any single location; block: contiguous set of locations up to 64kb long.
Data Paths	3 random mode, 1 switchable block/random mode.
Command Set	Equivalents for large subset of GDP high-level instructions (e.g., send, receive, lock object, etc.)
Automatic Operations	Message synchronization and queuing.
Protection	"Need to know" addressing at data structure level; attempted violations detected and reported by hardware; automatic detection of processor hardware errors when processors are configured in self-checking pairs.
Selected Timing Date (at 8 MHz)	
Maximum Data Rate	
Block Mode	5.3 mb/sec.
Random Mode	1.1 mb/sec.
High-level Commands*	
Send Message	145.25 μs.
Receive Message	141.5 μs.
Package	64-pin quad in-line (QUIP)
Power Requirements	+5V $\pm$10%, 2.5 Watts

*Preliminary figures, subject to change. System functions are for execution time only and do not include operand fetch or store times, which are highly dependent on memory configuration and bus arbitration.

SYSTEM 432/600

The System 432/600 is a set of board-level building blocks (see figure 4 and table 3) from which 432 computer systems can be built quickly with minimal investment and minimal hardware expertise. The 432/600 utilizes the multiprocessing capabilities of the 432 to provide a set of building blocks that can be configured with *unusual* flexibility.

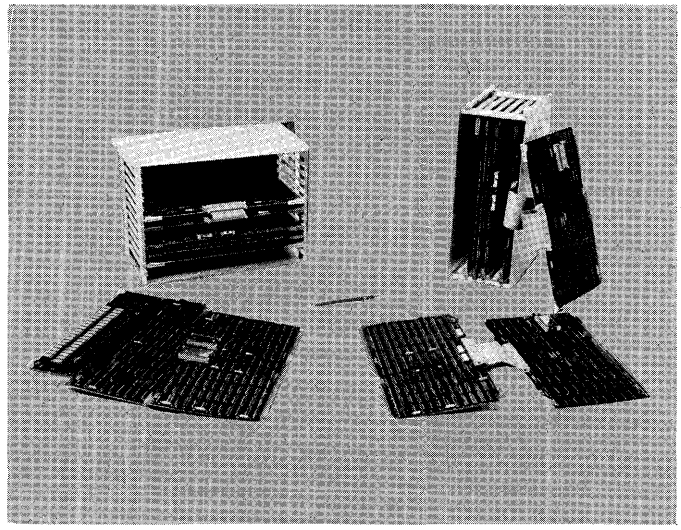


Figure 4.
System 432/600

Computing and I/O power are selected by mixing GDP boards and peripheral subsystems, respectively. By presenting a standard Multibus interface, the 432/600 permits peripheral subsystems to be built from Intel's comprehensive array of iSBC computers, memories and peripheral controllers. A 432/600 configuration may contain a total of six processing elements (GDPs and peripheral subsystems). Storage array boards of 128k-bytes and 256k-bytes permit precise matching of memory to application requirements, up to a total of four megabytes of central system storage.

In addition to these board components, the 432/600 offers diagnostic software, cardcages and backplanes, and a powered and cooled enclosure.

Table 3.
System 432/600
Specification Summary

System Organization	
Functional Elements	432 general data processor board; 432 interface processor/link board pair; memory controller board; 128/256 kb storage array boards.
System Bus	32-bit multiplexed address/specification/data bus; dedicated status and control signals; round-robin arbitration; 32 mb/sec. instantaneous data rate.
Peripheral Subsystem Bus	Standard Intel Multibus design, separate bus for each peripheral subsystem.
Form Factor	Intel iSBC (6.75 x 12 in.)
Power Requirements	+5V (±10%), 5-7 amps (max.) per board at 5V.
Data Processing	
Functional Element	general data processor board.
Extensibility	1-5 GDP boards per system.*
Memory Read Cycle	2.125 μs. (word access).
Memory Write Cycle	3.375μs. (word access).
I/O Processing	
Functional Element	Independent peripheral subsystem attached to central system via interface processor/link board pair.
Extensibility	1-5 peripheral subsystems per system.*
Data Rate	2.5 mb/sec. per subsystem.
Memory	
Functional Elements	Memory controller board, storage array board of 128 and 256 kb each.
Extensibility	128 kb - 4 mb, according to number and type of storage arrays.
Type	Dynamic RAM; onboard refresh.
Bandwidth	7 mb/sec.
Reliability	
Memory	Error checking and correction standard; all single-bit errors detected and corrected, all double-bit errors detected.
System Bus	Each byte parity-checked.
Error Reporting	Registers for system, memory and processor errors automatically updated by hardware; accessible to software.
Diagnostic Software	Rapid "go/no go" check of entire system; detailed diagnostics provided for each board type; executes on iAPX 86/88-based peripheral subsystem.
Packaging Accessories	
Back planes	6/12/18 slot units.
Cardcages	6/12/18 slot units.
Enclosure	Enclosed chassis with power and cooling, 18-slot cardcage, 12-slot system backplane, 6-slot Multibus backplane.

*The maximum number of GDPs and peripheral subsystems combined is 6.

ADA PROGRAMMING LANGUAGE

Ada (see figure 5 and table 4) is a modern high level language whose development was sponsored by the United States Department of Defense. In its basic form and "flavor," Ada resembles Pascal, probably the most influential language of the nineteen-seventies. Ada differs from Pascal in *scope;* while Pascal was developed to teach programming, Ada is a production language, explicitly

```
type        real_array is array
            (integer range <>) of float;
procedure   array_sum (
    a:                  in real_array;
    sum_x:              out float;
    sum_indexes:        out float;
    sum_squares:        out float)
is

begin
    sum_x := 0.0;
    sum_indexes := 0.0;
    sum_squares := 0.0;
    for i in a'range loop
        sum_x := sum_x + a(i);
        sum_indexes := sum_indexes + (a(i) * float(i));
        sum_squares := sum_squares + (a(i) ** 2);
    end loop;
end array_sum;
```

Figure 5.
Ada Program Fragment

Table 4.
Ada Specification Summary

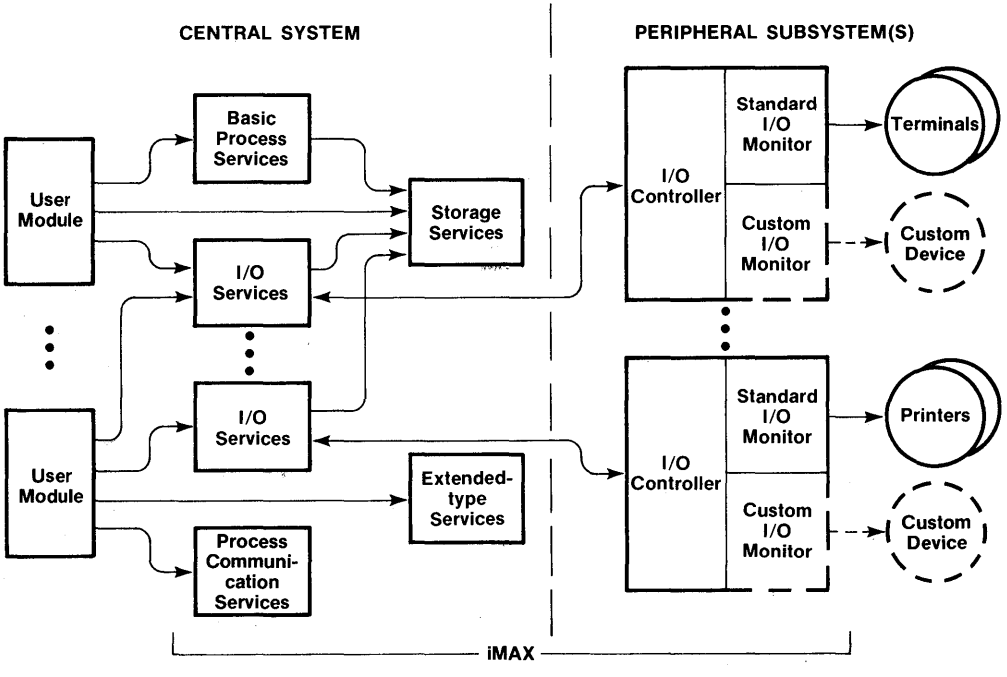
Data Types	Boolean, character, string, natural, integer, floating point, fixed point, enumeration, array, record, access, plus programmer-defined types.
Operators	
Logical	and, or, xor.
Relational/Membership	=, /= <, <=, in, not in.
Arithmetic	+, –, *, /, mod, rem, **, abs.
Other	+ (identity), - (negation), not (logical negation), & (catenation).
Control Structures	If-then-else, case, loop, begin-end, go to, return, exit.
Subprograms	Procedures and functions.
Module Structures	Subprograms, packages, tasks.
Concurrent Programming	Complete multitasking facility, including task definition, initiation, termination, prioritization, delayed execution, intertask communication and synchronization.
Input/Output	General-purpose record-oriented sequential file processing; special text file (line-and-column oriented) processing.
Other Features	Generic (macro-like) subprograms; exception detection and handling.

oriented toward the construction of software for embedded computer applications. Examples of Ada's additional capabilities include language constructs to support modular development of large systems of programs, concurrent programming (multitasking) and run-time exception handling.

iMAX 432 EXECUTIVE

iMAX (Multifunction Applications Executive) is a collection of software "components" (see figure 6 and table 5). Running on both the 432 central system and peripheral subsystems, iMAX provides basic services that are essential to most 432 applications. Carrying the notion of "components" further, users may selectively configure iMAX services into their systems, may replace iMAX modules with their own, and may add new executive services as well.

Figure 6.
iMAX Multifunction Applications Executive



System Organization	Modular "catalog" of Ada packages, each providing a related set of services for central system; peripheral subsystems coordinated by resident I/O controllers.
Programmer Interface	Ordinary Ada function and procedure calls; all calls checked at compile-time.
Storage Services	Dynamic segment creation; automatic reclamation and compaction of discarded segments in parallel with normal system operation.
Basic Process Services	Dynamic process creation, destruction and control (start, stop, synchronize, etc.); processes and subprocesses may be organized in trees; process scheduling by deadline and/or priority.
Process Communication Services	Port (data structure for queuing messages) creation; message transmittal (send, receive, sequence, etc.)
Extended-type Services	Creation, identification and access control for user-defined data structure types.
Input/Output Services	I/O monitors (device drivers) for terminal-like devices provided; users may write monitors for other devices; programmer interface supports varying degrees of device independence.
Initialization Services	Loading of initial central system memory image from designated peripheral subsystem; central system startup.

Table 5.
iMAX Specification Summary

SUMMARY

The Intel 432 is a comprehensive family of products, embracing chip- and board-level hardware components, a high level programming language and an applications executive. Together, these constitute a system designed to lower the initial cost and the continuing cost of developing complex, highly dependable end products. Such applications are further characterized by the execution of many concurrent activities, a heavy information processing orientation, and the ability to expand performance without changing software.

CHALLENGE AND INNOVATION

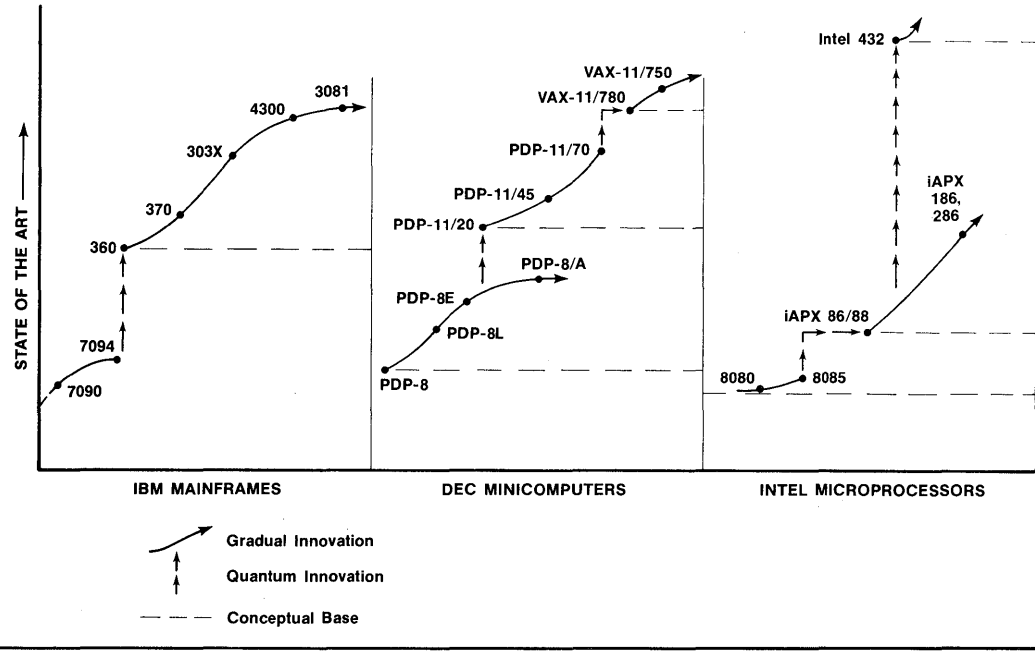
COMPUTER INNOVATION

Few fields are more dynamic and innovative than computing. New computers and lines of computers are introduced almost weekly. Yet at the same time, sampling a set of contemporary machines — even across manufacturers and classes (micro, mini or mainframe) — reveals a surprising resemblance in their basic designs.

A look at figure 7 helps to explain this phenomenon by examining computer families from IBM[1], DEC[2] and Intel; the pattern holds for most other manufacturers as well. The state of the art in computing is raised by innovation. As the chart shows, most innovation is gradual or evolutionary. A new computer (or series) usually provides an improvement over its predecessors in terms of address space, instruction set, speed or price, while carrying forward an earlier basic design concept. "Quantum" innovations, which establish new conceptual foundations,

[1]International Business Machines Corporation
[2]Digital Equipment Corporation

Figure 7.
Gradual And Quantum Computer Innovation



are rare. These few seminal machines instantly elevate the standard for the entire industry, changing the general perception of what a computer is and what it can do. The successful introduction of a quantum machine like the 360 or the PDP-11 inspires a new evolutionary cycle of derivation (and imitation) as the original manufacturer brings out enhanced models, and competitors incorporate the new concepts into their own products.

Of course this pattern is understandable when we consider that a quantum innovation is rarely possible, and then very difficult to realize. Computer innovation is enabled — if not driven — by technology. A sufficient body of technical advances must accumulate to provide the "raw material" for a new design, and technology must make the new machine buildable and economical. Even when the technical basis for a quantum innovation exists, however, market conditions must coincide to open the innovation "window."

History — in the form of past machines — encourages evolutionary development because a quantum innovation usually requires loosening compatibility constraints. Ironically, the very success of a previous innovative machine tends to restrain a company from building another one. To avoid obsoleting large customer investments in training and software, a new market must usually be identified for a quantum machine. Existing lines can then continue to be enhanced in an evolutionary fashion for the benefit of the present customer base, while the new machine builds another base.

The time, expense and risk required to pioneer a quantum computer also pushes most companies toward an evolutionary style of innovation. It is far easier to borrow and enhance proven concepts than to create new ones.

THE INTEL 432

The Intel 432 is a quantum innovation in computing. Technically, it combines software engineering and computer science advances of the 1970's with Intel's state-of-the-art MOS technology. The 432 program began in 1975 and the first chips were operational five years later; it represents Intel's largest investment in a single program.

The 432's innovations can be viewed on two levels. First, of course, it delivers 32-bit computer resources in a microcomputer package. This powerful combination opens up a whole new realm of products that can benefit from microcomputer-based design. Yet at the same time, the opportunities afforded by a 32-bit microcomputer system are deeply intertwined with an array of formidable challenges.

These challenges are inherent in the nature of the applications that a 32-bit microcomputer makes feasible for the first time. While these applications will vary enormously in function, they will share a core of common properties. Compared to 8- or 16-bit applications they will be an order of magnitude larger and more complex. Many of them will perform multiple functions concurrently, and most will evolve over many years. Finally, long-term dependable operation will be essential as people and organizations become increasingly reliant on these powerful systems. To grasp the magnitude of these new challenges, one might well consider that the software for many 32-bit microcomputer applications will rival *mainframe operating systems* in size, complexity, and the need for dependable operation.

What have we learned from the development of today's mainframe operating systems (and other ambitious programs)? The record is not encouraging. Many of these systems have never seen the light of day, having been aborted when it became apparent that they could not possibly satisfy their objectives. Those systems that have been placed in operation have uniformly been late, over budget, full of bugs in one release after another, and so fragile that they are almost impossible to modify with confidence. (In fairness, it must also be pointed out that many of these systems *do* "get the job done" in the

environment for which they have been defined — often a computer room with systems programmers and service personnel located nearby. This is far from the environment of the typical microcomputer-based product, however.)

In short, experience indicates that the traditional technology exemplified in most computers and languages is not up to the challenges of building complex, evolving, highly dependable systems. To build a 32-bit microcomputer that is a miniaturized conventional mainframe or supermini, or a scaled-up 16-bit micro, is to ignore history. The 432's second, and more profound, level of innovation is based on this recognition.

The remainder of this booklet examines the principal challenges facing the developers of complex computer-based products:

- obtaining the right *kind* of performance;
- *managing* software development and revision;
- insuring *dependable operation* of both hardware and software.

In each area the 432 responds with innovations designed to meet the challenge. Collectively, these innovations constitute a *new computer technology.*

THE EFFECTIVE PERFORMANCE CHALLENGE

To appreciate what we mean by *effective* performance, consider the following questions:

- How effective is a system that adds integers in a microsecond but requires 1,000 times as long to add real numbers?
- How effective is a system that has eight varieties of string translation instructions, none of which is ever generated by a compiler?
- How effective is a system that spends most of its time executing operating system overhead routines, rather than application code?
- How effective is any system that has a *fixed* level of performance, that cannot be adapted to changing needs?

Effective performance is more than raw instruction speed, it is speed applied to a *variety* of parameters to optimize total end product throughput, even in the face of changing demands.

A *single* 432 processor has an instruction set that effectively "automates" time-consuming functions normally performed in software. *Multiple* 432 processors work together to deliver a range of performance in both computation and input/output, similar in concept to the range provided by a compatible mainframe or minicomputer family. Finally, the current 432 product line anticipates the ability of technology to support improved performance in the future.

PERFORMANCE-DIRECTED INSTRUCTION SET

To improve the performance of any system one must apply optimization efforts selectively to obtain the best return. This is typically done by analyzing the system to see where it spends its time. That is, the system is studied to identify the routines that consume the most total time, either by virtue of lengthy duration, high frequency of execution, or both. This same approach has been applied to the design of the 432's instruction set. Rather than implement a raft of

"clever" new instructions, the 432 moves proven high-payoff functions into hardware. These fall into two general categories: numeric computation and operating system "overhead."

Floating point numbers provide a useful approximation of the real number system that people use for most calculations. They can express not only integers but also fractions and irrationals, and are further capable of great range and precision. Because of their versatility, floating point numbers are very attractive for microcomputer applications. Unfortunately, the floating point algorithms are quite complex. When they are implemented in software, floating point operations are so expensive that they must be ruled out for many applications. For example, addition and square root take about 1,600 and 19,600 microseconds respectively when executed in software on a 5MHz iAPX 86. By moving floating point algorithms into hardware, the 432 makes the convenience of floating point arithmetic a practical alternative for most applications (add and square root execution times are about 13 and 56 microseconds respectively). Note that this same argument is the rationale for the Intel 8087 Numeric Data Processor, which brings hardware floating point to 16-bit applications.

Three important functions, which are normally executed by operating system subroutines, have been absorbed into the hardware of the 432. First, as amplified in the following section, general data processors assume responsibility for program dispatching — switching themselves among multiple programs — with no intervention from software. Second, the 432 maintains pools of free memory; storage can be allocated from a pool with a single GDP instruction. Third, both GDPs and IPs support a general-purpose system of program-to-program communication. A message, consisting of any data structure in memory, may be sent or received in a single instruction; messages are queued automatically by the hardware so that send and receive operations may be executed independently of one another.

INCREMENTAL COMPUTING POWER

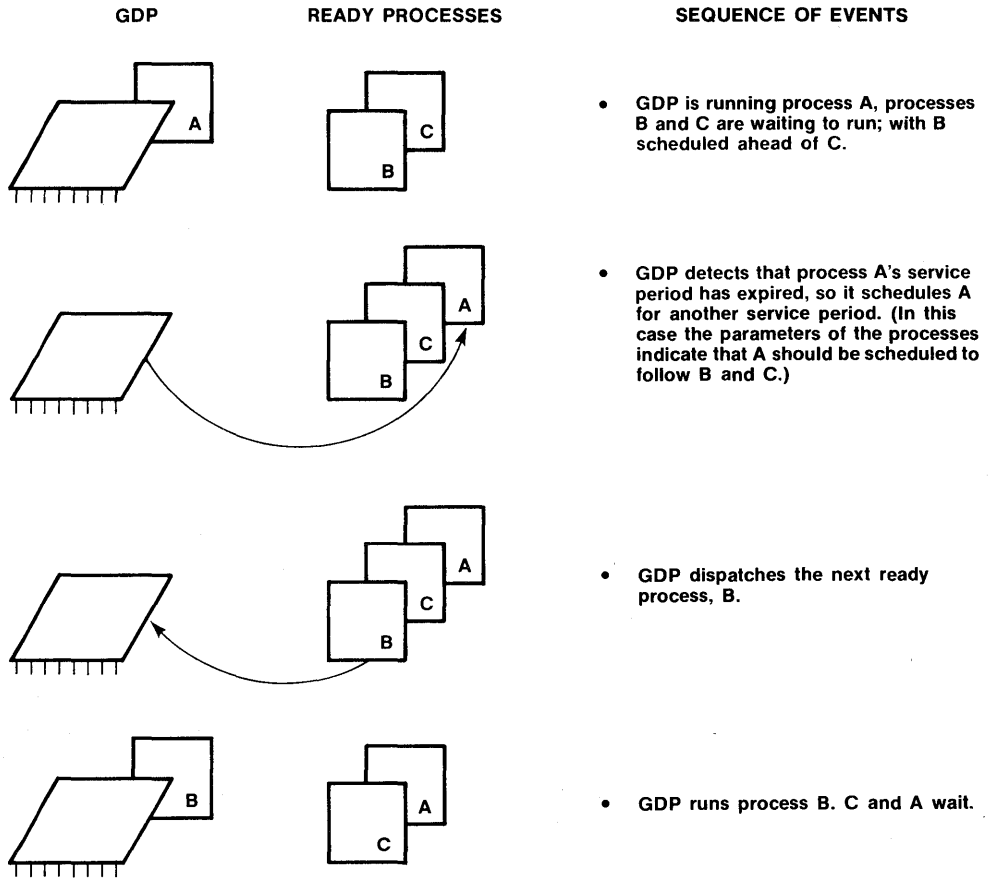
Since the introduction of the IBM System/360, typical mainframe and minicomputer product lines have been organized as compatible families. Each model of the family offers a different level of performance (and price). While the capability of an individual model is limited, the family as a whole spans a wide range of performance. To obtain a performance increment, the customer must change hardware (i.e., substitute one model for another), but — importantly — existing software is preserved intact.

This kind of flexible performance, taken for granted by mainframe and minicomputer users, is not so well-developed in microcomputers. A given microcomputer provides a single level of performance. Over time, new versions that run at higher clock rates are typically introduced; over more time, a new "generation" usually emerges with a different, but upward-compatible architecture. Thus, given the passage of time, an application can usually "trade up" to a faster processor. Often the substitution requires substantial hardware redesign.

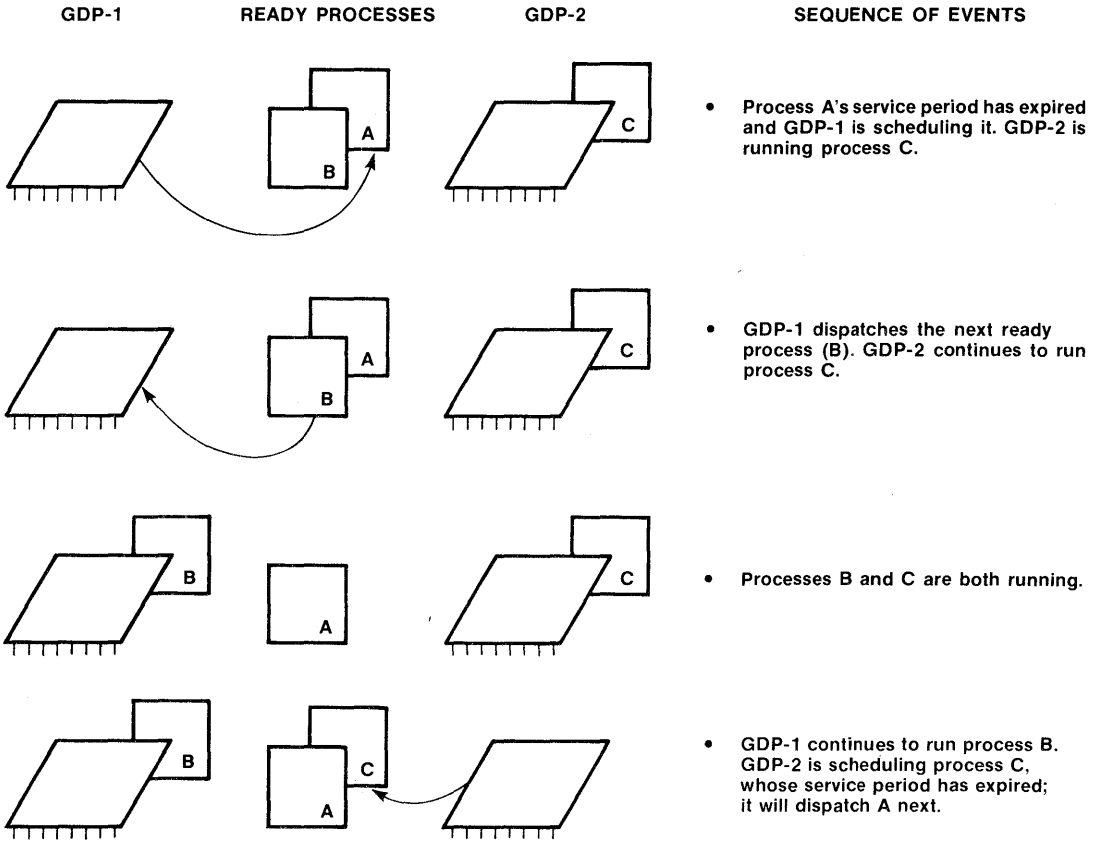
This approach is satisfactory in applications that are fairly well-defined with relatively bounded execution-time behavior, as microcomputer-based systems have tended to be. As applications become more complex and more dynamic, however, it becomes increasingly difficult to predict how much processing power a system will need to meet its performance goals. This uncertainty is a serious source of risk since an application must usually commit itself to a processor two years before any software has been written. Even when the "right" decision is made initially, design changes and extensions that crop up during development and after introduction can threaten a system's viability. A second limitation of the "new model" approach is that it effectively prohibits designing a family of end product models that use the same hardware and software components to provide different performance levels. In response to these challenges, the 432 has been designed with incremental performance as an inherent feature of the architecture.

Overall computational throughput in a 432-based design can be adjusted by changing the number of general data processors in the system. These processors are *self-dispatching*; the normal cycle of running a program unit (a process) until it waits for an event (blocks) or times out, re-scheduling it for subsequent execution, and then switching to the next ready process is performed automatically by the GDPs (see figure 8). Software, including the operating system, can be completely unaware of how many

Figure 8.
Self-dispatching



a. With One GDP



b. With Two GDPs

Figure 8.
Self-dispatching (cont'd)

processors are present in the system. In everyday terms, the effect is somewhat like the single service line used at many banks. The line of customers waiting to be served will move faster if additional tellers open their windows. Since the number of processors is invisible to software, this important facility of the 432 is called *transparent multiprocessing.*

While transparent multiprocessing is most attractive for its ability to add computing power to a system, it is also

important to recognize the significance of being able to *reduce* the number of processors in a system without altering software. If, in a multiple-GDP system, a processor fails, its circuit board can simply be removed from the system. The system can be restarted and it will then provide the same services but at a slower rate. In this way transparent multiprocessing permits a system to remain useful after a processor failure.

Of course, there are limitations to the performance improvement that can be realized by adding processors, and these break down into two general areas. First, in order to utilize the processing potential of multiple GDPs, there must be work for them to do. Most large applications naturally break down into a collection of parallel activities (processes or tasks) regardless of the computer they are destined to run on. (Transparent multiprocessing is, in fact, based on the recognition and exploitation of this natural tendency.) So in practice, most 432 applications lend themselves to effective utilization of transparent multiprocessing as a matter of course. Furthermore, when programmers know that multiple processors are available, they can often design algorithms that consciously exploit the parallel processing capabilities of the system.

The second limitation is the ability of the application's hardware configuration to keep the processors running at full speed. The throughput increment obtained by adding a processor can be constrained by memory bandwidth and latency. Accordingly the 432 permits wide latitude in the design of the memory system as well as the processor/memory interconnect.

DISTRIBUTED INPUT/OUTPUT

In much the same way that 432 computing power is adjustable by varying the number of GDPs in the system, 432 I/O power is a function of the number of peripheral subsystems attached to the central system. Since each peripheral subsystem is an independent computer, I/O capacity is variable along another dimension according to the character (i.e., hardware configuration) of each

peripheral subsystem. Additional terminals could be added to a system, for example, either by attaching them to existing subsystems with excess capacity, or by incorporating them in a new peripheral subsystem.

The central system (via an interface processor) presents a very general physical and logical interface to a peripheral subsystem. This encourages the design of different specialized peripheral subsystems that perform critical operations optimally. It also permits many existing systems to be attached to the 432 with a minimum of modification to hardware and software. Finally, the general interface simplifies the modification of a peripheral subsystem — to take advantage of improved technology, for example — and helps to insure that the changes are localized within the subsystem.

The 432 relies on the intelligence of peripheral subsystems to absorb the great majority of processing required to support I/O transfers. Distributing responsibility in this way allows multiple I/O operations to proceed in parallel with each other, and with computation and data processing operations running in the central system. A system's aggregate I/O capacity is subject to the normal limitations imposed by memory latency and bandwidth.

FUTURE PERFORMANCE

The 432 has been explicitly designed to provide improved performance in the future while maintaining the overall system framework described in this book. The classic precipitator of a new generation is the "running out of memory" syndrome that has been repeated again and again in microcomputers, minicomputers and mainframes alike. With 16 megabytes of real memory space and a trillion of virtual, the 432 moves this traditional barrier far into the future.

To further extend its useful lifetime, the 432 anticipates continuing semiconductor technology advances by carefully distinguishing between *architecture* and *implementation*. On-chip registers, caches, stack pointers and

so on have been made invisible because they are implementation features designed to provide speed, rather than function. Any or all of them can be redesigned to obtain better performance without affecting the machine as seen by a compiler — its architecture.

The 432 is also designed to permit new processors to be neatly integrated into the present family. The GDP and the IP are distinct functionally specialized processor extensions to the 432 *common base* architecture. As experience with the product line grows and new market needs develop, additional compatible processor types can be built to extend the common base architecture in other directions.

THE SOFTWARE MANAGEMENT CHALLENGE

Experience with mainframe-based programs has shown repeatedly that large software systems are among the most difficult engineering projects that men and women undertake. The large system that is delivered on schedule, within budget, and with performance as promised, is a rarity. Systems that involve real-time or concurrent processing — the norm in microcomputer applications — are even more challenging. Finally, successfully modifying or extending large systems has proven so difficult that many organizations spend well over half their budgets for software "maintenance."

Microprocessor users were originally spared the problems of large-scale programming because the earlier machines simply could not support large systems. The introduction of 16-bit machines, however, brought multi-person projects producing upwards of 100,000 lines of source code. To develop a system that exploits the power of a 32-bit microcomputer system in a timely manner will require multiple *teams* of programmers working in parallel. They will develop software that is orders of magnitude larger and more complex than anything ever written for an 8-bit machine. In this new era, managing software size and complexity becomes as critical to project success as discovering and expressing the algorithms that direct the operation of the end product.

The essence of the problem is that complexity increases exponentially with size: it is far easier to write 50 programs of 1,000 lines each than to write a single system of 50,000 lines. While limited address spaces and processing power made early microcomputer-based systems small by definition, machines like the 432 make it possible to build very large systems indeed. To learn from the mainframe experience, rather than repeat it, new tools are needed to reduce complexity where possible, and to otherwise actively *manage* it.

Addressing the challenge of reducing and managing software complexity, the 432 provides innovations at four levels: first, a machine that is an efficient target for high level language compilers; second, a language explicitly

designed for building large systems of programs; third, an executive that provides a "catalog" of widely-applicable core functions that can be selected, extended and modified according to local requirements. Finally, the 432 has been designed from the ground up to support concurrent programming and concurrent execution.

COMPILER-ORIENTED MACHINE

A given program can be written, tested and modified faster in a high level programming language than in an assembly language. The basic concept at work here is simplification: a high level language removes the attributes (and idiosyncracies) of the computer from the programmer's "problem set." The language replaces memory locations. registers, addressing modes and so on, with high level constructs: data structures (such as arrays, records and lists) and control structures (like loops and selection statements) that have been designed to promote the natural and correct expression of problem solutions.

High level languages have been accepted somewhat slowly in microcomputer-based systems primarily because the tradeoff between programming costs and manufacturing costs has often favored assembly language. Memory still dominates the cost of microcomputer system hardware; a compiler for a typical microcomputer will generally produce a larger program than will a *skilled* assembly language programmer. A larger program generally means slower execution, so to achieve comparable performance, a system programmed in a high level language will require faster (and more expensive) hardware. For applications that are comparatively small and that are produced in high volumes (e.g., those based on single-chip microcomputers), assembly language will continue to make sense. For very large applications, such as those made possible by a 32-bit microcomputer system, high level programming is a practical necessity, since every effort must be made to reduce the complexity of the programming task. For applications in the middle ground, the costs must be weighed case by case. Of course as labor and hardware costs continue their opposing trends, the balance will increasingly tip in favor of high level languages.

To make high level programming economically sensible for all 432-based products, the 432 general data processor has been designed to favor code generation by *compilers* rather than assembly language programmers. The machine itself makes it possible for straightforward, non-optimizing compilers to generate code that approaches assembly language programming in size and speed. The strength of this commitment to high level languages, and the success of the design, is indicated by the fact that Intel does not supply an assembler for the GDP, nor is one used in-house. The portion of the iMAX executive that runs on GDPs, for example, is written entirely in Ada.

There are no visible registers in the GDP; all instruction operands are memory-based. A good assembly language programmer regards registers as high-speed local storage which can be exploited by considering the run-time behavior of the program being written. A compiler, on the other hand, has no knowledge of the run-time behavior of the program it is translating; to it, registers present not so much an opportunity for optimization as a scarce resource which must be managed. Registers *are* used internally in the GDP to improve performance. By keeping registers "behind the scenes," rather than as visible features of the architecture, the GDP reduces compiler complexity. Moreover, new registers can be added to future GDPs to enhance performance without impacting existing software.

The GDP promotes the generation of fast, compact code by requiring fewer machine instructions per high level statement. A compiler for a typical computer must generate several machine instructions to implement each high level language statement. In contrast, the 432's instruction set (in conjunction with its data types and addressing modes) matches typical high level statements on a one-for-one basis (see table 6). A compiler for the 432 simply generates the "obvious" machine instruction(s).

In addition to requiring fewer machine instructions, 432 programs are also made more compact by the instruction encoding technique. As in many modern computers, the instructions that occur most often are shorter in length

Ada Statement and Interpretation[1]		432 Machine Instructions	Conventional Machine Instructions[2]
E := F;	- -assign simple variable (scalar) F to simple variable E.	1 (MOVE)	2
A(I) := X.B;	- -assign element B of record X to I'th element of vector - -A.	1 (MOVE)	3
A(I) := A(I) + C;	- -increment I'th element of vector A by value of simple - -variable C.	1 (ADD)	4
X.B := C/D(I);	- -divide simple variable C by I'th element of vector - -D, and assign result to element B of record A.	1 (DIVIDE)	4
A(I) := X.B-(E*F);	- -multiply simple variables E and F, subtract the result - -from element B of record X, and assign the result to the I'th element of vector A.	2 (MULTIPLY, SUBTRACT)	7

NOTES:

[1]Assume A..F are variables of the same type (e.g., all integers or all reals) and that the value of I is in the range 0..65,535.

[2]Assume a typical register-oriented computer. For example, the instructions required to implement A(I) := A(I) + C might be:

```
LOAD    reg1, I
LOAD    reg2, A(reg1)
ADD     reg2, C
STORE   A(reg1), reg2
```

Table 6.
High Level Language Statements And Machine Instructions

than those used less frequently (e.g., MOVE is shorter than SQUARE ROOT). Instruction references to top-of-stack operands are encoded *implicitly*; no bits are needed to specify the location of these operands. When a data item is used twice in the same instruction (e.g., A(I) := A(I) * B), only a single reference is encoded in the instruction. Finally, instructions are encoded *bitwise* (see figure 9): each instruction consists of a variable number of variable-length bit fields. The shortest instructions are six bits long, while the longest are over 300 bits. Successive instructions are located immediately adjacent to one another in memory, irrespective of word or byte boundaries. (The processor *fetches* instructions in 32-bit words and extracts the bit fields internally.) There are no unused bits in the instruction stream, either within one instruction or between two instructions.

The GDP's register-free architecture, high level instructions, data types and addressing modes, and very dense instruction encoding substantially narrow the "efficiency gap" between compiler-generated and hand-coded

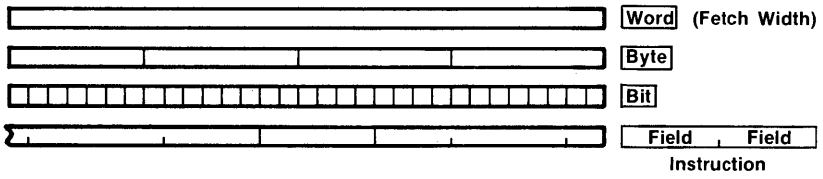


Figure 9.
Bitwise Instruction Encoding

programs. Straightforward, non-optimizing compilers will routinely produce code that approaches assembly language in efficiency. This in turn makes the benefits of high level programming economically practical for even high-volume 432 applications.

MODULAR PROGRAMMING LANGUAGE

Large software systems are inherently complex; no one can fully understand a system of 100,000 lines. Nor is there time for one person to develop a large system. As a result, large systems are classically developed according to a "divide and conquer" strategy. The large problem is divided into smaller independent subproblems that can be parceled out to teams and individuals. The subsolutions are developed in parallel and are integrated into a single solution that solves the whole problem. In a similar vein, large software systems are usually modified (corrected or extended) by identifying the part(s) of the system to be altered or added, making the changes and additions, and then reintegrating the old, new, and modified parts to produce a new version of the whole system.

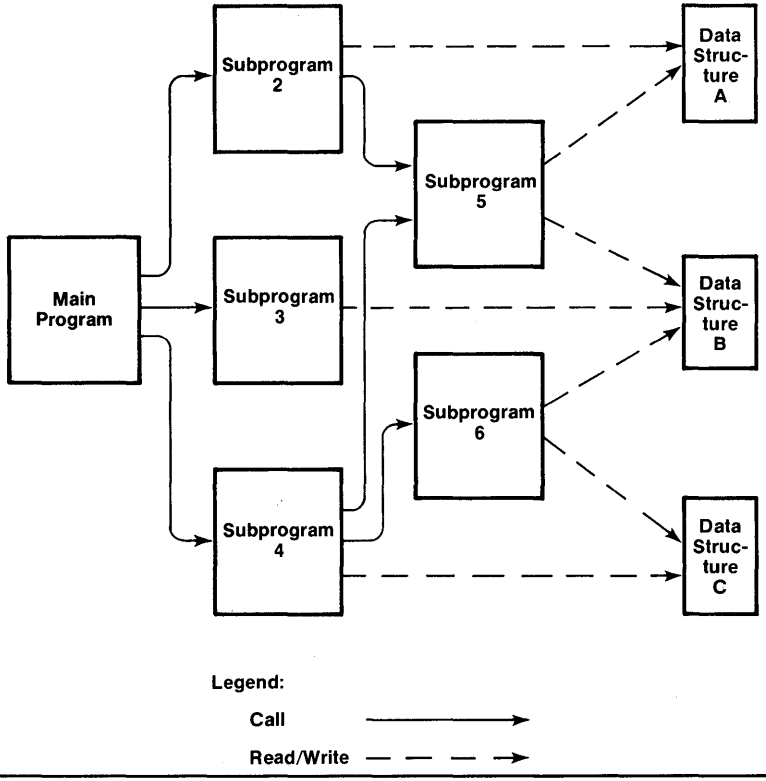
The parts into which a system is decomposed, both for initial development and later modification, are called modules. A module may be compiled independently of other modules, and provides a service that other modules can call upon. Modular programming has been practiced in various forms for many years; its goal is to make the

inherent complexity of a large system *manageable* by dividing the system into units that can be considered *independently*, that is, in isolation from the rest of the system. In theory, a programmer can then develop or modify a module without affecting — or even understanding — the rest of the system. Thus, in a modular system, it should be possible to:

- develop modules separately and in parallel with the assurance that they will fit together;
- quickly identify the module that is the source of a bug;
- introduce a new module into the system without jeopardizing what already exists;
- change one module without changing any others.

In practice, these goals have been discouragingly elusive. Figure 10 illustrates a conventional decomposition in which

Figure 10.
Subprogram-based System Organization



a module is a subprogram (e.g., a procedure) that performs a well-defined function. This "decomposition by function" is encouraged by most languages whose only support for modular programming is the ability to compile subprograms independently. With this approach, data is considered of secondary importance to algorithms; the data is typically defined "as needed" to make the algorithms work. As a result, systems designed in this manner usually contain a number of global data structures that are accessed by many different modules.

Modules that share access to a data structure violate the basic intent of modular design: they are made *interdependent* by virtue of their mutual dependence on the data structure. What are the consequences of this interdependence? First, programmers writing modules that share a data structure must coordinate the detailed definition of the structure and any changes made to it as the development progresses. Second, damage discovered in a data structure can be attributable to any of the modules that have access to it. Third, a new module that accesses a shared data structure threatens all other modules that use that structure. Fourth, a change in the way one module uses the data structure can necessitate changing some (it can be difficult to tell which) or all of the other modules that use it.

Recognizing the importance of modular programming as well as the shortcomings of past methodologies and tools, Ada defines a new kind of module as an inherent element of the language. This module is called a *package* (see figure 11), and is the basis for an integrated approach to modular programming. (This section concentrates on describing the *effects* of package-based program structure; more information on packages *per se* is provided in the *Software Engineer's Perspective.*)

A package defines both a data structure and a set of operations on that data structure. (Note that a "data structure" can represent an I/O channel, a bank account, or any other type of "thing" managed by the application.) An operation is written as an ordinary procedure or

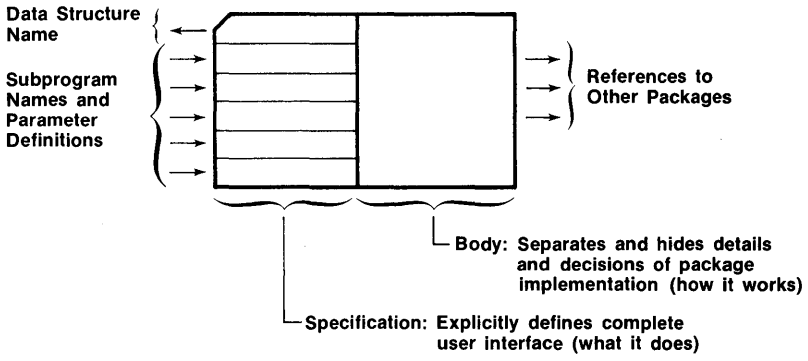


Figure 11.
General Form Of A Package

function that can be called to perform a service related to the package's data structure. In a banking application, for example, there might be a package that manages a data structure representing a bank account; this package might provide operations such as *open new account, close existing account, debit account, credit account, report account balance,* and so on. A file-handling package might provide *read, write,* and *close* operations. The (static) organization of a system's software is a network of packages that call each other's operations, as shown in figure 12.

The fundamental characteristic of a package is *localization.* One and only one module has responsibility for all operations on one and only one data structure. A data structure can be read or written only by the operations in its package; other modules may gain access to information in the data structure (or cause information in the data structure to be changed) only by calling these operations. Other modules know *what* a package will do when one of its operations is invoked, but all knowledge of *how* it works is hidden. This means that "how it works" can be changed without affecting users so long as "what it does" is preserved.

What are the consequences of basing a system on packages? First, development is simplified by the separation

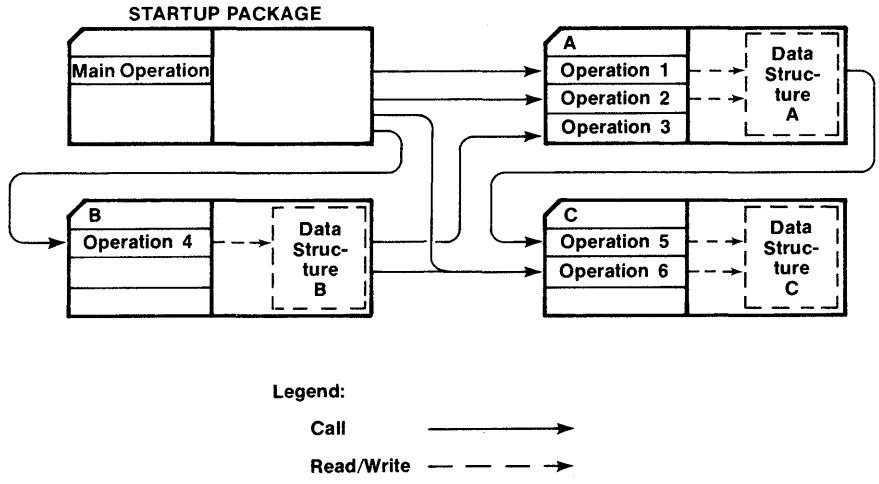


Figure 12.
Package-based System Organization

of package specifications ("what they do") from package bodies ("how they work"). The specification of a package is a contract between a package's implementors and a package's "clients." Once a specification has been established, clients and implementors can code in parallel; the specification decouples the use of a package from its implementation. Since clients depend only on a package's specification, a change to a package body (for example, to fix a bug or to improve performance) is guaranteed to affect no other module in the system. (Note that the Ada compiler checks to insure that caller/callee parameter lists match at *compile-time*, thus insuring that packages "fit" when they are linked together for testing.)

Second, if a data structure is found damaged, the damage can *only* be due to an operation in the single package that manages it. Third, a new package can be introduced into a working system with confidence, since its interaction with the rest of the system is limited to calls of existing package operations.

Most observers agree that the principal contributor to low programmer productivity is the inordinate amount of time that is devoted to program revision — fixing bugs,

enhancing services, or improving performance. Many large systems are in a state of constant change from the time development begins: new requirements are introduced, algorithms and data structures are found to be inefficient, "minor" misunderstandings are discovered and so on. Conventional techniques for decomposing systems into modules permit the correct operation of a module to depend on a data structure which can be altered by any number of other modules. A simple modification to such a data structure, or to a module that updates it, can easily trigger the modification of tens of other modules. By increasing module independence, package-based modular programming above all makes systems *changeable.*

MODULAR APPLICATIONS EXECUTIVE

Hardware engineers design microcomputer systems largely from standard chip- or board-level building blocks (disk controllers, memories, and so on). These components perform generic functions that are useful in widely different kinds of applications. This approach minimizes the amount of original design work, confining it to the truly application-specific aspects of the system.

The 432's iMAX executive is a kind of "component catalog" for 432 software engineers. Each iMAX component is an "off the shelf" Ada package that manages a basic resource (e.g., free memory) that most systems use, regardless of the application area. Electing to implement basic system services with iMAX packages reduces project risk since the most critical and elemental services (those which require intimate knowledge of the machine) have already been written, documented and tested. It also shortens time-to-market by permitting programmers to concentrate on the software requirements that are unique to the application.

Since iMAX is a collection of Ada packages (see figure 13), rather than a monolithic system, only the services that are

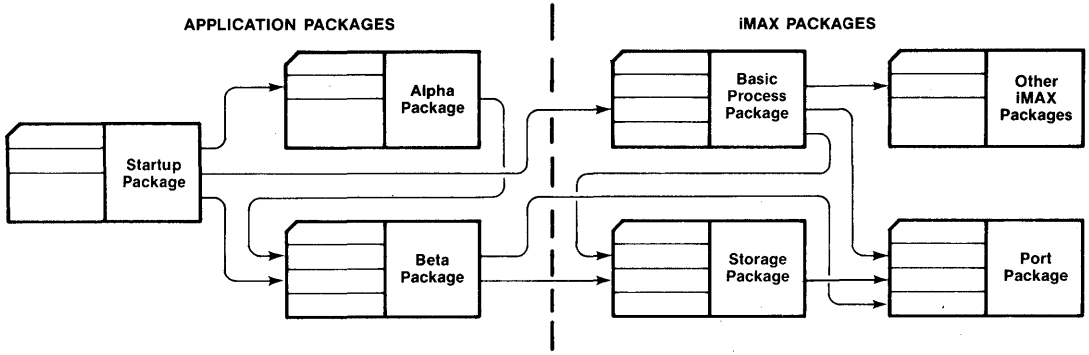


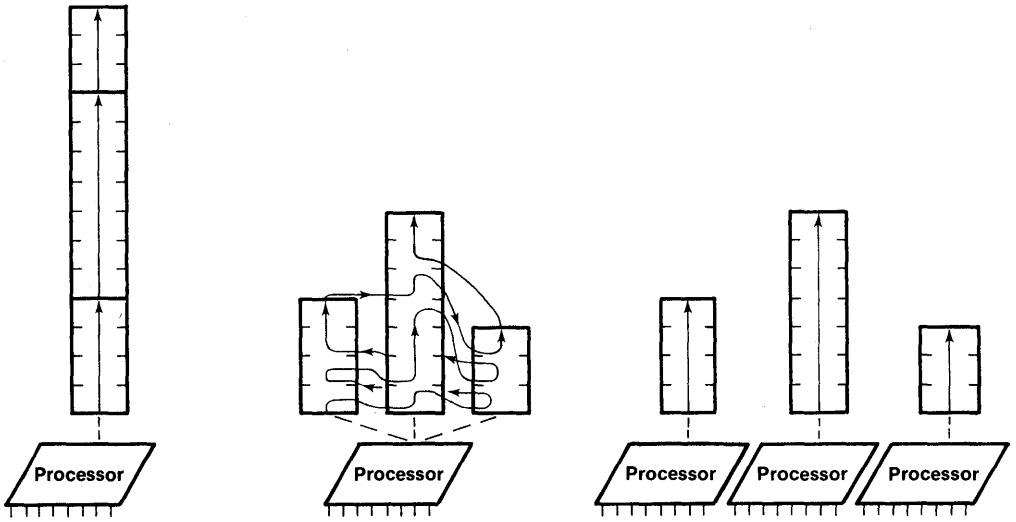
Figure 13.
iMAX Package-based Organization

germane to the application need be configured. No memory space or processor time is consumed by unneeded facilities. (Naturally, some iMAX packages use each other, so selection of one may implicitly select another.) Furthermore, users may add packages to iMAX, and, within limits, may also replace packages with equivalent local implementations. In sum, iMAX's package-based organization makes it remarkably adaptable to differing local needs.

CONCURRENT PROGRAMMING AND EXECUTION

Computers were invented to perform long calculations consisting of a great many steps executed in sequence, one after another. While computers are still used for calculation, their application has also broadened tremendously. Now it is rare to find a machine that is employed to execute programs sequentially one at a time. Instead, most computers are busy running many programs concurrently. The essence of concurrency is that more than one activity is underway — but not necesarily in actual execution — at the same time. Figure 14 contrasts sequential execution with the two types of concurrent execution, apparent and real.

It is useful to distinquish between concurrent *execution*, as described above, and concurrent *programming*. Concurrent execution can improve system throughput by



a. **Sequential Execution — Single processor executes one program to completion before beginning the next.**

b. **Apparent Concurrent Execution — Single processor is multiplexed among multiple programs; only one actually runs at a time.**

c. **True Concurrent Execution — Multiple processors execute multiple program in parallel.**

Figure 14.
Sequential And Concurrent Execution

increasing processor utilization. Most programs actually run in short bursts, executing some instructions then waiting for an "event" that must occur before processing can continue. Often the event is completion of an I/O operation or receipt of a message sent by another program. Rather than have the processor idle until the event occurs, the processor can switch to a ready program and run it. Multiplexing the processor among several programs keeps it busy and permits more work to flow through the system in a given period of time. Concurrent execution does not necessarily require attention from programmers; for example a multiprogramming operating system may multiplex a processor among independent programs that are unaware of each other's presence.

If a system supports concurrent execution, the added availability of a concurrent programming facility gives programmers the opportunity to simplify software design.

A system of concurrently running programs is a natural model of the "real world," particularly the world that microcomputers interact with. In the environment of the typical microcomputer-based product, many activities are underway at once: end users are entering data and requesting services, sensors are monitoring external conditions, controllers are adjusting equipment, and so on. Futhermore, these activities often interact with one another: one activity may produce data that is "consumed" by another, one activity may begin upon a signal from another and so on. A system's software is greatly simplified if its own structure reflects these natural activities and their interactions.

In general, support for concurrent programming has been limited to an operating system sandwiched between a sequential programming language and a sequential machine. Without language support, programmers have been forced to learn the idiosyncracies of an operating system in addition to their application language. Without machine support, it is very difficult for an operating system to efficiently multiplex the execution of concurrent programs, and even more difficult to provide efficient sharing of information while at the same time protecting programs from each other. Finally, without designed-in hardware support, it is extremely difficult to implement a system that provides real concurrent execution; in most computers true parallel execution is limited to I/O operations.

The 432 is specifically designed to support concurrent programming and concurrent execution — both apparent and real. Concurrently executing programs nominally run independently of one another; GDPs automatically multiplex themselves among ready programs, removing this overhead from operating system software. At the same time, concurrently executing programs may be designed to *cooperate* with each other; they may send messages, synchronize their execution and share data. Finally, the 432's transparent multiprocessing capability makes moving from apparent to real concurrent execution a simple matter of adding processors to the system.

Ada defines a unit of concurrency, the *task*, as a basic element of the language; a task is a program unit that may operate in parallel with other tasks. Ada gives programmers the ability to create tasks, activate and terminate them, pass information between them and synchronize their execution. All of this is done directly with high level language statements. Whether the concurrent execution of active tasks is apparent or real is purely a matter of the underlying hardware configuration; it does not need to be addressed in the code at all.

Since Ada's tasking facility is intended to be implemented on many different types of computers, its design has to some degree traded flexibility for ease of implementation. For example, it is difficult in Ada for one task to request a service of another and then continue running without waiting for the request to be acknowledged. The iMAX executive takes advantage of the 432's unique architecture to offer programmers a somewhat more general and efficient set of concurrent programming facilities. The executive also simplifies concurrent programming in languages which do not have built-in facilities; such facilities can be supplied in the form of a subprogram library that interfaces to iMAX.

Underlying both iMAX and Ada is a processor expressly designed for concurrent programming and execution. 432 general data processors automatically multiplex themselves among ready-to-run programs, with no software intervention whatever. The operations of sending and receiving messages are single hardware instructions, like add and subtract. These instructions execute rapidly because transmission requires only moving a one-word reference — the message "text" stays in place. Transmission is asynchronous and is buffered automatically by the hardware; a sender can produce messages at any time and at any rate and a receiver can receive them independently at its own convenience. "Overactive" senders, which might exceed the capacities of message buffers, are automatically controlled by the hardware.

Concurrent programming and execution extends to peripheral subsystems also. Since they are independent computers, peripheral subsystems can fully overlap I/O transfers with GDP operations. The same message-based protocol used in the central system to pass information between GDP programs is employed to send requests for I/O to programs running on peripheral subsystems.

REQUEST FOR READER'S COMMENTS

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

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

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

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

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

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

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

NAME _____ DATE _____

TITLE _____

COMPANY NAME/DEPARTMENT _____

ADDRESS _____

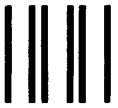
CITY _____ STATE _____ ZIP CODE _____

Please check here if you require a written reply. ☐

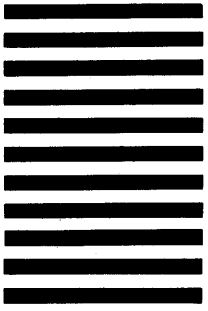
WE'D LIKE YOUR COMMENTS . . .

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

fold



NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES



BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 79 BEAVERTON, OR

POSTAGE WILL BE PAID BY ADDRESSEE

Intel Corporation
SSO Technical Publications Dept.
AL3-2-485
3585 S.W. 198th Ave.
Aloha, OR 97007

THE DEPENDABILITY CHALLENGE

It has always been important for microcomputer-based products to run dependably. Unlike most minicomputers and mainframes, micros usually find themselves installed far from personnel who can diagnose and repair a piece of hardware, not to mention fix a program bug. A microcomputer's "I/O devices" may include not only disks and terminals, but also furnaces, kidney machines, radar displays and other exotic and expensive equipment. A system failure can threaten equipment, organizations, and individuals.

A 32-bit microcomputer system intensifies the importance of dependable operation because it allows construction of products which represent unprecedented value to end users — so long as they work. Conversely, when an extraordinarily able machine — one that people rely heavily on — fails, it can suddenly become perceived as a tremendous liability. Thus, somewhat ironically, the *consequences* of a failure generally increase in severity with a product's utility. At the same time, the *likelihood* of a failure increases with the complexity of the system. High-utility end products, made possible by a 32-bit microcomputer system, will be correspondingly complex, at least internally. Developers of high-end products, then, face the difficult prospect of building systems that are both complex and dependable.

Dependable operation of both hardware and software components is a high priority in the design of the 432. Innovations in this area are based on a philosophy of avoiding run-time defects where possible, planning for the failures that will inevitably occur, and tightly confining the consequences of these failures. They include a floating point facility that greatly simplifies the development of reliable algorithms, error checking that detects many classical program bugs before they get into execution, an extensive run-time protection system, and processor self-checking for physical defects.

RELIABLE FLOATING POINT ARITHMETIC

Like most mainframes and some minicomputers, the 432 GDP provides a floating point facility that approximates the system of real numbers. Most floating point implementations, however, have proved difficult to use in practice. The straightforward algorithms developed by typical programmers often exhibit surprising, inconsistent, or inaccurate results, particularly when processing boundary values. As a result, many projects have had to either abandon floating point computation, or obtain the services of a numerical analyst.

The 432 implements the proposed IEEE standard for micro/minicomputer binary floating point arithmetic. The critical, time-consuming operations are implemented in GDP hardware and the remainder of the standard is supplied in an Ada package provided with the compiler. The proposed standard takes care to avoid past shortcomings and insures that floating point computation (which is inherently approximate) is as accurate and consistent as possible. In ordinary usage, results are rounded to the nearest representable value, and are rounded without bias in the case of ties (two values are equally near the true result).

The extended-precision temporary-real data type is provided expressly for holding intermediate results. Its extra range and precision practically eliminate the exceptions most commonly encountered in floating point arithmetic: overflow, underflow and harmful roundoff errors.

The 432's approach to floating point arithmetic greatly improves the reliability of calculations written by ordinary (numerically unsophisticated) programmers. It is an example of how the thoughtful design of hardware can contribute to the development of defect-free software, effectively eliminating one class of run-time error.

COMPILE-TIME CHECKING

Originally designed for military applications, one of Ada's primary goals is to promote the development of reliable programs. Two of its key features are strong typing and parameter checking.

Nearly a decade of wide experience with the Pascal language has verified the contribution that strong typing makes to program clarity and reliability. Ada borrows and improves upon this approach. As structured programming adds discipline to the expression of algorithms, strong typing adds discipline to the expression and use of data. It formalizes the good programming practice of introducing variables (and constants) before they are used; the introduction names the item and specifies its *type*. A few of the familiar types predefined in Ada are CHARACTER, INTEGER, FLOAT and ARRAY. Importantly, programmers may create new types which are appropriate for the application at hand. A data item's type defines its essential properties and its intended use in the program. (A more formal way of saying this is that an item's type specifies the values it may assume during execution, and the operations that are legal for those values.) Refer to figure 15 for some simple examples of typed variable and constant declarations in Ada.

Completely defining a variable's properties in one place improves program clarity, rather like introducing the team members and their positions at the start of a football broadcast. Besides informing the reader, typed declarations convey valuable information to the compiler. The compiler uses this information to insure that the *actual* use made of a data item matches its *intended* use. If a programmer defines two different types, say APPLE and ORANGE, the compiler will not permit variables of these two types to be added together, because they are not compatible. This consistency checking is the basis for characterizing Ada's type system as "strong."

Managing a large program with many variables is not a simple intellectual task; it demands methodical attention to detail. Under intense (i.e., normal) schedule pressures, "methodical" typically gives way to "expedient." When a

```
— — — — — — — — — — — — — — — — — —
— — use some predefined types
— — — — — — — — — — — — — — — — — —

increment:      integer;
avg_pages:      float;
page_no:        natural;
current_col:    natural := 1;
one:            character := '1';
print_width:    constant natural := 132;
page_depth:     constant natural := 60;
device_name:    string (8) := "SYSPRINT";

— — — — — — — — — — — — — — — — — —
— — define some new types
— — — — — — — — — — — — — — — — — —

type line       is string (print_width);
type page       is array (page_depth) of line;
header_page:    page;

— — — — — — — — — — — — — — — — — —
— — simple type-checking
— — — — — — — — — — — — — — — — — —

page_no := 1;       — — ok
page_no := one;     — — illegal
```

Figure 15.
Ada Strong Typing Examples

new programmer must modify someone else's program, it can be difficult for that person to understand the full implications of even a simple change. Strong typing is particularly valuable in these circumstances. In effect, it acknowledges human fallibility in stressful conditions; the compiler acts as a second reader that "walks through" the code during compilation checking for consistent usage. By cross checking stated intentions and actual behavior, strong typing catches many "clerical" errors before the program gets into execution.

Mismatched parameters are a classical source of error when many different people are developing interdependent modules. Ada carries the principle of strong typing beyond Pascal by checking subprogram parameters for consistency, even if the calling and called routines are *compiled separately.* Any discrepancy in the number of parameters, their types, or their order is detected at compile-time when it is simple and convenient to fix the problem. When a

simple mismatch is allowed to pass into execution, the resulting system behavior can be surprisingly difficult to trace back to the trivial source of the problem.

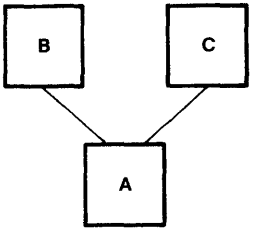
MODULE VERSION CHECKING

Another classical "bookkeeping" problem in developing, and especially modifiying, large systems is version mismatch. This is another situation in which a comparatively minor oversight can lead to disastrous consequences which are very difficult to diagnose. The basic problem is keeping track of the dependency relationships among a system's modules, so that when the interface (e.g., parameter list) of a module is changed, all dependent modules are also modified to match the new version. (This situation cannot be detected at compile-time. The compiler can follow the chain of dependency down from the module it is compiling to modules it *uses*, but not up to modules that use *it*; the compiler has no way of knowing which modules depend on the module being compiled.) Figure 16 illustrates a common scenario, and shows how the 432 linker detects the discrepancy before the (mis-)modified system gets into execution. As with type checking, this is a simple error to fix when it is caught early.

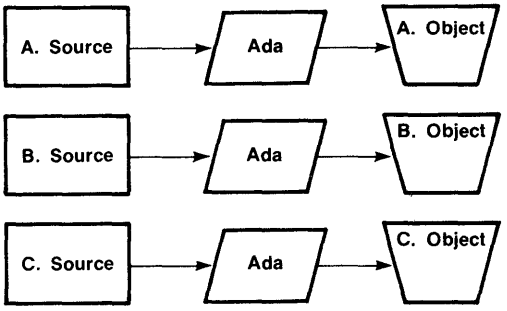
RUN-TIME PROTECTION IN HARDWARE

Debugging a large software system, particularly one with many concurrent activities, is extremely difficult. Only a fraction of the cases the system will encounter in actual execution can practically be tested. Complex timing interactions can make system behavior (and, more importantly, misbehavior) unrepeatable for purposes of diagnosing an error or testing a fix. In fact, it is impossible to positively demonstrate that a system *fully* works.

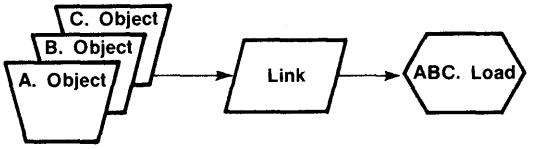
- Modules B and C depend on (use) Module A.



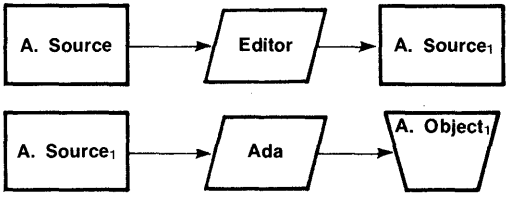
- The source modules are separately compiled.



- The object modules are linked together.



- An error is discussed in module A's **interface** (e.g., it requires a new parameter). A is modified and recompiled.



- The system is relinked; however, Link-432 discovers that modules B and C reference an old version of A. They must be updated to match A's new interface.

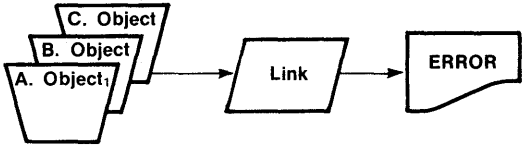


Figure 16.
Module Version Checking

As a practical matter, then, it is prudent to accept the inevitability of latent bugs in a complex system, and at the same time to insure that they do the least possible damage when they are executed. This is the basic rationale for a run-time protection system. Such a system *expects* errors and exceptions to occur during execution, detects them rapidly, and provides the information necessary to make an appropriate response. Like compile-time type checking, run-time protection represents the acknowledgment of fallibility and the determination to limit its impact.

An extensive facility for defining, detecting and handling run-time exceptions is inherent in the Ada language. The bulk of the 432's protection system, however, is implemented in hardware and is independent of Ada. It is built into hardware for four reasons. First, not all systems built with the 432 will use Ada exclusively; code generated by other translators must be prevented from interfering with the rest of the system, including Ada's "safer" code (even Ada permits "unsafe programming" under special conditions). Second, hardware-based protection is not susceptible to corruption (innocent or malicious) by programmers. Third, hardware protection is fast — software-based protection mechanisms can exact high performance penalties. Finally, hardware-based protection permits the construction of *dynamic systems* that can safely deal with new users, programs and devices that did not exist when the system was compiled. Hardware-based protection makes it possible, for example, to add a new service to a *running* system without jeopardizing the integrity of existing functions.

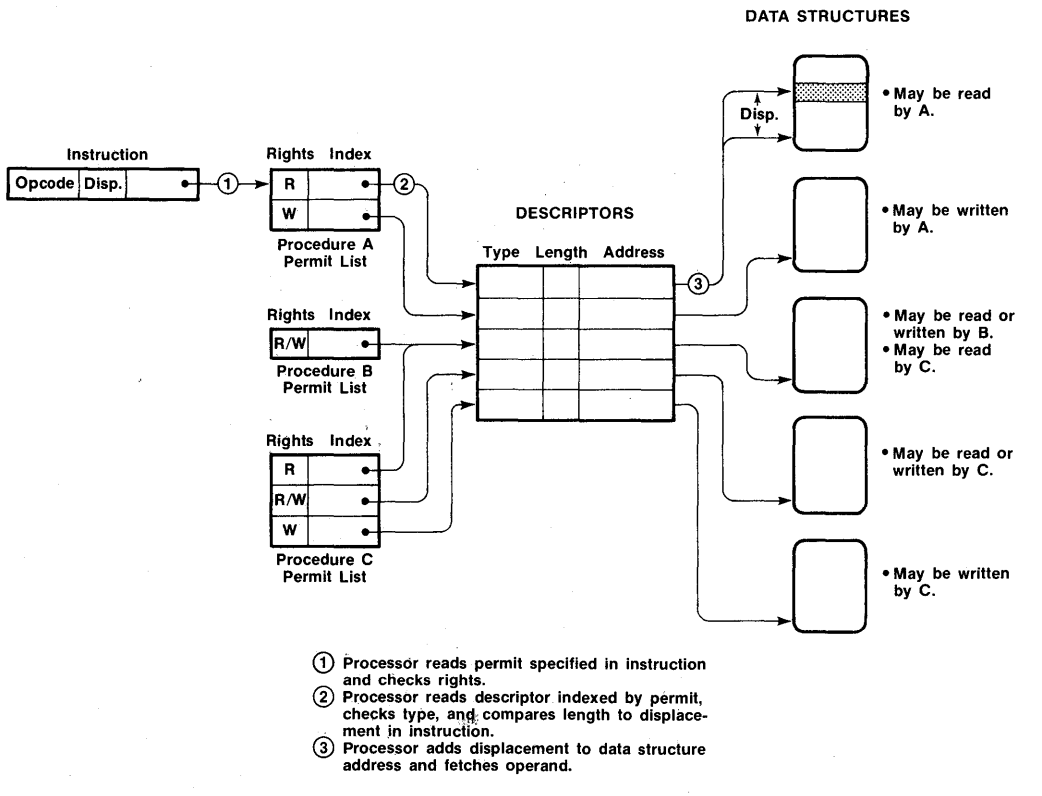
Designed into the hardware from its inception, the 432's protection system is efficient, precise and flexible; it applies to both computation and input/output operations. It is based simply on controlling the access of programs to information. The level of information to which protection is applied is the data structure, and the level of program to which access is controlled is the procedure (or function).

Underlying the protection system is the "need to know" principle: a procedure should have access to only the

information it requires to perform its function and nothing else. By "access" we mean both *what* information is addressable in any way, and *how* addressable information can be used. This principle is applied uniformly to "operating system" as well as "application" procedures; as a result, no procedure is more "privileged" than any other. The iMAX executive is protected from application procedures because they have no way to gain access to its data; the reverse is true as well, except as the application explicitly passes information to the executive.

In order to obtain protection without unduly sacrificing performance, the 432's protection system is integrated into the hardware's addressing mechanism. The overall approach is depicted in figure 17.

Figure 17.
432 Protected Addressing



All information in the memory space of the 432 is collected into data structures. There are many distinct *types* of data structures; some of them are predefined for use by the hardware and the executive, and others are defined by applications. Each data structure is represented by a descriptor in a central table, which is maintained by the processors. A descriptor contains the address of the data structure (432 instructions do *not* contain addresses), its length and its type. Centralizing all addresses in this manner makes it simple to move a data structure even when many procedures have access to it; changing the address in the descriptor causes all subsequent references to automatically pick up the new address.

Each procedure has a unique list of "permits" which completely defines the data structures it can address. Each permit contains a set of *rights* that specify how the procedure may access the corresponding structure. The fundamental rights are read-only, write-only, and read-write; other rights relate to specific types of data structures and grant/deny permission to perform certain operations. A permit also contains an index to the descriptor for the associated data structure.

A procedure's permit list is initially built when it is compiled, and it may expand during execution. For example, if the procedure creates a new data structure dynamically, the hardware adds a permit for the new data structure to the list. However, a procedure has no way of operating on its permit list as ordinary data; it is impossible for it to change the rights or descriptor index. Note that since each procedure has its own permit list, a called procedure does not inherit access to data structures from its caller, other than those explicitly passed as parameters. Thus, a program's instantaneous *access environment* changes from one procedure call to the next.

For each of its operands, a machine instruction contains a short encoded reference to the permit for the data structure containing the target data item. The instruction also specifies the displacement of the item from the beginning of the structure. Using the displacement, the

permit, and the descriptor, the processor calculates the actual physical address of the item to be operated on. In doing so, the processor also determines that the following are true:

- the procedure's rights for the data structure permit the operation that is being attempted;
- the data structure's type is consistent with the operation being attempted;
- the data item lies within the actual extent of the data structure.

If the hardware detects an attempted protection violation, it aborts the instruction before executing it. It also writes diagnostic information into a predefined data structure and passes control to a software routine which has been designated to handle the error. This automatic facility is called fault detection and reporting. (The hardware also faults if it detects a conventional exception during instruction execution, such as division by zero, and arithmetic overflow or underflow.)

Significantly, the 432's fine-grained protection system promotes the safe, flexible *sharing* of information between cooperating procedures running concurrently on the same or different processors. If one procedure wants to pass a data structure to another, it is only necessary to create a new permit containing the rights that are to be granted to the receiver and then *send* the permit — there is no need to move the data structure at all. The 432 has built-in instructions for performing these operations simply and efficiently. When two procedures hold references for the same data structure, one of them can temporarily prevent access by the other by *locking* the data structure (for example, during an update). The hardware automatically defers an attempt to gain access to a locked structure.

Note that the 432 interface processor uses the same mechanisms to insure that I/O operations do not violate the system's protection standards. From the 432's perspective, an I/O operation transfers the content of a data structure between a peripheral subsystem and central

memory. Software running on the peripheral subsystem directs the transfer, but the IP does all central system addressing. The same permit lists are provided for peripheral subsystem software, and the IP performs the standard consistency checks, fault detection and so on.

SELF-CHECKING PROCESSORS

Even a system that is apparently free of software defects will experience hardware failures during its life, particularly in infancy and in old age. Like a software bug, a hardware failure may cause a system to produce invalid information; such invalid information may be passed on to end users or may be "consumed" internally possibly bringing the system down. Dependable systems must therefore expect and cope with hardware as well as software failures. The 432 exploits the unique capabilities of VLSI components to allow construction of processor modules that detect their own malfunction.

The tolerance of a computer-based system to hardware faults can be viewed as a combination of its fault *coverage* and its *availability*. Fault coverage refers to the percentage of possible failures that the system is able to detect, and to the tightness with which the system localizes the source of a failure and confines its impact on the rest of the system. Availability is the inverse of downtime; it depends on the rate of fault detection and the time required to recover — to return to a usable condition, perhaps with reduced capability. It is important to note that fault tolerance is ultimately based on fault coverage. While a high-availability system may provide "non-stop" operation, without good fault coverage it may also be producing invalid information because it is insensitive to its own faults.

Applications vary in their need for fault-tolerance as shown roughly in figure 18. Each design must weigh the cost of hardware failures against the cost to build detection and recovery facilities into the system. Over time, however, the increased reliance on computer-based systems tends to drive most applications toward increasingly fault-tolerant operation.

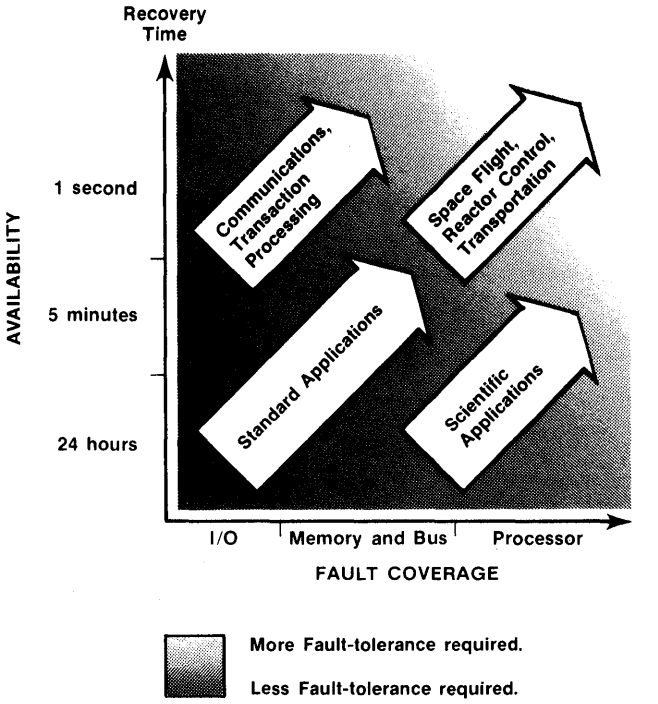


Figure 18.
Fault-tolerant Systems

Technology has responded to these demands with techniques for adding redundant codes to data such as parity, cyclic redundancy check (CRC), and error checking and correcting (ECC). These techniques detect data transformations and can be applied successfully to buses, I/O devices and memory. The System 432/600, for example, uses ECC memory, which detects all single- and double-bit errors and corrects single-bit errors; it also parity-checks each byte of the system bus.

Much less progress has been made in detecting processor failures at reasonable cost. Processors by definition transform data, and it is extremely difficult to design a coding technique that distinguishes *incorrect* transformations. Practically speaking, processor fault detection requires redundant machines and factors of cost and size

have precluded this approach for all but the most critical applications. A second factor that works against the development of redundant processors is the halving of reliability that occurs when the component count is doubled. Unless the single processor is extremely reliable, and recovery time is very fast, the doubling of the failure rate may reduce availability by more than the added fault detection is worth.

Taking advantage of the inherent high reliablility, small size and low cost of VLSI components, the 432 significantly expands fault coverage to include nearly instantaneous detection of processor failures. This is achieved simply by replicating identical processors. All GDPs and IPs have built-in circuits that permit two processors to be wired together into a single self-checking module (see figure 19). The module monitors itself cycle-by-cycle, performing every operation in parallel and comparing the results. If it cannot guarantee correct operation (the checker's result differs from the master's), the module automatically stops itself in the *next cycle* and notifies the rest of the system. The checking covers defects on the VLSI chips themselves, as well as mechanical flaws in bond wires, sockets and solder joints. No speed penalty is incurred when processors check each other in this manner; the self-checking module executes at the same rate as a single processor.

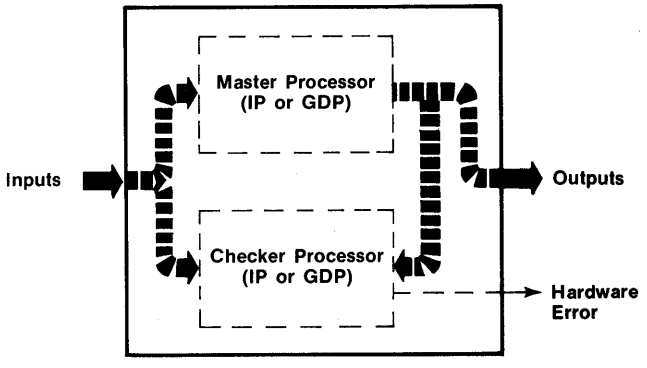


Figure 19.
Self-checking Processor Module

Transparent multiprocessing and self-checking processor modules permit the construction of systems that provide broad fault coverage and rapid recovery. Since a failed processor module identifies itself, it can be pulled from the system immediately and the system can then be restarted. If the module is an IP, one peripheral subsystem will be unavailable, but the system may still be able to provide a useful level of service; if the failed module is a GDP, the other GDP's can absorb the processing load automatically.

SUMMARY

A 32-bit microcomputer system represents tremendous computing *potential*: the raw material for constructing end products of unprecedented capability and value. The advent of this class of machine ushers in a new era of microcomputer-based system development in which unequalled opportunities are intimately linked with formidable challenges. In one stroke, a 32-bit machine lowers the barriers that have traditionally limited the application of microcomputers, and eliminates the constraints that have effectively shielded past projects from the difficulties of developing complex, large-scale, highly dependable systems.

Drawing on the experience of the mainframe community in developing comparable systems, the remarkable new capabilities of MOS technology, and important contributions from computer science and software engineering, the Intel 432 has been designed to profit from history rather than repeat it. It forges a new technology aimed at reducing the lifecycle costs of complex computer-based applications.

In its broad outlines, the 432 indeed supplies the functional capabilities of a 32-bit microcomputer: it delivers a very large address space, virtual memory support, and a rich variety of data types, instructions and addressing modes. What makes the 432 an extraordinary product family, though, are its innovative responses to the challenges inherent in bringing a complex, dependable product to market quickly and economically, and then supporting and enhancing that product over a period of years. Specifically, the 432 delivers:

- an instruction set that moves critical, time-consuming functions from software into hardware;
- a range of computational performance based simply on the replication of general data processors;
- a range of input/output performance based on the replication of identical or different peripheral subsystems;
- a standard high level language that is explicitly designed for constructing very large, highly reliable systems of concurrent software;

- an applications executive consisting of packaged components that can be configured, modified and extended to precisely match local needs;
- extensive pre-execution error checking to detect common software bugs before they get into execution;
- run-time protection in hardware to automatically detect software errors with the lowest possible overhead;
- the ability to build self-checking processor modules that immediately detect their own malfunction with no performance penalty.

Finally, the present design of the 432 anticipates future progress in MOS technology and application of the system to more specialized market areas by means of new processor types. It is a family of products that establishes the new leading edge today and will stay at the forefront for years to come.

U.S. AND CANADIAN SALES OFFICES

ALABAMA

Intel Corporation
303 Williams Avenue, S.W.
Suite 1422
Huntsville 35801
(205) 533-9353

ARIZONA

Intel Corporation
10210 N. 25th Avenue
Suite 11
Phoenix 85021
(602) 869-4980

CALIFORNIA

Intel Corporation
7670 Opportunity Road
Suite 135
San Diego 92111
(714) 268-3563

Intel Corporation
2000 East 4th Street
Suite 100
Santa Ana 92705
(714) 835-9642
TWX: 910-595-1114

Intel Corporation
5530 Corbin Avenue
Suite 120
Tarzana 91356
(213) 708-0333
TWX: 910-495-2045

Intel Corporation
3375 Scott Blvd.
Santa Clara 95051
(408) 987-8086
TWX: 910-339-9279
TWX: 910-338-0255

COLORADO

Intel Corporation
650 S. Cherry Street
Suite 720
Denver 80222
(303) 321-8086
TWX: 910-931-2289

CONNECTICUT

Intel Corporation
36 Padanaram Road
Danbury 06810
(203) 792-8366
TWX: 710-456-1199

U.S. AND CANADIAN SALES OFFICES

FLORIDA

Intel Corporation
1500 N.W. 62nd Street
Suite 104
Ft. Lauderdale 33309
(305) 771-0600
TWX: 510-956-9407

Intel Corporation
500 N. Maitland Avenue
Suite 205
Maitland 32751
(305) 628-2393
TWX: 810-853-9219

GEORGIA

Intel Corporation
3300 Holcomb Bridge Road
Norcross 30092
(404) 449-0541

ILLINOIS

Intel Corporation
2550 Gulf Road
Suite 815
Rolling Meadows 60008
(312) 981-7200
TWX: 910-651-5881

INDIANA

Intel Corporation
9101 Wesleyan Road
Suite 204
Indianapolis 46268
(317) 875-0623

IOWA

Intel Corporation
St. Andrews Building
1930 St. Andrews Drive N.E.
Cedar Rapids 52402
(319) 393-5510

KANSAS

Intel Corporation
9393 West 110th Street
Suite 265
Overland Park 66210
(913) 642-8080

MARYLAND

Intel Corporation
7257 Parkway Drive
Hanover 21076
(301) 796-7500
TWX: 710-862-1944

MASSACHUSETTS

Intel Corporation
27 Industrial Avenue
Chelmsford 01824
(617) 256-1800
TWX: 710-343-6333

MICHIGAN

Intel Corporation
26500 Northwestern Hwy.
Suite 401
Southfield 48075
(313) 353-0920
TWX: 810-244-4915

MINNESOTA

Intel Corporation
7401 Metro Boulevard
Suite 355
Edina 55435
(612) 835-6722
TWX: 910-576-2867

MISSOURI

Intel Corporation
502 Earth City Plaza
Suite 121
Earth City 63045
(314) 291-1990

NEW JERSEY

Intel Corporation
Raritan Plaza
2nd Floor
Raritan Center
Edison 08837
(201) 225-3000
TWX: 710-480-6238

NEW YORK

Intel Corporation
300 Motor Pkwy.
Hauppauge 11787
(516) 231-3300
TWX: 510-227-6236

Intel Corporation
80 Washington Street
Poughkeepsie 12601
(914) 473-2303
TWX: 510-248-0060

Intel Corporation
2255 Lyell Avenue
Lower Floor, East Suite
Rochester 14606
(716) 254-6120
TWX: 510-253-7391

NORTH CAROLINA

Intel Corporation
2306 W. Meadowview Road
Suite 206
Greensboro 27407
(919) 294-1541

U.S. AND CANADIAN SALES OFFICES

OHIO

Intel Corporation
Chagrin-Brainard Bldg., No. 300
28001 Chagrin Blvd.
Cleveland 44122
(216) 464-2736
TWX: 810-427-9298

Intel Corporation
6500 Poe Avenue
Dayton 45414
(513) 890-5350
TWX: 810-450-2528

OREGON

Intel Corporation
10700 S.W. Beaverton-Hillsdale Hwy.
Suite 324
Beaverton 97005
(503) 641-8086
TWX: 910-467-8741

PENNSYLVANIA

Intel Corporation
510 Pennsylvania Avenue
Fort Washington 19034
(215) 641-1000
TWX: 510-661-2077

Intel Corporation
201 Penn Center Boulevard
Suite 301W
Pittsburgh 15235
(412) 823-4970

TEXAS

Intel Corporation
2925 L.B.J. Freeway
Suite 175
Dallas 75234
(214) 241-9521
TWX: 910-860-5617

Intel Corporation
313 E. Anderson Lane
Suite 314
Austin 78752
(512) 454-3628

Intel Corporation
6420 Richmond Avenue
Suite 280
Houston 77057
(713) 784-3400
TWX: 910-881-2490

UTAH

Intel Corporation
3519 Lexington Drive
Bountiful 84010
(801) 292-2164

VIRGINIA

Intel Corporation
1501 Santa Rosa Road
Suite C-7
Richmond 23288
(804) 282-5668

WASHINGTON

Intel Corporation
Suite 114, Bldg. 3
1603 116th Avenue, N.E.
Bellevue 98005
(206) 453-8086
TWX: 910-443-3002

WISCONSIN

Intel Corporation
150 S. Sunnyslope Road
Brookfield 53005
(414) 784-9060

CANADA

Intel Semiconductor Corporation
Suite 233, Bell Mews
39 Highway 7, Bells Corners
Ottawa, Ontario K2H 8R2
(613) 829-9714
TELEX: 053-4115

Intel Semiconductor Corporation
50 Galaxy Boulevard
Unit #12
Rexdale, Ontario, M9W 4Y5
(416) 675-2105
TELEX: 06983574

INTERNATIONAL SALES OFFICES

AUSTRALIA

Intel Semiconductor Pty. Ltd.
Suite 2, Level 15, North Point
100 Miller Street
North Sydney, NSW, 2060
Tel: 450-847
TELEX: AA 20097

BELGIUM

Intel Corporation S.A.
Rue du Moulin a Papier 51
Boite 1
B-1160 Brussels
Tel: (02) 660 30 10
TELEX: 24814

DENMARK

Intel Denmark A S
Lyngbyvej 32F 2nd Floor
DK-2100 Copenhagen East
Tel: (01) 18 20 00
TELEX: 19567

FINLAND

Intel Finland OY
Sentnerikuja 3
SF-00400 Helsinki 40
Tel: (0) 5624455
TELEX: 123 332

FRANCE

Intel Corporation S.A.R.L.
5 Place de la Balance
Silic 223
94528 Rungis Cedex
Tel: (01) 687 22 21
TELEX: 270475

GERMANY

Intel Semiconductor GmbH
Seidlstrasse 27
D-8000 Muenchen 2
Tel: (089) 53 89 1
TELEX: 523 177

Intel Semiconductor GmbH
Mainzer Strasse 75
D-6200 Wiesbaden 1
Tel: 0812 6121 700874
TELEX: 04 186 183

Intel Semiconductor GmbH
Wernerstrasse 67
P.O. Box 1460
D-7012 Fellbach
Tel: (0711) 580082
TELEX: 7254826

Intel Semiconductor GmbH
Hohenzollern Strasse 5
3000 Hannover 1
Tel: (0511) 327081
TELEX: 923625

Intel Semiconductor GmbH
Vertriebsburo Dusseldorf
Ober-Ratherstrasse 2
4000 Duesseldorf 30
Tel: 49211 651054/55 56
TELEX: 8586977

INTERNATIONAL SALES OFFICES

HONG KONG

Intel Semiconductor Ltd.
99-105 Des Voeux Rd., Central
18F, Unit B
Hong Kong
Tel: 5-450-847
TELEX: 63869

ISRAEL

Intel Semiconductor Ltd.
P.O. Box 1659
Haifa
Tel: 972/452 4261
TELEX: 46511

ITALY

Intel Corporation Italia Spa
Milanofiori, Palazzo E
20094 Assago (Milano)
Tel: 8240006 or 8240706
TELEX: 315183 INTMIL

JAPAN

Intel Japan K.K.
Flower Hill-Shinmachi East Bldg.,
1-23-9 Shimachi, Setagaya-ku
Tokyo 154
Tel: (03) 426-9261
TELEX: 781-28426

NETHERLANDS

Intel Semiconductor Nederland B.V.
Oranjestraat 1
3441 Ax Woerden
Netherlands
Tel: 31-3480-112-64
TELEX: 47970

Intel Semiconductor B.V.
Cometongebouw
Westblaak 106
3012 Km Rotterdam
Tel: (10) 149122
TELEX: 22283

NORWAY

Intel Norway A/S
P.O. Box 92
Hvamveien 4
N-2013
Skjetten
Tel: (2) 742-420
TELEX: 18018

SWEDEN

Intel Sweden A.B.
Box 20092
Enighetsvagen 5
S-16120 Bromma
Tel: (08) 98 53 90
TELEX: 12261

SWITZERLAND

Intel Semiconductor A.G.
Forchstrasse 95
CH 8032 Zurich
Tel: 1-55 45 02
TELEX: 557 89 ich ch

UNITED KINGDOM

Intel Corporation (U.K.) Ltd.
5 Hospital Street
Nantwich, Cheshire CW5 5RE
Tel: (0270) 62 65 60
TELEX: 36620

Intel Corporation (U.K.) Ltd.
Dorcan House
Eldene Drive
Swindon, Wiltshire SN3 310
Tel: (0793) 26 101
TELEX: 444447 INT SWN

Intel Corporation, Ltd.
Beam Street 46/50
GB-Nantwich-Cheshire CW5 5LJ
Tel: 44 270626 560
TELEX: 36620

Intel Corporation
3065 Bowers Avenue
Santa Clara, CA 95051

Intel International
Rue du Moulin à Papier 51, Boite 1,
B-1160 Brussels, Belgium

Intel Japan K.K.
Flower Hill-Shinmachi East Bldg.
1-23-9, Shinmachi, Setagayu-ku
Tokyo 154, Japan

Printed in U.S.A. /881/100 K/C333/RRD

Intel Corporation
3065 Bowers Avenue
Santa Clara, CA 95051

Intel International
Rue du Moulin à Papier 51, Boite 1,
B-1160 Brussels, Belgium

Intel Japan K.K.
Flower Hill-Shinmachi East Bldg.
1-23-9, Shinmachi, Setagayu-ku
Tokyo 154, Japan

Printed in U.S.A./881/100K/C333/RRD