IBM RT Personal Computer Technology

Form No. SA23-1057

Foreword

IBM RT Personal Computer Technology is a collection of papers by the developers of the RT PC. These papers describe the innovative aspects of the RT PC—what we set out to build, how we built it, and how it works today. The papers were written by technical professionals for readers who are conversant with the vocabulary and concepts of computers and programming.

This book is a one-time statement by the developers for historical and background purposes. Although there are several overview articles that describe how the various components work together, the emphasis is on the novel parts of the RT PC system. *IBM RT Personal Computer Technology* is intended to supply the reader with an understanding of the things that make the RT PC unique, not to provide detailed descriptions of all of the elements of the RT PC system.

The papers in this book are not intended to replace IBM publications in describing the capabilities of the system components and how to use them. Keep in mind that the papers are for general technical communication purposes; they do not represent an IBM warranty or commitment to specific capabilities in the referenced products.

A variety of structures and levels of detail may exist in the papers because they were written as technical articles by different specialists. In order to preserve their authenticity and vitality, the papers have not been revised for consistency of style or method of presentation. These papers will not be updated to incorporate future developments.

This book is the work of many hands, but special acknowledgment is due to Bert Buller of the Hardware Architecture Group for coordinating the engineering articles, and to Herb Michaelson, Publications Consultant, for shaping both the book and the individual articles.

Frank Waters, Editor

Copyright

Copies of this book, SA23-1057, can be obtained from the local IBM branch office. IBM employees can order copies from Mechanicsburg.

Cover: An IBM RT Personal Computer Model 10 with a larger and somewhat faster ancestor in the background—an IBM System/370 Model 158 MP.

Preface

Introducing a new architecture to the computer marketplace is never done casually. The cost and effort of transition from one architecture to another must be justified by substantial advantages. It is always tempting to apply advances in technology to improving the performance of existing architectures. Ultimately, however, refinement is subject to the law of diminishing returns. Continuing advancement requires fundamental changes.

The hardware and software architectures that were originally created for personal computers had to accommodate the speed and size constraints of the processors and storage devices that were available at the time. Techniques that were known to be effective on mainframes and minicomputers were simply too costly to implement on personal computers. Our intent in designing the RT PC has been to use recent technological and architectural advances to avoid the structural limitations of earlier designs.

The IBM RT PC is a new synthesis of computer concepts. It combines:

- A very fast Reduced Instruction Set 32-bit processor for efficient execution of programs compiled from a higher-level language,

- a resource manager to provide virtual machine, storage, and I/O functions and to ensure data integrity and processing continuity,

- a multitasking, multiuser operating system that can be tailored to make the RT PC suitable for a variety of user requirements,

- a coprocessor feature that allows users to run programs written for the IBM PC without interfering with the normal operation of the RT PC,

- and a wide variety of displays, printers, communications adapters, and processing features,

- in a box that fits on or under a desk.

With the RT PC, the architectural sophistication of the personal computer has caught up with that of the mainframe. Perhaps more important, we have laid a foundation for more efficient exploitation of future advances in both hardware and software technology. The open-endedness of the resource manager and operating system at all levels means that we can easily take advantage of new applications, devices, and communications techniques.

The development of the RT PC system has been a multi-location effort. We have incorporated architectural advances from Yorktown Heights, technology innovations from Burlington, graphics peripherals and applications from Kingston, and engineering and programming developments from Austin. The RT PC has been the work of hundreds of individuals, both within and outside of IBM. The articles in this book describe some of their contributions. I want to convey to everyone involved my gratitude for their efforts and my respect for their accomplishments.

W. Frank King
Group Director of Advanced Engineering
Systems Development
Engineering Systems Products
Independent Business Unit

Contents

Reader's Guide

The *IBM RT Personal Computer Technology* book is divided into four parts. The first describes the main hardware elements of the RT PC system. The second section discusses the RT PC's 32-bit microprocessor. The third section covers the RT PC software and the fourth the PC AT coprocessor and AIX's PC DOS emulation functions.

A reader's guide, for a topical approach to this book, is as follows:

IBM RT PC Architecture and Design Decisions

G. Glenn Henry

Introduction

The architect of a new system must start by making a series of high-level decisions about the hardware and software structure. These early choices shape all of the ultimate details of the product. An understanding of the "reasons why" is important to the person who is trying the understand the resulting system. In this paper, I will give a very general overview of the structure of the RT PC and explain the rationale for each of the major decisions that dictated that structure.

Product Objectives

The original objective of the project that resulted in the RT PC was to build a high-function workstation with capabilities far beyond those of personal computers. Specifically required were:

- A high-performance 32-bit processor
- Large amounts of primary and secondary storage
- High-function virtual storage capabilities
- High-function APA display devices
- A full-function, multi-tasking operating system
- A high degree of usability
- A flexible, extendable, and open architecture.

In addition to these technical objectives, there were a number of important practical requirements relative to producing a successful product:

- Easy portability of existing IBM and non-IBM applications
- Easy migration of users and their applications and data from existing systems
- The ability to take advantage of I/O attachments and devices used on other systems
- Straightforward ways for other IBM and non-IBM development areas to add and modify system functions (i.e., an "open" system).

These challenging and, in some cases, conflicting objectives, when combined with available technology, shaped the system design as described in the following sections.

System Structure

Figure 1 shows the logical structure resulting from the objectives and design systems.

Figure 2 provides a more detailed look at the structure of the RT PC from a physical viewpoint. As you will see, the specifics of the system components reflect "something old, something new, something borrowed, and something Blue." The RT PC includes proven technologies and functions, innovations,

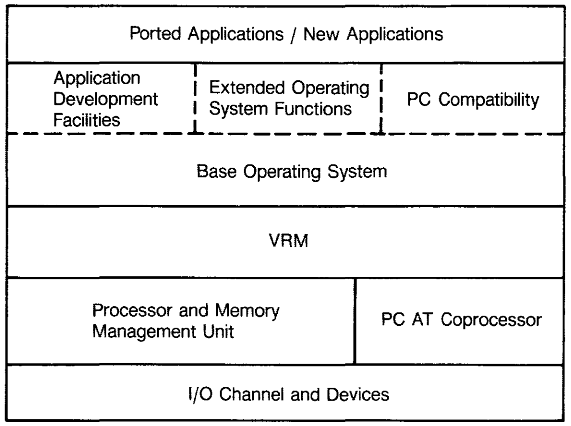

Ported Applications / New Applications
Application Development Facilities
Extended Operating System Functions
PC Compatibility
Base Operating System
VRM
Processor and Memory Management Unit
PC AT Coprocessor
I/O Channel and Devices

Figure 1 Logical Structure of the RT PC

interfaces that are compatible with other systems, and IBM-exclusive technology. It is this combination of approaches that allows the RT PC to meet its sometimes conflicting objectives.

I/O Channel and Devices

The I/O structure chosen for the RT PC was basically the PC AT 16-bit I/O Channel, with some performance improvements. This choice makes it possible to use most of the existing PC AT I/O attachment cards, while providing acceptable levels of I/O performance with native RT PC I/O devices. While providing compatibility with the PC AT, the RT PC I/O channel provides more usable capacity to I/O devices, since processor RAM is not connected to the I/O channel and the RT PC

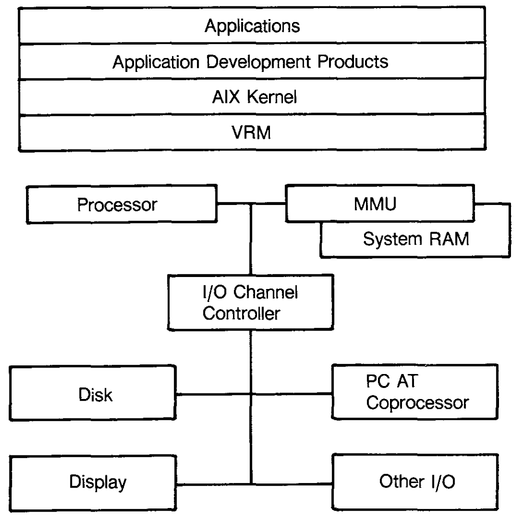


Figure 2 Physical Structure of the RT PC

I/O Channel Controller includes performance-assist features such as 32-bit "assembly" burst transfer.

Processor and MMU
The most critical choice was obviously that of the processor and associated Memory Management Unit (MMU). In spite of the obvious implications of the objective to ease migration of existing applications, we chose a new processor—the IBM Research/OPD Microprocessor (ROMP).

The major reasons for choosing the IBM ROMP were:

- It provides a full 32-bit architecture,
- with high performance (approximately 2 MIPs),
- using a Reduced Instruction Set Computer (RISC) architecture making it particularly suitable as a target for compilers (the preponderance of code expected to be executed on the RT PC was anticipated to be generated by compilers),
- on a single chip for a low-cost, high-performance solution,
- with an associated MMU chip providing advanced virtual storage capabilities,
- and finally, there was no vendor microprocessor available with the full set of these capabilities.

The IBM ROMP is a single-chip derivative of the 801 processor project of IBM Research and thus benefits from the processor architecture analysis and advanced compiler architecture design activities associated with the 801. In addition, the IBM ROMP has special features such as "Load Multiple" and "Store Multiple" that extend the RISC architecture approach to provide increased efficiency and performance in a microprocessor implementation.

The virtual storage functions provided by the MMU chip are very powerful. For example, the 32-bit processor address is extended to a 40-bit virtual address with a high-performance, hardware-managed "inverted" page table translation approach. This significantly reduces the size of the page tables for the large virtual address space while providing very fast virtual-to-physical address translation. Even today, I do not know of another MMU that provides this level of function and performance.

Virtual Resource Manager
When designing the software structure for the RT PC, we decided to build a Virtual Resource Manager (VRM) to control the real hardware of the machine. The VRM presents operating systems with a Virtual Machine Interface (VMI) that not only conceals the complexities of virtual memory management and numerous I/O device types, but provides the operating system with a significantly more powerful set of functions than are available on the bare machine. It is therefore not accurate to think of the VRM as a pure hypervisor, like VM/370. The VMI is, in effect, a higher-level machine to which guest operating systems can be converted. We considered the VRM necessary because:

- The operating system base that we wanted to use for the RT PC was not built to run on a computer with virtual memory, did not provide real-time I/O capabilities, and didn't provide dynamic install and configuration functions. We decided to provide these important functions "under" the operating system, as opposed to making extensive modifications to the kernel of the existing operating system. For example, the VRM provides a very fast preemptive interrupt-based I/O structure, virtual storage management functions, and dynamic loading and binding of I/O device drivers. This allows, for example, the operating system kernel to be paged and complex, multi-tasking I/O device drivers to be implemented.
- We wanted to achieve a higher degree of program isolation from the hardware details than is possible with the current personal computer operating systems. That is, the VRM resembles BIOS on the IBM PC, but it provides a much higher level of hardware independence. For example, generic device classes are supported at the VMI, allowing high levels of device transparency and I/O redirection.

- We wanted to allow hardware coprocessors to execute concurrently with ROMP without making major changes to the existing base and with minimal overhead for resource allocation and management (see PC Compatibility, below).

- We needed to isolate the development of the RT PC software from changing hardware characteristics during the development process. While this was an "internal" IBM requirement, the success of the VRM in meeting this goal validates its architecture and implementation features relative to providing high levels of hardware transparency to the user software.

Consistent with the objectives for a flexible and open system, the VRM provides complete facilities for the user to implement and install code in the VRM. That is, the detailed hardware structure is isolated from software in the preponderance of cases where that is desirable, but conversely, all hardware details are available to user functions that require them.

Operating System and Extensions
As the base for the RT PC's Advanced Interactive Executive (AIX[1]) operating system, we chose AT&T's UNIX[2] System V[3]. We chose UNIX because it provides considerable functional power to the individual user, provides multi-user capabilities where needed, is open-ended, and has a large user and application base. We concluded that AT&T's System V suited our purposes better than alternative UNIXes because of the larger number of applications that had been built to run on that base, as well as for a variety of practical reasons.

[1]"Aix" is a trademark of International Business Machines Corporation.

[2]Trademark of AT&T Bell Laboratories.

[3]The UNIX component of AIX was developed by IBM and INTERACTIVE Systems Corporation. The UNIX component is based on INTERACTIVE's IN/ix, which is based in turn on UNIX System V, as licensed by AT&T Bell Laboratories. (IN/ix is a registered trademark of INTERACTIVE Systems Corporation.)

In choosing UNIX, however, we accepted the need to make significant extensions and enhancements to meet the needs of our expected customers and target applications. This is, of course, the classical trade-off between choosing an existing software system for its pragmatic characteristics versus developing a new system with (hopefully) fewer deficiencies but limited applications and user familiarity. We chose to start with UNIX and fix the deficiencies while retaining upward compatibility for all UNIX System V interfaces.

Some of the major enhancements made were:

- A Usability package to provide easier access to the capabilities of the UNIX command language and to simplify the implementation of full-screen dialogs

- Multiple, full-screen virtual terminal support to permit a single user to run several interactive applications concurrently, time-sharing the console display

- Enhanced console support including extended ANSI 3.64 controls, color support, sound support, and mouse support

- An indexed data management access method that is integrated into the base UNIX file system structure (this allows UNIX system utility functions such as "cp" to transparently operate on composite data management objects consisting of an index file and a data file)

- Extensions to exploit use of the powerful virtual storage support; in particular, mapped file support which allows an application to "map" a file into a 256 megabyte virtual address space, and access it with loads and stores, versus reads and writes (a derivative is used by the system to provide mapped text segment support, allowing paging "in place").

- Enhanced signals to allow flexible exception-condition handling

- A variety of floating point support functions

- Simplified installation and configuration processes.

PC Compatibility
In addition to UNIX application portability, our original objective of easing user and application migration required a high level of compatibility with the IBM PC family. This was provided by:

- An IBM PC AT hardware coprocessor that includes an Intel 80286 along with associated hardware to provide a high level of PC AT hardware compatibility

- An IBM PC DOS "shell" on the AIX Operating System allowing DOS command syntax and semantics to be used to invoke AIX functions

- IBM PC compatibility modes in the RT PC BASIC and Pascal compilers, providing IBM PC BASIC and Pascal-compatible functions

- IBM PC diskette access utilities and access methods.

One of our key technical decisions relative to compatibility was to allow the PC AT coprocessor to execute PC programs *concurrently* with ROMP programs, sharing system resources such as main storage and the system console. This unique capability is provided by a combination of the coprocessor hardware card and the VRM, which manages the allocation and sharing of resources in such a way that the coprocessor's concurrent execution is transparent to the operating system and application programs. For example, the VRM allocates the console keyboard to either the coprocessor or the ROMP and monitors keystrokes for a "hot key" sequence signalling a need to switch to the other processor. In a similar fashion, other system resources are managed so that the coprocessor applications seem to execute in a virtual terminal just as the ROMP applications do.

Application Development Facilities
For the initial release of the RT PC, we placed a high priority on providing function to facilitate application development. We extended the already rich UNIX application development support functions with:

- a host-compatible SQL data base manager providing both an API and a full-screen user interface,
- a FORTRAN compiler, BASIC compiler and interpreter, and Pascal compiler (in addition to the C compiler and assembler included in the base operating system),
- a "net BIOS" set of facilities for access to the IBM PC Network,
- and many base operating system extensions: message services, shared segment manager, etc.

The Result
We believe the design choices presented here and the specific designs highlighted in the following papers allow the IBM RT PC to meet its original objectives. Further, these directions provide the architectural and design base for improvement with minimal disruption as technology progresses.

Hardware Description

P.D. Hester, J.T. Hollaway, and F.T. May

Introduction

Design Philosophy
The IBM RT PC system hardware was designed with the following basic philosophy in mind.

- A new family of workstation systems should be based on the most recent advances in microcomputer technology.
- An architecture should be established to ensure:
  - — The effective integration of a 32-bit virtual memory microprocessor with existing and new 8-bit and 16-bit I/O adapters
  - — The effective addition of I/O devices and adapters as technology trends progress
  - — The attachment of coprocessors for compatibility and performance enhancements
  - — The ability to incorporate user-installable performance enhancements over the life of the product
- A strong relationship should be maintained with the IBM Personal Computers.
- The product should allow customer setup and service.
- The initial product offering should clearly demonstrate the long-range potential of the design.

Hardware Summary
The RT PC workstations have consoles that contain the electronics, storage devices and power supply. They cable attach to the display, keyboard and other optional devices to meet the configuration requirements for the customer's applications. The IBM RT PC is available in two basic packages: the IBM 6150 is a floor-standing unit and the IBM 6151 is a desk-top console similar in size to the IBM PC AT. The 6150 Model 25 provides the maximum extendability, but most options are available on all models.

The workstations have a wide range of standard and optional hardware components. Data storage is provided on 5-1/4-inch hard disks and diskettes. A large system board is used to package the base electronics and card slots in each model. A number of cards and adapters have been designed specifically for the RT PC, and the I/O channel slots are designed so that many existing IBM PC and PC AT cards can also be used. Operator input can be made with a 101-key keyboard and an optional two-button mouse or tablet pointing device.

In addition to the standard I/O channel slots on the system board, there are unique slots for each of the 32-bit system components. The new IBM-designed ROMP 32-bit microprocessor and its corresponding Memory Management Unit (MMU) are packaged on a processor card which comes with each model. There is also a separate 32-bit slot for an optional floating point accelerator card. Two other dedicated slots are provided for system memory cards.

The unique ability to execute both IBM PC and IBM PC AT programs concurrently with native RT PC programs is provided by an optional coprocessor card which plugs into one of the I/O slots. Other coprocessor options provide for faster performance with additional PC AT memory cards and a math coprocessor chip.

The system memory is packaged on 1- and 2-megabyte cards which plug into two dedicated memory slots and provide expansion of models up to 3 and 4 megabytes. The hardware architecture allows for addressing up to 16 megabytes of real memory.

A wide variety of display subsystem options is available. In addition to existing IBM PC displays and adapters, three new offerings are available with monochrome and color APA features. Also, computer-aided design applications can be run using a serial link adapter card attached to an IBM 5085 Graphics Workstation in a host-based network.

Hardware Architecture

The RT PC system combines a 32-bit microprocessor (ROMP) and Memory Management Unit (MMU) with a standard 16-bit I/O channel. The system is partitioned so that the 32-bit system components operate independently from the 16-bit I/O channel. This approach provides both high-performance, 32-bit processing and compatibility with standard 16-bit I/O adapters. The RT PC hardware structure is shown in Figure 1.

The RT PC utilizes the IBM-developed ROMP microprocessor and corresponding MMU packaged on a processor card which plugs into the system board. The ROMP implements a Reduced Instruction Set Computer (RISC) architecture with 118 instructions, 16 32-bit general-purpose registers, and a full 32-bit data flow for both addresses and data. Most register-to-register operations execute in one 170-nanosecond cycle. Performance is typically 1.5 to 2.1 MIPs, depending on the instruction mix.

The MMU implements a "single level store" address translation architecture which converts the 32-bit system address to a 40-bit (1 terabyte) virtual address for translation. Internal translation buffers within the MMU convert the 40-bit virtual address to a 24-bit (16 megabyte) real address. Hardware is also provided in the MMU to automatically reload the translation buffers from main memory page tables as required. The MMU also contains the ECC logic for system memory, and some of the control logic for system memory and the IPL and power-on self test ROM.

Details of the ROMP microprocessor and MMU architecture are described by Hester, et

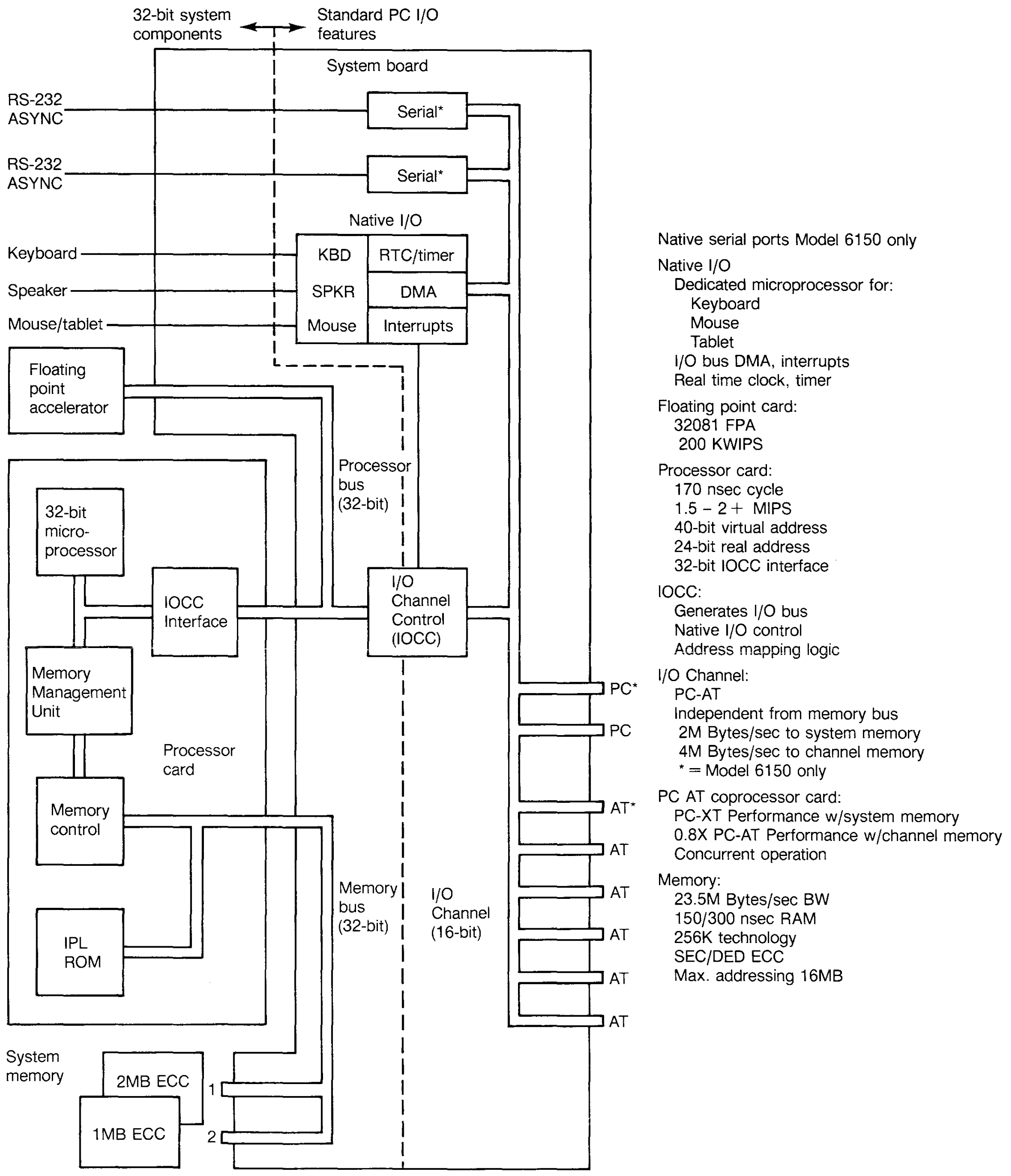


Figure 1 RT PC System Architecture

al.[1]. Chip implementation details are described by Waldecker, et al.[2].

Both the ROMP microprocessor and the MMU are custom designed VLSI components using an IBM 2 micron NMOS process. Both components are packaged in a pin grid array package on a 36-millimeter ceramic substrate. The ROMP contains approximately 45,000 devices on a 7.65 x 7.65 millimeter chip, with the MMU containing approximately 62,000 devices on a 9.0 x 9.0 millimeter chip.

In addition to the ROMP and MMU, the processor card (see Waldecker, et al. [3]) contains logic to adapt the 32-bit packet-switching microprocessor channel to an asynchronous 32-bit processor channel connected to the optional floating point accelerator card and the system board I/O Channel Converter (IOCC). A dedicated memory channel is also generated from the MMU for connection to the system memory cards. Five IBM technology bipolar gate arrays of approximately 300 gates each and vendor TTL logic are used for interfacing to the processor channel and memory channel.

Clock generation for the microprocessor, memory management unit, and system memory is provided on the processor card. Independent clock generation is provided on the system board for I/O channel timing. This makes it possible for higher performance processor cards and system memory cards to be supported as technology permits, without affecting I/O channel timing.

The optional Floating Point Accelerator (FPA) card attaches to the 32-bit processor channel and provides improved performance for floating point applications. This card utilizes a National Semiconductor NS32081 Floating Point Unit and operates independently of the ROMP. Multiple floating point register sets are provided for rapid context switching. Performance is approximately 200,000 Whetstone instructions per second. Various aspects of the FPA are covered by Smith [4].

Two dedicated slots are provided for system memory, which attaches to the processor card through the memory channel. The memory channel consists of an independent 40-bit data bus and 24-bit address bus. The data bus includes 32 bits of data and 8 bits of error correcting code (ECC). The RT PC ECC allows automatic detection and correction of all single-bit system memory errors, and detection of all double-bit errors. The 24-bit address bus is capable of addressing up to 16 megabytes of system memory.

Standard 256K RAM technology is used on the system memory cards, with a minimal amount of support logic. All memory timing, control, and ECC functions are provided by the processor card. System memory is two-way interleaved on each memory card, with one bank containing only even addresses and the other bank containing only odd addresses. This interleaving technique, combined with 150-nanosecond access time RAMs, provides a system memory bandwidth of 23.5 megabytes per second (4 bytes every 170 nanoseconds). Details of the memory cards are described by Rowland [5].

The system board contains all of the channel conversion functions to adapt the 32-bit ROMP Storage Channel (RSC) to a PC AT-like I/O channel. I/O channel support functions such as an interrupt controller, DMA controller, and real time clock and timer are also provided on the system board. Timings, address assignments, interrupt assignments, DMA assignments, and related functions of the RT PC I/O channel were designed to be as compatible as possible with the PC AT I/O channel. In addition, new features such as burst and buffered DMA and shareable interrupts were added to improve the channel performance and usability. The IBM 6151 provides one 8-bit PC slot and five 16-bit PC AT slots. The IBM 6150 provides two 8-bit PC slots and six 16-bit PC AT slots. Timing and performance of the I/O channel are the same in all models. Phelps and Upton [6] discuss various characteristics of the RT PC I/O channel.

In addition to the channel conversion functions, the system board contains a programmable translation control facility to support accesses from adapters on the I/O channel to system memory. A separate dedicated microprocessor is provided to handle the keyboard, speaker, mouse, and tablet interface. The IBM 6150 also includes two built-in RS-232 serial ports with DMA capability for attaching terminals, printers, or other I/O devices.

The optional Intel 80286 based coprocessor card plugs into an I/O channel slot and provides compatibility with PC and PC AT programs. In addition to the 80286 and optional 80287 math coprocessor, this card contains control logic that intercepts 80286 accesses to selected I/O addresses. A combination of this logic and system software allows sharing of system I/O adapters such as displays, keyboards, and files. Alternately, system software can program this logic to allow direct coprocessor access to private I/O adapters. Appropriate mapping is also provided by system software that allows PC applications written for the PC monochrome or color adapter to run on an RT PC using a native APA display. Operation of the coprocessor card is described by Irwin [7].

PC programs for the coprocessor can be stored either in system memory or in dedicated, I/O channel-attached memory. Coprocessor performance is typically that of a PC when executing programs in system memory and about 80 percent that of a PC AT when using I/O channel-attached memory.

I/O Devices
The selection of I/O devices was made by considering technology trends, the requirements of evolving applications, system performance, and physical power and packaging constraints. The generic set of devices required for electronic workstations is well established in the industry. There are a number of vendors that specialize in each of these devices and are very competitive in advancing the state of the art in their respective areas of the industry. The architecture of the RT PC system allowed the designers to select the preferred devices to meet the anticipated marketing opportunities as the project proceeded and the requirements changed. A few examples are outlined below.

Hard File Subsystem
The major decision in the selection of hard files was to use the 5-1/4-inch form factor. The size of the desk-top and floor-standing consoles is most directly affected by the size of the files. The technology was moving toward 5-1/4-inch files even though the 8-inch files were still improving in capacity and cost. The main reservation in the selection of these files was the average access time, which is in the range of 40 milliseconds versus 25 to 30 milliseconds for 8-inch files. Transfer rates are the same, at 5 megabits per second. System performance work was done to understand and compensate for this difference.

The capacity of the files to make available in the models also varied as the project advanced. Files as small as 10 to 20 megabytes were seriously considered, but the final system design point required the larger capacities of 40 and 70 megabytes. The combination of these larger files, and the option to have a model with three files, substantially expanded the range of potential applications that can be adapted to the product.

Another example of the flexibility of the design approach to the file area is illustrated by the fact that the disks and diskettes are attached by the use of the IBM PC AT Fixed-Disk and Diskette Drive Adapter, an existing card in the IBM PC product family.

An external streaming tape drive and a separate adapter card that attaches to the RT PC I/O channel are available as an option. The streaming tape unit provides a capacity of 55 megabytes using a standard 1/4-inch tape cartridge.

Display Subsystems
The display subsystem is generally the most obvious I/O device to the users of electronic workstations. The characteristics of the display are described by the parameters of size, number of picture elements (PELs), PEL density, monochrome and color, and front-of-screen performance. Substantial cost differences exist among these parameters. The applications selected by the users determine which of these parameters affect their choice of display subsystem. Details of the various displays and adapters are described by St. Clair [8].

The design decision was to offer a wide variety of display options and ensure that the architecture allowed for future options to be provided as the applications evolve. The RT PC can meet the needs of a wide variety of users, so their display needs are expected to be diverse.

The RT PC system provides for the attachment of existing adapters and displays of the IBM PC products. Specific IBM PC displays which have been tested for announcement are the Monochrome Display and the Enhanced Color Display with their respective adapters.

The new displays and adapters provide direct processor access to a 1024 x 512 bit map with a display viewing area of 720 x 512 PELs. Hardware assist provides for text and graphics alignment to the PEL level. This design point was determined to be a good trade-off between cost and total screen PELs for both monochrome and color applications.

A higher function monochrome display subsystem provides a larger viewing area of 1024 x 768 PELs and extensive hardware assist for high-speed vector-to-raster conversion from a vector list buffer. This design point is also considered to be a good trade-off between cost and total screen PELs.

A full 1024 x 1024 color display with existing advanced computer-aided design, CAD, applications is provided by the ability to use a serial link adapter card to attach the RT PC system to an IBM 5085 Graphics Workstation in a host-based network.

Mechanical And Electrical Packaging
Two different physical packages were selected for the RT PC system. The IBM 6151 package is very similar to the IBM PC AT desk-top configuration. The IBM 6150 has a floor-standing console to provide room and power for more adapter cards and files.

The key requirements for the packaging were set in compliance with the basic design philosophy mentioned above. One was to provide for the use of existing IBM PC cards, which established minimum physical dimensions and power requirements. File capacity requirements to support the addressing capabilities of the processor dictated the need to have models that can hold multiple files. A system board is used to mount connectors for the various cards that make up the base models and to provide extra slots for options.

Equally important were the requirements to design the product for automated manufacturing. This included limiting the number of separate subassemblies, establishing a standard packaging form for each of the subassemblies delivered to the manufacturing line, minimal use of internal cables for interconnections, and selection of one standard card size for all cards. Details of the manufacturing process are described by Bartlett, et al.[9].

Quality And Reliability
Many design and manufacturing decisions during RT PC development were made to ensure a high quality and reliable product. These included design for automation concepts in the original design, development of an automated manufacturing process, component selection process, analysis of early life failures, and automated error logging during system run-in testing.

A diagnostic program based on an expert system (see Burns and Williams [10]) is provided to aid the customer in diagnosing system problems. This program tests all RT PC system board functions and all I/O adapter functions. Major considerations in the development of the diagnostic program were providing a user interface with simple selection of desired tests and reporting test results in a concise manner. These characteristics were considered mandatory for a machine with customer setup and service.

Both the RT PC design and manufacturing processes have resulted in the ability to produce a complex workstation in high volumes. The component selection and qualification process, failure analysis, and run-in testing allowed supporting a 12-month warranty.

Conclusion
The RT PC system was designed to bridge the gap between the personal computer products introduced during the past few years and emerging advanced 32-bit workstations with extensive virtual memory management facilities. These workstations will become the basis of computing systems that have extensive storage, display and communications requirements to satisfy new applications as they evolve. Specifically, the RT PC:

- Introduces an IBM-developed, high-performance, 32-bit RISC architecture with virtual memory.
- Combines the new 32-bit features with a standard PC I/O channel.
- Provides an optional PC coprocessor for compatibility with existing PC application programs.

The development cycle for the product was executed during a period of continual change in technologies and design specifications, so an architecture was defined to provide consistency of design decisions in this environment. The architecture was tested on numerous occasions and allowed for needed design changes that preceded the initial product announcement.

The use of an open architecture similar to the earlier IBM PC products allows for extension by anyone who chooses to develop hardware attachments and applications that enhance the features of the base machines. This approach continues to be successful in personal computer systems and should be successful in the more advanced workstation-oriented systems.

Numerous enhancement possibilities are obvious to designers and users of this new class of advanced product. The architecture is capable of supporting increased memory capacity, higher capacity files, higher performance displays, other local area networks, higher speed host attachments, and other coprocessors. Enhancements should follow as time and new technologies allow interested companies in the computer industry to respond to the business opportunities.

Some enhancements may be limited by the I/O devices, or by the I/O channel characteristics. Others may be limited by computational speeds in the main processor, the IBM PC AT coprocessor, or the floating point accelerator. Performance requirements of new applications will reveal these constraints. Potential solutions to many anticipated future requirements have already been defined by the development team.

It is hoped that the decisions made during the RT PC system development process will survive the test of time, that the technologies provided by the product will satisfy the needs of developers, and that the applications built on the product will meet the needs of users ranging from technical professionals to office workers.

References

1. P.D. Hester, Richard O. Simpson, Albert Chang, "The RT PC ROMP and Memory Management Unit Architecture," *IBM RT Personal Computer Technology,* p. 48.
2. D.E. Waldecker, C.G. Wright, M.S. Schmookler, T.G. Whiteside, R.D. Groves, C.P. Freeman, A. Torres, "ROMP/MMU Implementation," *IBM RT Personal Computer Technology,* p. 57.
3. D.E. Waldecker, K.G. Wilcox, J.R. Barr, W.T. Glover, C.G. Wright, H. Hoffman, "Processor Card," *IBM RT Personal Computer Technology,* p. 12.
4. Scott M. Smith, "Floating Point Accelerator," *IBM RT Personal Computer Technology,* p. 21.
5. Ronald E. Rowland, "System Memory Cards," *IBM RT Personal Computer Technology,* p. 18.
6. Sheldon L. Phelps and John D. Upton, "System Board and I/O Channel For The IBM RT PC System," *IBM RT Personal Computer Technology,* p. 26.
7. John W. Irwin, "Use Of a Coprocessor For Emulating The PC AT," *IBM RT Personal Computer Technology,* p. 137.
8. Joe C. St. Clair, "IBM RT PC Displays and Adapters," *IBM RT Personal Computer Technology,* p. 31.
9. Charles W. Bartlett, A.V. Burghart, George M. Yanker, "Manufacturing Innovations to Increase Quality and Reduce Cost," *IBM RT Personal Computer Technology,* p. 40.
10. Nancy A. Burns, C. Edward Williams, "Use Of Artificial Intelligence To Diagnose Hardware," *IBM RT Personal Computer Technology,* p. 35.

Processor Card

D.E. Waldecker, K.G. Wilcox, J.R. Barr, W.T. Glover, C.G. Wright, H. Hoffman

Introduction

The processor card provides the central processing and memory management functions of an IBM RT PC system. It interfaces with memory cards [1] and I/O hardware on the RT PC system boards [2] in a manner which readily supports future enhancements of memory cards or the processor card itself. The card contains the ROMP processor and MMU memory management chips [3] plus support functions including ROM and internal clock generation.

Card Functional Overview

A functional diagram for the processor card is given in Figure 1. Logic on the card adapts the ROMP and MMU interfaces for other system components and develops a special test interface [3] that supports hardware and software debug without removal of the ROMP processor chip or any perturbation of the physical card components.

The MMU connects to ROM and a Reference-and-Change Array on the processor card as well as with the interleaved memory cards. The ROM contains programs to perform power-on diagnostics and IPL functions for the system. The diagnostic display indicators are also driven by logic on the card. Failure isolation is improved by packaging the diagnostic ROM and the display drivers with the processor components.

The memory-specific controls were placed external to the MMU to improve flexibility in

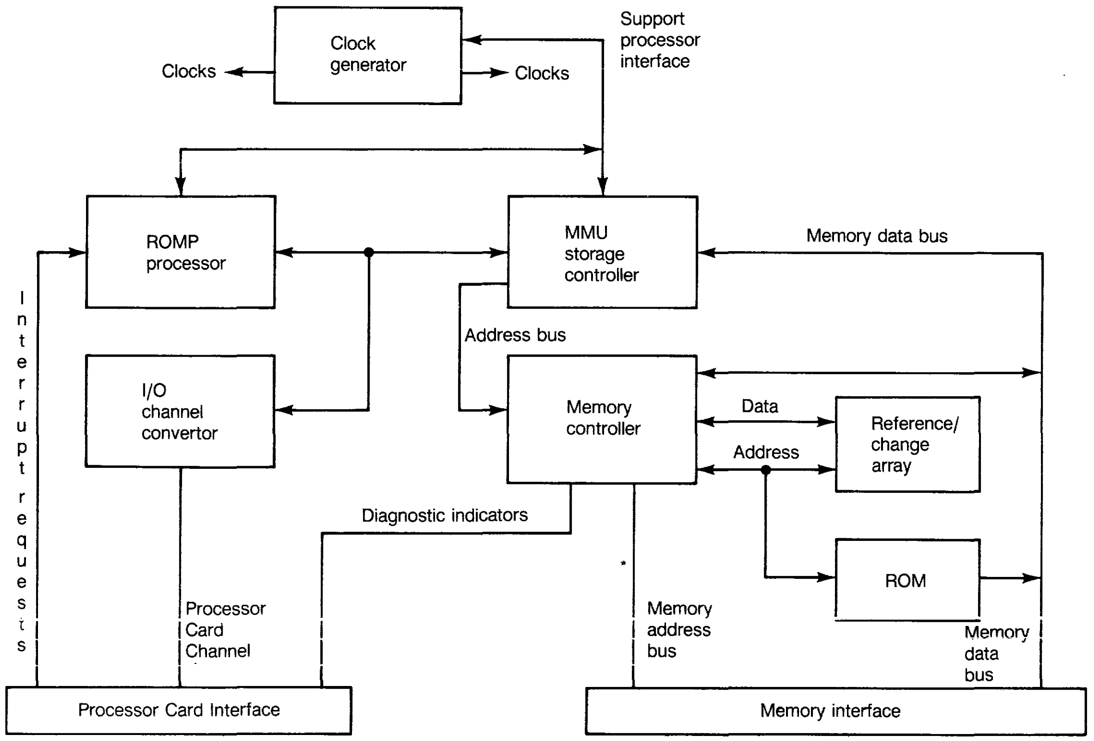


Figure 1 Processor Card Functional Diagram

selecting dynamic RAM and ROM modules for optimum cost/performance characteristics in the system. The Reference-and-Change Array is external to the MMU due to space limitations on the chip. (The function of the Reference and Change Array is explained later.)

I/O interface adapter logic was included on the processor card to improve fault isolation and to enhance performance. This approach also restricts the high frequency, repetitive clocks to the card and minimizes electromagnetic interference exposures for the system.

The clock generator logic is driven by a 23.5294 MHz oscillator to provide clocks for a 170-nanosecond processor cycle time. The timing relationship between clocks on the card is tightly controlled by synchronizing the clocks in FAST TTL modules.

Processor Card Interfaces
Two independent system interfaces, the processor channel and the memory channel, connect the processor card to the rest of an RT PC system. A test interface is also available for special hardware and software debug functions.

The memory channel can connect to one or two memory cards containing a maximum of 16 megabytes of memory. The memory cards are described in the article by Rowland [1].

The processor channel is an asynchronous, 32-bit, bidirectional channel which connects to the I/O Channel Controller (IOCC) on the RT PC system board [2] and to the Floating Point Accelerator Card [4].

Memory Interface
The memory interface logic manages an address bus, a data bus, and control signals permitting two memory accesses to be in progress simultaneously. The interface connects to both ROM and RAM and is flexible regarding the amount of each which may be present in a system. Memory refresh is managed and directed to an idle memory bank when possible, thus reducing refresh interference.

The memory interface logic contains two IBM bipolar gate arrays, TTL logic for memory card interface buffering, four ROM modules, and a 16K by 1 static RAM which is the Reference-and-Change Array.

Two memory card slots on the system board and a range of memory card configurations are supported. Memory cards ranging in size from 512K bytes to 8M bytes can be supported by decoding four control lines (two from each memory card slot in use). During a memory access, the memory card capacity indicators and several bits of the storage address bus are used to verify that the access is to a valid memory address. Although the processor card accepts a one or two memory card configuration, "slot-0" must be used first. Also, the memory capacity of the card in slot-0 must be greater than or equal to the memory capacity of the card in slot-1.

The RAM on each memory card is divided into two independent banks (an even bank and an odd bank) for interleaving purposes. The even bank operates on requests with an even fullword storage address from the MMU and the odd bank operates on the odd address requests. All accesses to storage are done using fullword data transfers. Interleaved memory improves system performance by allowing two memory accesses to be in progress at the same time. (A timing diagram which shows interleaved memory operation is included in the paper on ROMP/MMU Implementation [3].) A separate 10-bit RAS/CAS address bus is provided for each bank of storage. The memory card interface consists of the two 10-bit address buses, a 40-bit bidirectional data bus (including 8 bits ECC), and 21 control lines.

The processor card memory interface generates all clocks and controls needed to operate the memory modules on the memory cards. Four separate row address strobe clocks are created to support the maximum memory card configuration (i.e., even bank/slot-0, odd bank/slot-0, even bank/slot-1, and odd bank/slot-1). Two column address strobe clocks are provided: one for the two even banks, and one for the two odd banks. Enable signals are provided to control read/write of the memory as well as the direction of the data bus.

The MMU chip makes use of an external array consisting of one "Reference" bit and one "Change" bit for each page of real storage in the system. The Reference bit indicates if the corresponding page has been accessed, and the Change bit indicates if the page has been altered. This information is used by system software for page replacement decisions. The Reference-and-Change Array is implemented with a 16K by 1 static RAM. The storage address bits and the storage interface control signals from the MMU are used to determine if a RAM access has taken place and, if so, to set the appropriate bits in the array. The Reference-and-Change Array is also accessible to system programmers via the Programmed Input/Output (PIO) commands to the MMU.

Eight bits of data (comprising two decimal digits) for the Diagnostic Display LED indicators are driven by the memory interface logic. During IPL the LED indicators are used by software to show the hardware diagnostic program that is either currently executing or which has failed. In addition, the memory interface logic automatically sets the LED indicators to "88" if the ROMP processor stops due to a severe error or due to a halt command during debug.

Interface to the System Board
The design objectives of the processor card's interface to the system board (the "processor channel") were high performance, non-critical timings, simple system board attachment logic, and isolation of the processor card from the system board. The desirability of high performance, relaxed timing, and simple attachment is obvious. Isolation is

advantageous because it limits the most time-critical signals and clocks to the processor card, allowing better performance. It also allows future improvement of processor cycle time without impact on system design. Finally, restricting the high-speed signals to the processor card reduces the potential for electromagnetic radiation problems.

The interface between the processor card and the system board is 32-bits wide, multiplexed, and asynchronous. The address and data transferred over the processor channel are in the same format as when transferred over the RSC. The processor channel has some attributes of a general-purpose bus, but the processor card is the sole bus master, and all bus timing is under its control.

On the system board interface, there are two slaves, the I/O Channel Controller (IOCC), which develops the I/O channel, and an optional floating point card. The processor may access the IOCC or floating point card using Load or Store instructions (memory-mapped I/O) to segment 15 of the virtual or real memory space. In addition, the IOCC may access system storage through DMA.

The basic memory-mapped I/O operation involves two types of transfer: request and reply. All operations begin with a request, and replies occur if the operation is a read or a translated write. The requests and replies may be uncoupled from each other, or they may be attached. This is controlled by the slave using handshaking signals. A request consists of address and data. In response to a request from the system processor, the processor channel interface logic outputs address, then asserts DATA STROBE, changes the address to data, removes DATA STROBE, and finally disables its data bus drivers. Address and data are latched on the

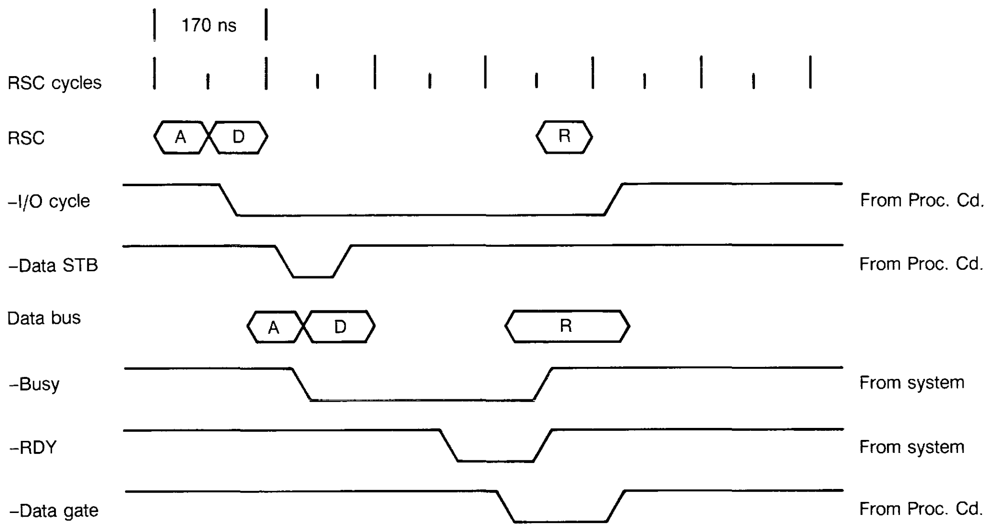


Figure 2 Programmed I/O Cycle

positive and negative transitions of DATA STROBE, respectively. If the slave requires a "single envelope" cycle, it may lock the channel by activating the BUSY line. Otherwise, new requests may appear on the interface. A reply to a request may occur either within a single envelope cycle or at some later time. When a reply is available, the READY line is asserted. The processor card responds with a gating signal which is used to enable the slave's drivers onto the interface. The basic PIO cycle is shown in Figure 2.

The DMA cycle is managed by the processor card. The system board makes a request, and when able, the processor responds with gating signals used by the system board to gate out the address and data for the transaction. The I/O interface logic generates a request on the internal storage channel and sends a reply (if required) to the system board. Overlapping DMA requests are not performed. The basic DMA cycle is shown in Figure 3.

In the RT PC system, the IOCC always causes a single envelope cycle. Accesses to the floating point card, however, may overlap each other or IOCC cycles.

For processor-originated I/O cycles, the maximum transfer rate is approximately 7.8 megabytes per second. Transfers to the system I/O channel are paced by the speed of the I/O channel. Transfers to the floating point card could approach the theoretical maximum, limited primarily by the program doing the transfer. For DMA transfers, the interface supports a transfer rate of about 4 megabytes per second.

The I/O interface is implemented primarily with three bipolar IBM gate arrays, with a total of about 1000 gates.

Test Support
The processor card provides many functions which aid in the testing of both hardware and software in the RT PC system. These functions are built into the ROMP chip set,

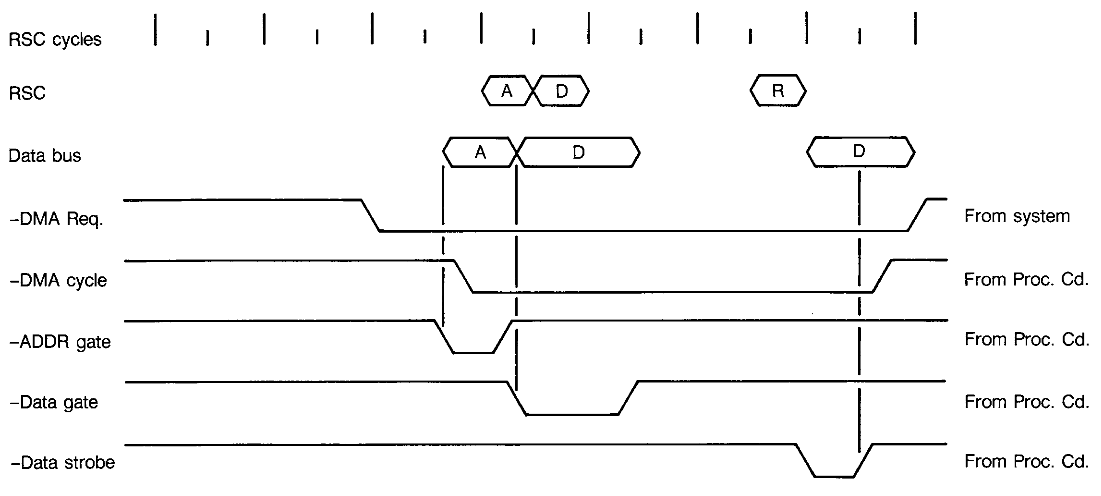


Figure 3 DMA Cycle

and are accessed by simply connecting another computer (currently an IBM PC with special interface card) to the processor card.

The ROMP, MMU, and clock chips were designed to provide easy testability with minimal external hardware. This means that unlike many development systems, no special emulator circuit is required and the physical hardware configuration is not altered for test. This is an especially important advantage for card-level test, as the ROMP processor is soldered on the card and removal is time consuming and inconsistent with high-volume manufacturing.

The support computer, called the ROMP Support Processor, processes commands typed in by the user, sends required signals to the processor card, and displays the internal state of the ROMP and MMU, as well as ROM or RAM memory.

The test features can be divided into three categories based on function provided:

- register display and alter
- clock control for breakpoints, etc.
- memory display and alter.

The first set of functions gives display and alter capability for the internal registers and control state of the ROMP. The ROMP provides access to its internal state through its Level-Sensitive Scan Design (LSSD) scan strings (i.e., all registers and latches are connected as shift registers for test purposes). By scanning data into or out of these registers, all of the ROMP's registers may be examined or altered. By scanning certain data into the shift registers and clocking the ROMP, the general-purpose registers may be accessed. Examining the contents of the ROMP's registers does not interfere with what would be the normal processor execution if the Support Processor were not attached (other than requiring the processor to be halted and then restarted), and in fact provides excellent control and visibility into the workings of the processor card and internal chip logic.

The second set of functions provides control of the system; stopping, starting, setting breakpoints, and other functions. Stopping and starting the processor card is done by causing the clock chip to stop or start the ROMP clocks. Breakpoints, which can be specified as either instruction or microcode addresses, are set up by scanning information into the ROMP which causes it to signal via a sync output just prior to executing the instruction or microcode at the indicated address. The clock chip can then stop execution by turning off the ROMP's clocks if a stop on compare has been selected. The sync signal is also available as an output from the ROMP, and is very useful as a trigger event for a logic analyzer. Microcode cycle or instruction stepping is performed by clocking the ROMP for a single machine cycle (for microcode stepping) or until its instruction complete line goes active (for instruction stepping). This provides very accurate visibility into the execution sequence down to the microcode level, if desired.

The third set of functions provides the ability to display and alter memory and internal MMU registers. The MMU provides a serial port to allow memory or register reads and writes to be performed. The serial port is completely separate from the memory interface, and therefore can be used whether or not the ROMP is functional. This provides a good starting point for system debug, since a substantial part of the processor card logic (MMU, ROM, and memory interface) can be checked without the services of the ROMP. Accesses to memory through the serial port do not affect any outstanding memory requests which the MMU may have previously buffered; therefore the system may be stopped, memory displayed, and the system restarted with no effect on normal execution.

The serial port is also used to upload or download programs between memory and PC disk/diskette.

Test features providing stopping plus register and memory display/alter capability are built into the RT PC processor card and are extremely useful for both hardware and software debug. The hardware debug aids are helpful in debugging many software problems which are time critical, branch off to an unknown point, or result in a processor stopped condition (e.g., closely-spaced, multiple program checks).

Physical Configuration
In addition to the ROMP and MMU modules, the 4.5" X 13" card contains six bipolar IBM gate arrays, TTL components (including ROM), and various passive components such as resistors and decoupling capacitors. There are two 100-pin connectors—one for the system board and the floating point accelerator card, and a second which connects to the memory cards. The special test interface is via a 60-pad arrangement on the top of the card, which is gripped by a special connector on the ROMP Support Processor cable. Figure 4 shows the processor card layout. The physical card contains four signal planes plus one ground and one voltage plane.

Conclusion
The RT PC packaging and processor card design approach are well suited to responding to future technology developments. The memory cards connect only to the processor card memory interface, permitting the relative timing between the memory and processor to be easily changed as technology improves.

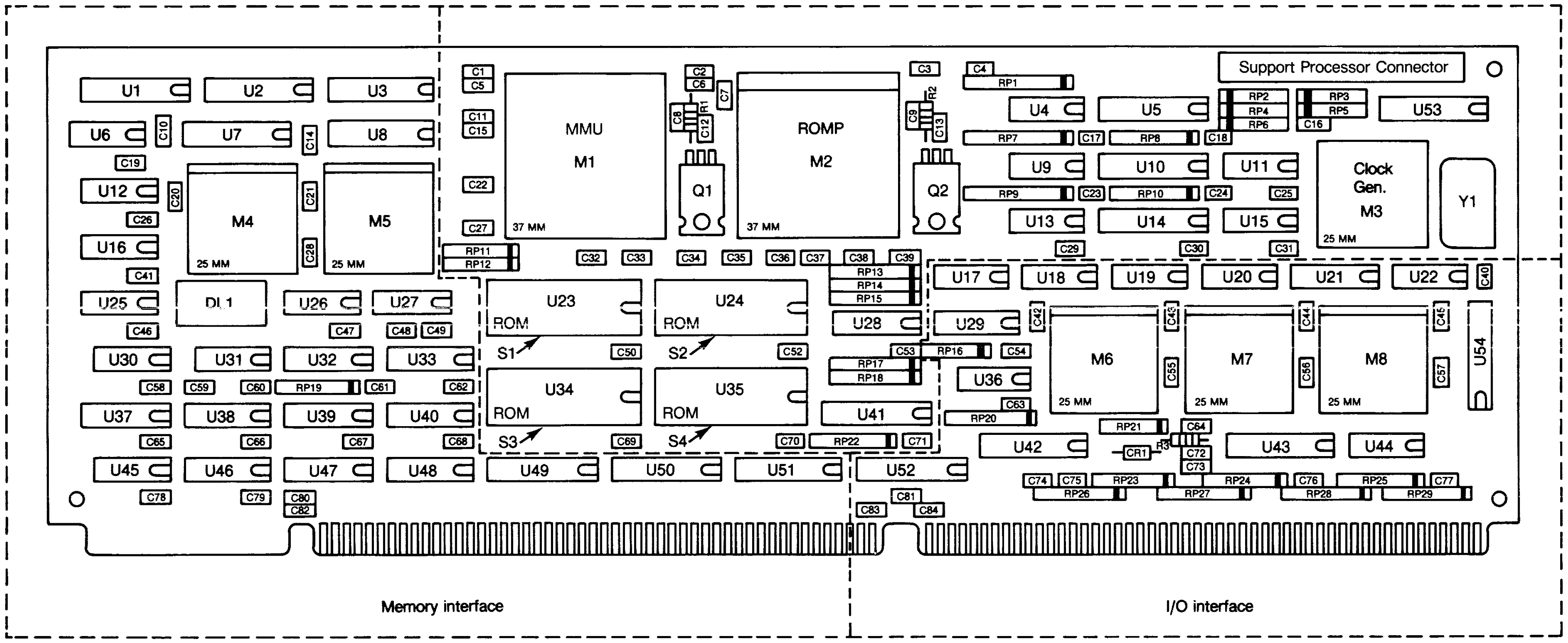


Figure 4 Processor Card Layout

The interface to the system board effectively decouples the processor card from the remainder of the system timing. Thus the processor card performance can be improved without impacting the system design.

References

1. Ronald E. Rowland "System Memory Cards," *IBM RT Personal Computer Technology,* p. 18.
2. Sheldon L. Phelps and John D. Upton, "System Board and I/O Channel for the IBM RT PC System," *IBM RT Personal Computer Technology,* p. 26.
3. D.E. Waldecker, C.G. Wright, M.S. Schmookler, T.G. Whiteside, R.D. Groves, C.P. Freeman, A. Torres, "ROMP/MMU Implementation," *IBM RT Personal Computer Technology,* p. 57.
4. Scott M. Smith, "Floating Point Accelerator," *IBM RT Personal Computer Technology,* p. 21.

System Memory Cards

Ronald E. Rowland

Introduction

Each IBM RT PC system memory card contains two independent memory arrays and associated support circuitry. The architecture of these cards provides for full two-way interleaving between the two arrays, with one array containing only even-addressed words and the other containing only odd-addressed words. This on-card interleaving scheme allows for a data word access every 170-nanosecond machine cycle while using industry-standard 150-nanosecond dynamic random access memories (DRAMs). Each data word access consists of 32 data bits and eight error correction code (ECC) bits. The availability of four data bytes every 170 nanoseconds results in a memory interface bandwidth of 23.5 megabytes per second.

The RT PC contains two dedicated slots for the system memory cards. The memory chips are packaged on 1-megabyte and 2-megabyte memory cards which provide for system memory configurations of 1M bytes, 2M bytes, 3M bytes, and 4M bytes. The 1M-byte interleaved ECC memory card design is based upon 64Kx4 DRAM technology while the 2M-byte card uses 256Kx1 DRAM technology.

The hardware architecture allows for cards containing up to 8M bytes of memory per card and for a total system memory addressing capability of 16M bytes. The use of eight ECC bits per data word supports the use of an error correction scheme capable of correcting all single-bit errors, detecting all double-bit errors and detecting the majority of multiple-bit package errors. The architecture also provides a means of automatically identifying the characteristics of the system memory configuration to the remainder of the system.

Memory Card Architecture

The RT PC system memory card architecture is shown in Figure 1. The card is divided into two independent arrays with each array having its own support circuitry to allow for full two-way interleaving between the arrays. The interleaving between the arrays is performed on a word (32 data bit) boundary. One array is used only for even-addressed word references and the other is used only for odd-addressed word references. The interleaving function is provided on a single memory card, allowing for a system memory configuration utilizing only one memory card and leaving room for memory expansion in the other dedicated memory slot.

The processor card operates on a basic machine cycle of 170 nanoseconds and the processor can cycle each of the memory arrays in two machine cycles. This translates to a memory array cycle time of 340 nanoseconds with one memory reference performed every array cycle. Each memory reference gains access to 32 bits (four bytes) of usable data. Since the two memory arrays on the memory card are independent, the processor card can operate them one machine cycle out of phase with each other in an interleaved fashion. The two-way interleaving allows for a memory reference to occur every machine cycle. The 4-byte data access in combination with the two-way interleaving results in a system memory channel throughput rate of four bytes every 170 nanoseconds (or a 23.5 megabytes per second bandwidth). This memory interface performance is approximately quadrupled over a conventional 16-bit microprocessor system operating at 12 MHz.

The figure shows that the input signals required to control these cards are very similar to a standard DRAM component. The card interface consists of 10 multiplexed address lines (ADDR 0- 9), a Row Address Strobe (RAS), a Column Address Strobe (CAS), a Write Enable (WE), and a Bus Enable (BEN) for each array on the card. The 40 data lines (DATA 00-39) are shared by the two arrays. Since the cards were architected to accept various DRAM technologies, two additional Bank Address Bit (BAB) lines and a Refresh (REF) line were added. These additional lines are shared between the two arrays on the card and are only valid when the RAS line for a given array is activated. The BAB lines contain the same information that the high order address lines contain during the column address phase of the memory cycle. This information is required at the beginning of the cycle (during the row address phase) for cards with multiple banks

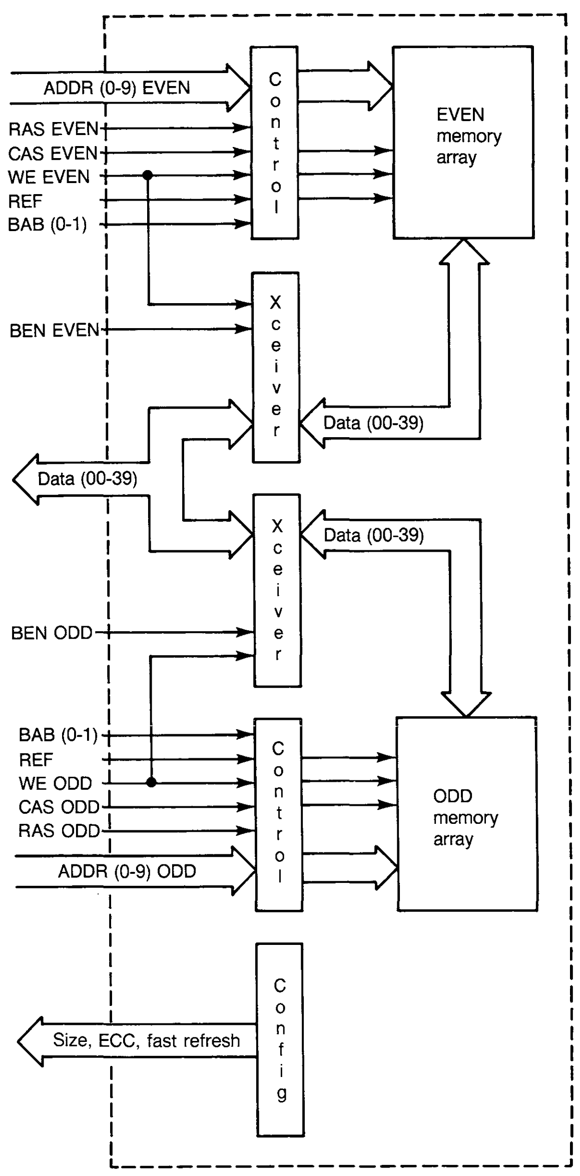


Figure 1 System Memory Card Logical Dataflow

per array so that the appropriate bank of the array can be activated. The Refresh line is required on these types of cards in order for the on-card logic to refresh the entire array (and not just one of the banks).

The RT PC system memory cards are attached to the system processor card via the memory channel. All controls for proper operation of memory read, write, and refresh cycles are provided by the processor card. Since the memory arrays share a common data bus (on the memory interface), the processor card also has the responsibility to ensure that there are no data bus usage conflicts between the two arrays.

DRAM Technology
Due to the interleaved nature of the card, each card must be able to supply the system with a minimum of 80 bits (32 data bits and 8 ECC bits from each array). When using standard 256Kx1 DRAMs in 16-pin dual inline packages (DIPs) to achieve this minimum, the result is 80 DRAM modules per card. Since the physical card size is approximately 60 square inches (one side), the minimum of 80 256Kx1 DRAM DIPs is also the maximum quantity that can be packaged on one card. Use of this standard packaging technique dictates a memory card capacity of two megabytes (2MB) with 256Kx1 DRAM DIPs. If such a card were to be implemented using standard 64Kx1 DIPs, the resulting card capacity would be 512 kilobytes.

Technical and marketing considerations deemed that a 1M-byte memory card would be necessary for the RT PC system. Since using 256Kx1 DRAM DIPs resulted in a card with a minimum of 2M bytes and using 64Kx1 DRAM DIPs resulted in a card with a maximum of 512K bytes, a new approach was needed. Various packaging methods which would allow doubling the quantity of 64K memory chips on the cards were explored. These alternative techniques included the use of surface-mounted components (SMCs), single in-line packages (SIPs), zigzag in-line packages (ZIPs), and piggy-back modules. Any of these 64K DRAM solutions to the problem would require a total of 160 DRAM chips and the use of a non-standard packaging technique to achieve the desired 1M byte capacity. The added cost and power requirements of these approaches made them unattractive for production.

The granularity question of how to combine 1M byte of memory, a 32-bit data access, and two-way interleaving on a card with 60 square inches of surface space was resolved by using a new version of the 256K DRAM chip. The answer came in the form of the 64Kx4 DRAM. The 64Kx4 DRAM is a 256K DRAM technology, but it is four bits wide instead of the usual one bit wide arrangement. With the 64Kx4 DRAM the minimum requirement of 80 bits per card could now be accomplished with only 20 DIPs and the 1MB card could be implemented with 40 modules (instead of the 160 required when using a 64K DRAM technology). The 64Kx4 DRAM also provided for a card design with varying capacities. The quantity of DRAM components could now be increased in increments of 20, yielding card capacities of 512K bytes, 1M bytes, 1.5M bytes and 2M bytes. With the merchant DRAM marketplace commanding a price premium for the 64Kx4 over the 256Kx1 DRAMs, it was decided to keep the 2M byte card design based on the 256Kx1 DRAM modules rather than designing it to be an expanded version of the 1M byte card.

Automatic Memory Identification
The RT PC system memory cards provide information to the processor card and to the system board identifying the system memory configuration. This identification indicates the capacity and functional requirements of the installed memory cards. The information is provided on five static output lines, with three of the five lines identifying a card capacity of

512K bytes, 1M bytes, 2M bytes, 4M bytes, or 8M bytes of memory. Another identification line indicates that the card has an increased refresh requirement allowing the processor card to automatically double the refresh rate. This line was added due to the uncertainty of the refresh requirements for future 1M-bit DRAM technologies.

These identification lines are basically "hardwired" on each memory card and provide the RT PC system with all the information required for proper operation of the installed system memory. This automatic identification mechanism relieves the customer of having to adjust any switch settings whenever the system memory configuration has changed.

These lines are routed to the Memory Configuration Register (MCR) on the system board and to the Memory Management Unit (MMU) on the processor card. The MCR is read by the system software to determine the memory configuration and the MMU uses this information for proper control of the memory. The information and functions provided by this mechanism include determining:

a. the amount of physical memory installed (on a per card basis),

b. the physical address range of each card,

c. whether or not the current memory configuration is a valid one,

d. whether or not the current memory reference is to a valid physical location,

e. and if the memory requires a normal or a doubled refresh rate.

Summary

The RT PC system memory cards achieve a bandwidth roughly quadruple that of a PC AT. This improvement is the result of employing a full 32-bit data interface and utilizing two-way interleaving. In addition to the 32 data bits, eight ECC bits are provided for correction of all single-bit errors and detection of most multiple-bit errors by the system MMU. The architecture allows for use of any of the common DRAM technologies now in production or envisioned over the next several years. The automatic card-identification features provide the system with the capability of determining the system memory configuration and relieve the customer of switch setting whenever the configuration is altered.

The cards are somewhat restrictive when trying to maximize their memory capacity using standard DIP type components, due to the physical size of the cards. This limitation can be overcome, however, by using other packaging techniques (SMC, SIP, ZIP, etc.) with 256K DRAMs or with the availability of 1M-bit DRAM technologies. The performance of the system memory interface could be extended beyond the current 23.5 megabytes per second limit by utilizing faster DRAMS in conjunction with a decreased basic machine cycle time. Other alternatives to extending the memory interface performance include using a different interleaving scheme, the addition of a cache, or a combination of the above approaches.

Floating Point Accelerator

Scott M. Smith

Introduction
The IBM RT PC Floating Point Accelerator (FPA) is an optional feature which provides significantly enhanced performance for floating point, math-intensive applications. It consists of one 4.5″ x 13″ circuit board which plugs into a special slot in the RT PC system board. The FPA is based upon the 10 MHz National Semiconductor NS32081 Floating Point Unit (FPU).

Since the NS32081's hardware interface is considerably different than the RT PC internal bus, logic is added to adapt the part to the RT PC. In any such non-native adapter design there is a risk of decreasing performance. Several features are included in the design to minimize and compensate for the potential performance loss. The most significant of these are discussed in this paper. They are: overlapped processing between the system 32-bit microprocessor (ROMP) [1] and the FPA, use of an external register file, and program synchronous exception handling.

Note that while the topic of discussion here is design features which improve floating point accelerator performance, it is difficult to assess the contribution of each feature to overall system performance. That level of analysis is beyond the scope of this paper.

Overlapped Processing between ROMP and FPA
Before we begin the discussion which is the subject of this section, some background material on the ROMP's use in a floating point environment is needed.

The ROMP itself has no explicit floating point commands. In the base RT PC system, a software floating point emulator provides the floating point arithmetic capabilities. If the customer requires improved floating point performance, the optional FPA may be added to his system. There is a compiler option called "compatible mode" which allows the resulting object code to run on either the emulator or the FPA. The other option, called "direct mode," produces code which will run only with the FPA installed.

The FPA is attached as a memory-mapped I/O device and its commands are encoded in the address. Specifically, the address is x'FFxxxxxx' where 'FF' is the FPA's channel address and 'xxxxxx' is the FPA command. Figure 1 shows a block diagram of the RT PC system and how the FPA is attached to it.

In order to execute an FPA command, the ROMP loads the command (x'FFxxxxxx') into one of its registers. If data is to be sent as part of the command, it is also loaded into a ROMP register. Several ROMP instructions are required to accomplish loading of these registers. The instructions require a minimum

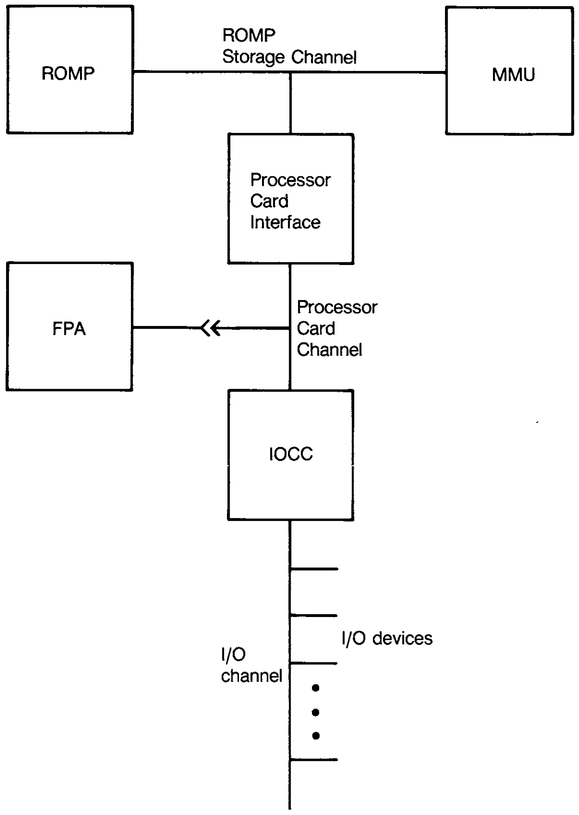


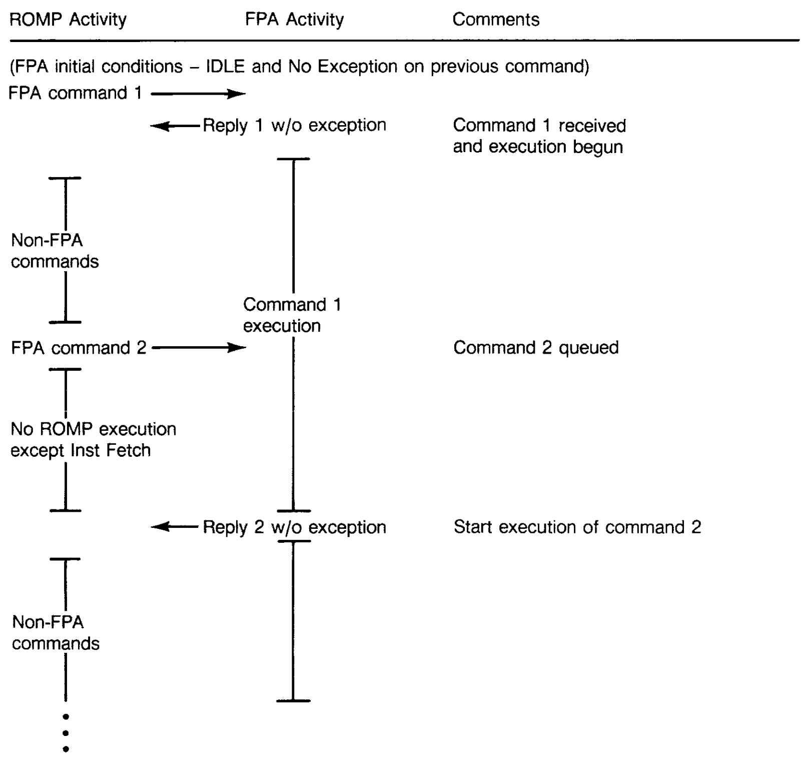
Figure 1 Attachment of Floating Point Accelerator

of 170 nanoseconds each to execute and the memory accesses that they initiate take a minimum of 850 nanoseconds. The command is then sent to the FPA using a ROMP Load or Store command. When the address and (optional) data appear on the ROMP Storage Channel, the Processor Card Interface recognizes the FPA's address and accepts the command. The command is then sent to the FPA over the Processor Card Channel. Depending upon logic delays, 115 to 188 nanoseconds elapse between the address/command first appearing on the bus and its being strobed to the FPA. The data (if any) will be strobed 242 to 315 nanoseconds from the first appearance of the address. Because the FPU clock is 10 MHz and the ROMP clock is asynchronous, an additional 0 to 100 nanoseconds delay is incurred due to synchronization.

Once the command is received, a reply is returned to the ROMP indicating successful receipt of the command. In the case of commands issued using ROMP Loads, the requested data is also returned in the reply.

If the ROMP were used to drive the NS32081's protocol directly, several such command/reply sequences would be required to execute an FPA command, e.g., Floating Add, Subtract. The ROMP cannot execute the protocol fast enough to avoid significantly degrading the NS32081's performance. Logic is therefore provided on the FPA to execute the NS32081's protocol while receiving only complete floating point commands such as Fadd, Fsub, and Fmul from the ROMP. This logic provides the base mechanism for overlapping ROMP and FPA operation.

If the reply is sent back to the ROMP early on FPA commands issued by ROMP Stores, the ROMP can continue to do other work such as setting up the next FPA command while the FPA executes the current command. In fact this concept is extended in the FPA to allow it to actually receive the next command from the ROMP while executing the current command.



ROMP Activity
FPA Activity
Comments
(FPA initial conditions – IDLE and No Exception on previous command)
FPA command 1
Reply 1 w/o exception
Command 1 received and execution begun
Non-FPA commands
Command 1 execution
FPA command 2
Command 2 queued
No ROMP execution except Inst Fetch
Reply 2 w/o exception
Start execution of command 2
Non-FPA commands

Figure 2 Overlapped Processing Between the ROMP and the FPA

Figure 2 illustrates how the overlapped processing between the ROMP and the FPA actually works. FPA command 1 is issued by the ROMP (Store assumed). The FPA receives the command, returns the reply, and begins executing the command. In the meantime, the ROMP upon receipt of the reply proceeds to execute other code, in this case preparing and issuing the next FPA command. The FPA, still executing command 1, queues command 2. At this point the ROMP can only issue instruction fetches to fill its internal buffer. Once the FPA finishes command 1, it immediately begins executing command 2. If command 2 was issued using a ROMP Store, the reply can be returned immediately, allowing the ROMP to proceed as before. If the command was issued using a ROMP Load command, the FPA must wait for execution to complete in order to obtain the data to be returned to the ROMP in the reply.

The External Register File

The following table shows several key instruction execution times for the NS32081 running with National Semiconductor's 10 MHz NS32032 microprocessor [2].

INSTRUCTION	CASE	SINGLE PRECISION	DOUBLE PRECISION
ADD	MEM-MEM	7.8 µs	8.4 µs
	REG-REG	7.4 µs	7.4 µs
MULTIPLY	MEM-MEM	5.2 µs	7.2 µs
	REG-REG	4.8 µs	6.2 µs
MOVE (NO CONVERSION)	MEM-REG	2.6 µs	2.8 µs
	REG-MEM	2.9 µs	3.1 µs

Several relationships among these execution times are of interest. An ADD or MUL Single memory-to-memory is only 0.4 µs slower than the register-to-register version of the commands. An ADD or MUL Double memory-to-memory is only 1.0 µs slower than the register-to-register versions. The time to load one single precision operand into an NS32081 register is 2.6 µs, or 2.2 µs slower than executing an ADD or MUL using the same operand memory-to-memory versus register-to-register. The time to load a register double is 1.8 µs slower than executing an ADD or MUL memory-to-memory versus register-to-register. The times to move an operand to memory are slightly slower still.

While adds and multiplies are the most often used floating point arithmetic operations, loads and stores occur even more often. In particular, at least one non-register-resident operand is needed for most floating point operations. Computations commonly occurring in engineering and scientific problems such as matrix inversion, dot product evaluation, and polynomial evaluation seem to have this characteristic. Fast load and store commands are one key to the performance of a floating point accelerator.

By providing an external register file on the RT PC FPA, the Floating Point Register (FPR) single-precision write float register command is reduced to 1.9 µs measured at the FPA interface. The single-precision read float register (into ROMP register) command is reduced to 2.2 µs. It should be noted that the corresponding times using the internal registers would increase by approximately 1.5 µs in the FPA system due to interface delays, becoming 4.1 µs and 4.4 µs respectively. The result is a savings of 2.2 µs for both FPR reads and writes.

Since the ROMP can transfer a maximum of 32 bits in a single bus cycle, two FPR read or writes must be used to handle double-precision operands. Thus, the savings is doubled or 4.4 µs per double-word move. (ROMP Load and Store Multiple commands could have been used to move double-precision operands, but doing so did not seem justified.)

Once the external register file logic is added, the cost to expand the number of registers in the file and to provide multiple register sets is minimal. The RT PC FPA has 32 sets of 16 32-bit registers. Of the 16 registers in a set, the last two are reserved for status and system use, providing the user with 14 FPRs, or six more than are provided internally by the NS32081. A command is provided which changes the register set pointer and restores the FPA status register associated with the new register set. The register sets may be allocated to different tasks in the same or different jobs. For this reason, the register set switch command (called 'Task Switch') is executable only from the ROMP supervisor state. Figure 3 shows the organization of the FPA's external register file and the associated address register.

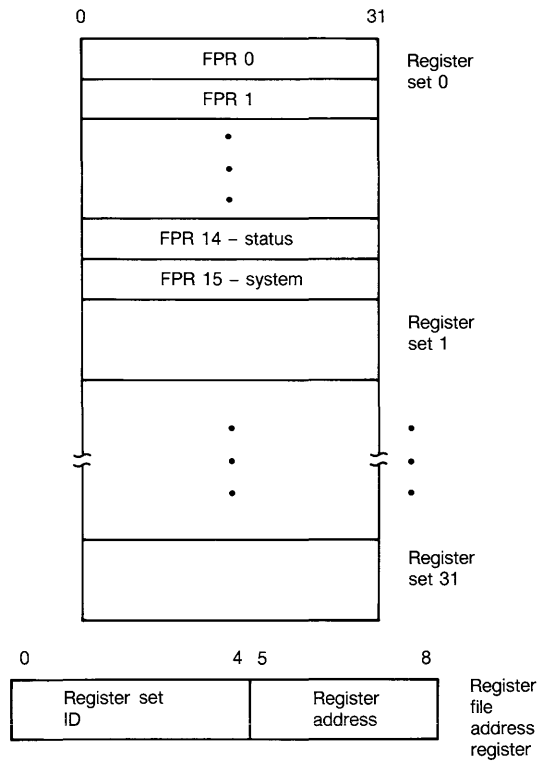


Figure 3 External Register File Organization and Addressing

A Lock command (executable only in supervisor state) is provided which places the FPA in a mode where it will execute only a few special supervisor state commands in its instruction set. This command is used when the RT PC supervisor switches to a task which does not use the FPA, to prevent inadvertent alteration of another task's FPA state.

Exception Handling
The NS32081 supports the IEEE 754 Floating Point Standard, but requires external software support for a fully conforming implementation. Thus, an efficient mechanism is needed to allow the FPA to request assistance from software as necessary and resume normal execution at the proper point. The ROMP could read the FPA's status after each command to determine if assistance is needed. While this approach would maintain synchronization between the ROMP program and the FPA, it is hardly an efficient mechanism.

The FPA uses the ROMP Program Check mechanism to signal the ROMP when it needs support to conform to the IEEE 754 Standard or to report a floating point exception as defined in the IEEE Standard, i.e., overflow, underflow, divide by zero, invalid operand, or inexact result. Thus, if the FPA detects a situation in which it needs assistance, it signals an 'Exception,' causing a ROMP Program Check. It does not alter any of the data registers and copies the command needing assistance into register 15 (numbering from 0) of the active register set. The 'FF' portion of the command is implicit and is therefore replaced by information (received with the command) on the ROMP state when the command was issued. Some of the items contained are bus read (Load) or write (Store), virtual or real addressing mode, and problem or supervisor state. The Virtual Resource Manager (VRM) reads the FPA's Status Register (FPASR) and finds that the FPA caused the Program Check. The FPASR, register 14 in a set, also contains information indicating the reason help was requested. VRM passes control and the contents of the FPASR to the supporting software. This program, called FPFPX, can read register 15 to determine what the command was. It can also read the contents of the FPRs specified by the command. FPFPX completes the operation if possible and causes execution to resume at the proper point using information saved when the Program Check occurred.

There is, however, one problem. The section on overlapped operation between the ROMP and the FPA states that the reply is sent back to the ROMP shortly after execution is begun in order to facilitate the overlap. While some types of exception conditions such as parity errors and illegal commands can be detected before the start of execution, others such as overflow, underflow, and inexact result cannot be until execution is complete. The problem is solved by reporting these exception conditions in the reply for the *following* FPA command. If one of these conditions is detected, the command which caused it is saved as are the unaltered FPRs (0-13). The FPASR is set to indicate the exception condition. When the next FPA command is received, its reply is returned immediately with the 'Exception' line asserted. The new command itself is discarded, but will be reissued by the VRM after FPFPX processing if normal processing can continue. Figure 4 shows how such an exception is reported, how the result might be corrected, and how normal execution is resumed.

As an example, suppose that the ROMP issued a "Fadd short FPR1 to FPR2 into FPR2" to the FPA (FPA command 1 in Figure 4) and that FPR1 contains a denormalized number. The NS32081 cannot process denormalized numbers so the FPA would return a reply with 'Exception' asserted when it tried to execute FPA command 2. The FPASR would next be read by VRM. When FPFPX is activated, it would read FPR15 and find a valid Fadd short command. It would then read FPR1 finding the denormalized number. The command would be emulated in accordance with the IEEE Standard to produce the correct result, which would be stored by FPFPX in FPR2. FPA command 2 would then be reissued as shown by VRM and control returned to the active task. Normal execution would resume.

Conclusions
Several methods for enhancing performance in floating point accelerators have been discussed. Overlapped processing between the ROMP and the FPA allows each to process at a faster rate and still maintain synchronization with the other. The external register file provides increased read and write FPR performance by taking advantage of the fact that in the NS32081 relatively little time penalty is incurred in using externally versus internally stored operands. Expanding the register file to multiple register sets enhances task switching performance. Use of the ROMP Program Check mechanism to notify the executing program of FPA exception conditions provides an efficient, synchronous exception handling mechanism.

The RT PC Floating Point Accelerator improves system performance of the Whetstone bench mark from approximately 12K WIPS (Whetstone Instructions Per

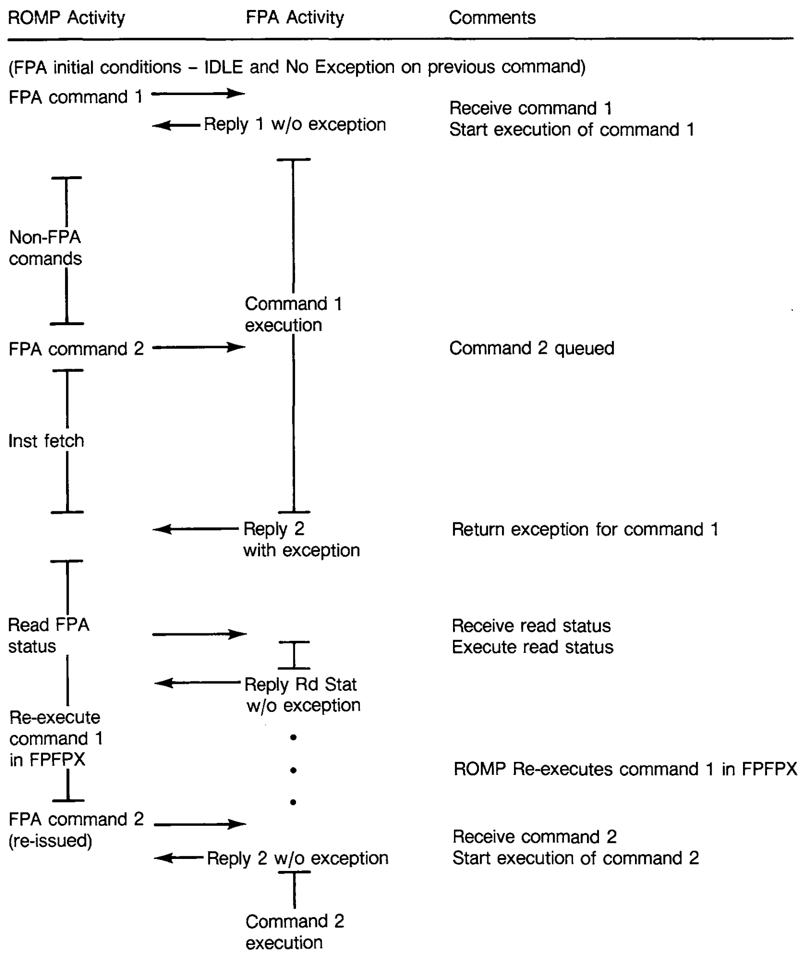


Figure 4 Exception Processing between ROMP and FPA

Second) to approximately 125K WIPS in the compatible mode and over 200K WIPS in the direct mode.

References

1. D.E. Waldecker, C.G. Wright, M.S. Schmookler, T.G. Whiteside, R.D. Groves, C.P. Freeman, A. Torres, "ROMP/MMU Implementation," *IBM RT Personal Computer Technology,* p. 57.
2. National Semiconductor Corporation, Series 32000 Instruction Set Reference Manual, pages B-11 and B-12.

System Board and I/O Channel for the IBM RT PC System

Sheldon L. Phelps and John D. Upton

Introduction
The IBM 6150 and 6151 system boards are designed to efficiently support the new IBM 32-bit microprocessor (ROMP). The system board integrates the processor card, the system memory cards, and the I/O subsystems. The 32-bit address and data structures of the ROMP are translated into the 8 and 16-bit PC and PC AT compatible I/O channel by the Input/Output Channel Controller (IOCC).

The partitioning of the design and some of the major hardware divisions were governed by the physical size of the system boards. Other decisions were based on the size of the system units and the cost of the item. While the system board design was constrained by physical size, the objective of flexibility was met in many areas by the use of approximately 20 Programmable Array Logic (PAL) devices. The PALs allowed design changes with minimal impact to the system board layout and schedules.

This article will include information about configurations, design features, data transfer methods and I/O channel performance.

System Boards
There are two basic system boards for the IBM RT PC system. The system board of the IBM 6150 (floor-standing unit) and the system board for the IBM 6151 (table-top unit) are functionally the same, except the IBM 6151 has no RS232 serial ports and two less I/O slots. The logic hardware design of the two system boards is the same. This allows the same system software to be used.

The system board contains two major groups of logic. These are the I/O Channel Converter/Controller (IOCC) and the I/O Subsystem. The IOCC contains the logic that emulates the PC I/O channel while the I/O Subsystem incorporates the adapters and system support functions normally found on the IBM PC, PC XT and PC AT system boards. Figure 1 shows a conceptual layout of the functions on the system boards.

The IOCC is responsible for translating the RT PC 32-bit processor channel to one compatible with the IBM PC product family. Much of the design of the IOCC was driven by the need to provide features which would enhance the system's use in multi-user environments. Because of the complexity of its logic and a very short design schedule, much of the IOCC is implemented in Programmable Array Logic. This design approach resulted in a compact, flexible design which was easy to modify as the system design point was refined.

The IOCC function is accomplished using three groups of logic. Together they make up about fifty percent of all logic on the system board. The logic groups are known as the I/O Channel Converter, the I/O Channel Controller and the Address Translation Controller.

The first logic partition of the IOCC is the I/O Channel Converter. It is the function of this logic to demultiplex the 32-bit processor channel and generate the signals needed to emulate the 8/16-bit PC I/O channel. An internal 16-bit data bus is generated on which resides the I/O Subsystem. The I/O Channel Converter handles data width translations to and from the 32-bit wide processor channel to the 8/16-bit wide I/O channel data bus. The second function of the Converter is to implement the protection mechanism for the I/O channel and I/O Subsystem. It does this by selectively disallowing certain accesses to the I/O channel and I/O subsystem. This gives the RT PC Virtual Resource Manager the ability to protect system resource integrity in a multi-user configuration. The final function of this logic partition is to report error conditions to the ROMP and maintain status information on the state of the system board for use in error recovery.

The second partition of the IOCC is the I/O Channel Controller. This is the logic that controls the flow of data and addresses on the I/O channel. The controller logic also sets up the proper data alignment on the I/O channel for each transfer. Finally, the Controller has responsibility for the protection of system board resources from alternate controllers operating on the I/O channel.

The last logic partition of the IOCC is the Address Translation Controller. This logic is used during Direct Memory Access (DMA) to

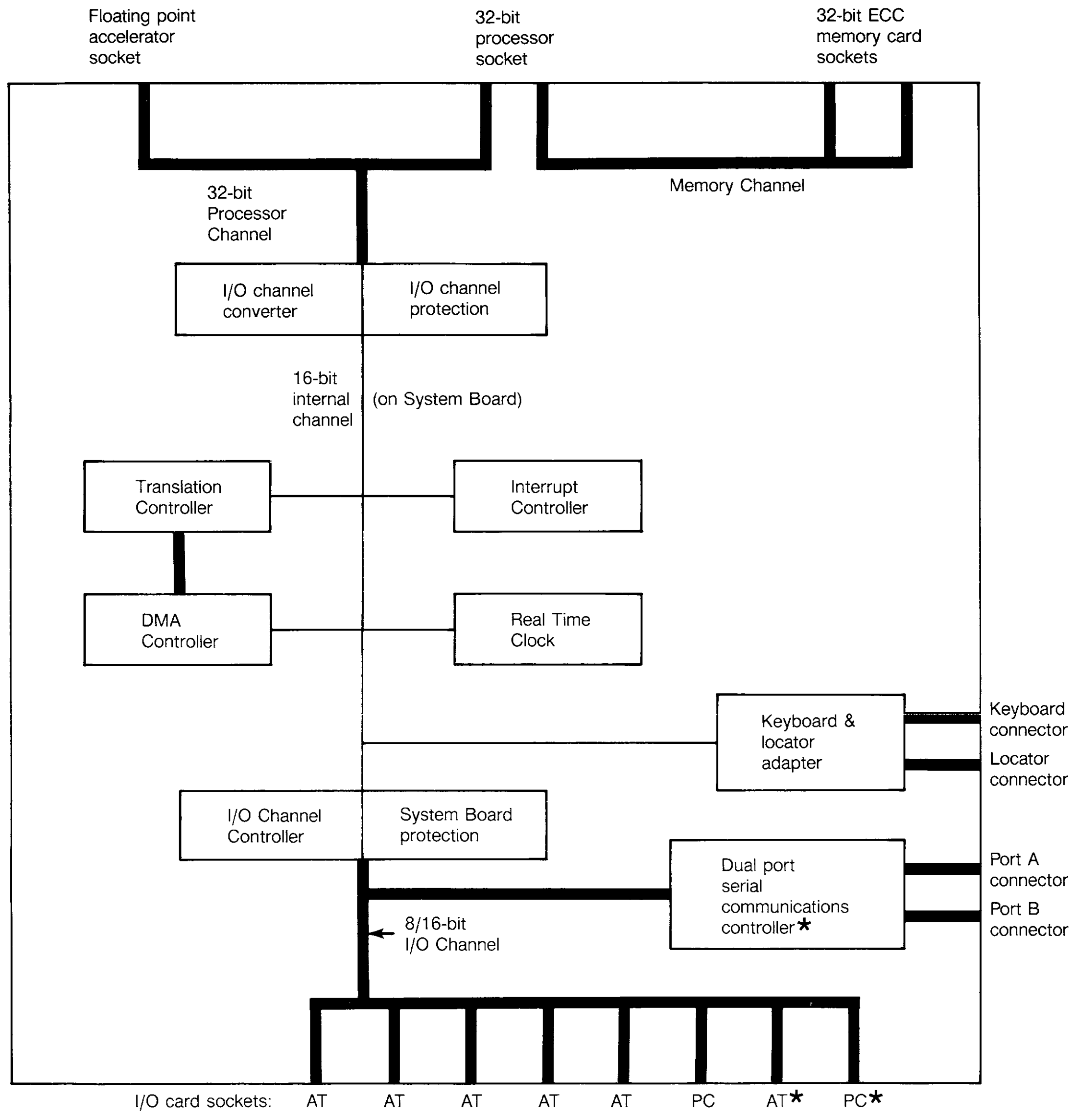


Figure 1 Block Diagram Of System Board
*Items on 6150 only.

couple the 32-bit address space of the RT PC processor channel with the 16/24-bit address space of the IBM PC and PC AT. In addition to its function of coupling the two address spaces, the Address Translation Controller is also capable of relocating addresses during the DMA transfer. This relocation is used to specify the new destination address for the data, the destination memory (I/O channel or system), and in the case of system memory, the translation mode (real or virtual).

The I/O Subsystem on the system board provides the basic system functions found in the IBM PC product family. This gives the RT PC system a high level of compatibility with preexisting PC family hardware adapters. The I/O Subsystem resides on the Internal Channel as shown in Figure 1. This channel is isolated from both the processor channel and the I/O channel.

The DMA facilities on the system board are very similar to those found on the IBM PC AT. Standard 8237A DMA Controller modules are used. The DMA controllers allow the use of either 8-bit or 16-bit devices and support 16-bit alternate controllers. The two DMA controllers are not cascaded together. All channel arbitration is implemented in separate logic to enhance performance.

In addition to the seven DMA channels of the PC AT, the RT PC system supports an eighth channel for the RT PC 80286 Coprocessor. This extra channel supports a specially modified arbitration protocol for coprocessor applications. The special arbitration allows the coprocessor to use the I/O channel while the ROMP executes out of system memory. The coprocessor will relinquish the I/O channel whenever any other device needs to use it.

Like the PC AT, I/O channel memory refresh is generated with special logic. This logic, however, differs in that refresh is executed in a burst manner. To minimize I/O channel overhead from refresh, the system will wait until at least five refresh cycles are required before requesting service. The refresh logic is capable of saving up to 16 refresh requests between uses of the I/O channel.

Included on the system board is an MC146818 Real Time Clock module. This device maintains time for the system and also contains system configuration information. Keeping system configuration data in a battery backed RAM eliminates the need for user changeable jumpers on the system board. In addition to saving system data and keeping time, the module is used to generate the system "heartbeat" timer for the ROMP.

The system board uses two 8259A Programmable Interrupt Controllers to implement the interrupt system. These controllers are not cascaded. They are connected to two separate ROMP interrupts. This allows the ROMP early knowledge as to the priority class of the interrupt. The interrupt controllers are isolated from the I/O channel by logic which allows diagnostic software to emulate interrupts during system Power On Self Tests.

An 8051 microprocessor is used for the system console input devices. The logic supports attachment of the RT PC system keyboard and the RT PC system locating device. The locating device interface uses a serial RS232C-like interface protocol. This flexibility allows the attachment of other RS232C input devices. The speaker for the RT PC system is mounted in the keyboard. Besides its normal use as an audio output device it also is used to provide audio feedback for key depressions on the soft-touch keyboard.

The IBM 6150 also has two Serial Communication Ports on the system board. These ports are implemented with a single 8530 Serial Communications Controller module. Additional logic is provided on the system board to support DMA transfer capability and to give flexible control of the RS232C interface signals.

Address Translation and Protection
Since the RT PC systems were designed with multi-user capability in mind, logic was incorporated into the system board to allow several addressing modes and provide for system resource protection by the operating system software. These features are not generally required for a single-user, stand-alone system. Address translation is used only during DMA operations. Two distinct address translation modes are supported on the system board. Each has a specific design point and use in the RT PC system. Similarly, there are two ways in which the adapters on the system board are protected from unauthorized access.

The RT PC system utilizes a 32-bit address bus on the processor channel and yet has a 16/24-bit address bus on the I/O channel. The coupling of these two address spaces is handled by the Address Translation Controller. For a Processor Input/Output (PIO) cycle, the I/O channel is seen as one of two 16M-byte segments in the 32-bit address space. Accessing either segment results in a transfer cycle on the I/O channel. The result is that the 64K-byte PC I/O address space and the 16M-byte PC AT memory address space are mapped into the 32-bit address space on the processor channel.

DMA devices present a unique problem to the system board. The DMA Controllers are capable of generating only 16 bits of address on their own. Thus the Address Translation Controller must supply the remaining 8 bits if the destination is the I/O channel and the remaining 16 bits if the destination is system memory. Furthermore, it must determine the proper destination bus for the data.

Since an alternate controller operating on the I/O channel only has access to 24 address bits, the Address Translation Controller provides a programmable means of allowing access to the system memory address space. Also, additional logic is provided to allow the alternate controller to select real or virtual addressing when the destination is system memory. If desired, the Address Translation Controller may be programmed to allow the alternate controller to run using I/O channel attached memory.

The heart of the Address Translation Controller is the array of Translation Control Words (TCWs). The TCWs are actually a 1K by 16-bit high-speed RAM. Each word of the TCW is an address translation element. Half of the TCWs are used by Page Mode to map a 2K-byte section (a page) of system memory. The other half is used by Region Mode to map 32K-byte sections of system memory. The DMA Mode Register on the system board is used to determine which type of TCW word will be used for a given DMA channel.

The half of the TCW entries dedicated to Page Mode is further divided into eight groups of 64 entries each. Each of the groups is used to provide the address mapping for a single DMA channel. In this way each DMA channel is allowed to transfer up to 128K bytes of data. The remaining half of the TCWs is shared by all DMA channels operating in Region Mode. In this mode, each TCW entry maps 32K bytes of memory. Thus region mode devices can access all of the 16M-byte address space. The Address Translation Controller can be set up so that some portions of this address space are mapped to the I/O channel-attached memory while others are mapped to system memory.

Protection of valued system resources is the key to a maintainable multi-user system. The RT PC system incorporated several protection mechanisms into the IOCC design point. Each of these protection functions is programmable, for greater flexibility.

First, the PC I/O channel may be protected from access by an application program. This protection is separate for the I/O and memory address spaces. The operating system always has access to the I/O channel, while application programs may be disallowed from accessing either the I/O map, memory map, or both, on the I/O channel. This ensures that the operating system has complete control over, and always knows the state of, each system resource. This protection helps keep a wayward application from crashing the whole system.

Secondly, the system board resources may be protected from illegal access by an alternate controller. The RT PC 80286 coprocessor is one such controller. A programmable mechanism exists on the system board that will selectively disallow access to certain system board hardware resources (DMA controllers, for example). Three groups of system board logic can be separately protected from access by an intelligent device on the I/O channel. Each group contains system resources of a different level of importance to system integrity. In this way the system can be insulated from programming errors on intelligent adapters or coprocessors.

I/O Channel Operations and Data Transfer
The I/O channel on the IBM RT PC system is designed to be compatible with the IBM Personal Computer family. Most adapters designed for the IBM PC or PC AT will operate properly on the I/O channel. However, it must be pointed out that they will require the RT PC 80286 Coprocessor card to operate them unless special programs for the ROMP are written to drive them.

The IOCC on the RT PC system board performs the necessary transformations between the processor channel (IBM protocol) and the PC AT compatible I/O channel.

Two methods of data transfer are used in the RT PC system. They are Processor Input/Output (PIO) and Direct Memory Access (DMA). When PIO is the method of data transfer, the I/O channel address is coming from the address/data bus of the processor channel. Addresses on the processor channel can go to several different places. Therefore, the IOCC will only respond to those addresses which are being sent to it. Figure 2 shows the format of the 32-bit address on the processor channel. Bits 0-7 of the processor channel address select the I/O channel as the destination of the address. Bits 8-31 are the I/O channel address. This method of addressing is used by ROMP to send control information and data to the IOCC, I/O subsystems and I/O devices.

When the addresses are gated on to the I/O channel address bus they are sent to all of the address lines SA0-SA19 and LA17-LA23. They are held valid for the complete I/O cycle. Therefore, Bus Address Latch Enable (BALE) is not driven on the I/O channel by the system board. It will remain active throughout the cycle.

DMA is the second method of transferring data on the I/O channel. DMA has two modes of operation; one method is the alternate controller mode and the other is DMA device

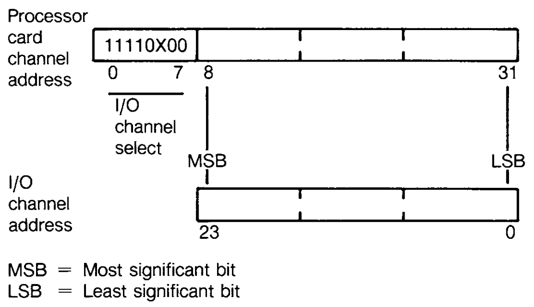

Processor card channel address
11110X00
0
7
8
31
I/O channel select
MSB
LSB
I/O channel address
23
0
MSB = Most significant bit
LSB = Least significant bit

Figure 2 Processor Card Channel to I/O Channel Addressing

mode. An alternate controller is an adapter that is capable of gaining control of the I/O channel and driving the address and control lines to perform data transfer. Alternate controllers are limited to 16-bit adapters, but can operate on any DMA channel. A DMA device is an adapter that requires the DMA controller on the system board to drive address and control lines in order to perform the data transfer. Addressing from the I/O channel to the processor channel is handled by a DMA controller and uses the Translation Control Words (TCWs).

The IOCC handles data alignment as shown in Figure 3. ROMP always uses the IBM convention for its operations. This is shown on the line titled "Processor card channel data." A "WORD" on the processor channel is 32 bits, with bit 0 being the most significant bit and bit 31 being the least significant bit. But a "WORD" on the I/O channel is 16 bits, with bit 0 being the least significant bit and bit 15 being the most significant bit. This is the convention used by the IBM PC compatible I/O channel. For example, bytes A and B are the most significant bytes on the processor channel. The bytes go through two transforms to get to the 16-bit I/O channel. The transforms are done in one step. They are

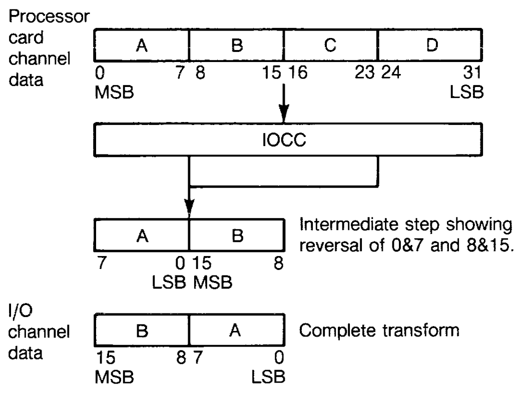


Figure 3 IOCC Data Transformations

shown in two steps for clarity. First the IOCC transfers the bits as-is, with only the numbers of the bits being changed. That is, bit 0 is renumbered to bit 7 and bit 7 to bit 0. The IOCC also reverses the bytes themselves.

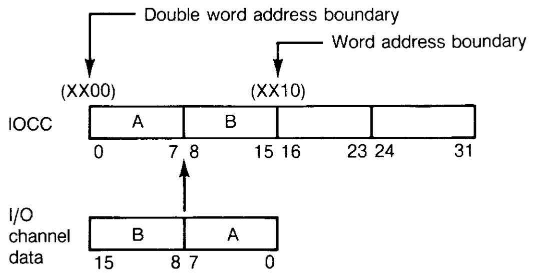
The ROMP can direct 32-bit writes to the I/O channel. The IOCC receives the 32 bits as shown in Figure 3. The IOCC will then send bytes A and B to the I/O channel for a 16-bit device, then multiplex processor channel bytes C and D to the I/O channel in the same manner as bytes A and B (as described above). This is done in separate I/O cycles. If the adapter is an 8-bit device, the IOCC will multiplex each byte A, B, C, and D separately to the I/O channel on SD0-SD7.

The RT PC system is introducing its own versions of burst mode DMA and buffer mode DMA. Burst mode DMA allows an alternate controller to take several I/O channel cycles each time it gets control of the I/O channel. Buffer mode DMA describes an alternate controller performing two 16-bit memory writes to the IOCC before the IOCC writes to system memory. System memory bandwidth is improved because only one 32-bit write is done rather than two 16-bit writes (see Figure 4). This applies to reads also. Buffer Mode DMA is limited to use by alternate controllers. Data transfers must begin on a double-word boundary. Bandwidth on the I/O channel is up to 2 megabytes per second when an adapter is designed to transfer data using both buffer and burst mode DMA.

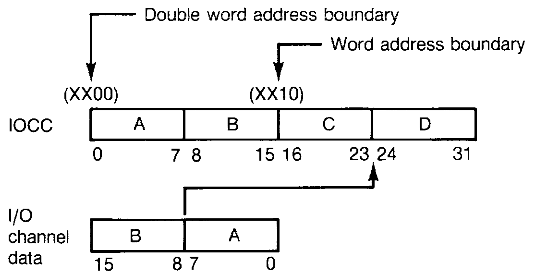
Interrupt sharing is being used to solve the problem of the limited number of interrupt levels available on the PC I/O channel. It is also used to resolve some conflicts (two adapters designed to use the same level) and to allow multiple adapters of the same type to use a single level. The shareable interrupt circuit is compatible with PC and PC AT. Adapters with shareable interrupt circuits and adapters with nonshareable interrupt circuits can be plugged in at the same time. Only one type can be enabled at a time on any one interrupt level.

Conclusion

The RT PC system boards have an IBM PC and PC AT compatible I/O channel with a maximum I/O channel bandwidth of 2 megabytes per second. A single system board design was implemented for both the IBM 6150 and the IBM 6151. Because of packaging constraints, two I/O slots and the two Serial Ports were removed from the implementation of the IBM 6150. Some of the new features of the RT PC system boards are the facilities of the IOCC. They include protection of the important system registers from intelligent devices on the I/O channel and special address alignment from the processor channel to the I/O channel. The IOCC also includes the logic to handle all of the data alignment and multiple cycle requirements for the processor channel's 32-bit data bus to the I/O channel's 8/16-bit data bus. Also included on the system boards is the logic and memory for DMA addressing through the TCWs and control of buffer and burst mode DMA operations from the I/O channel.



First step in Buffer Mode DMA transfer



Second step in Buffer Mode DMA transfer

Figure 4 Buffer Mode DMA Transfer

IBM RT PC Displays and Adapters

Joe C. St. Clair

Introduction

All-points-addressable displays allow a wide variety of applications, from windowed text processing to graphics, to be run on a system. However, this flexibility comes at a price. The large number of picture elements (pixels) that must be processed can make it difficult to update the screen rapidly. To solve this problem for a range of applications that have different cost-versus-function needs, a family of three all-points-addressable display adapters has been designed. This family of adapters solves the problem of updating the display screen rapidly with a combination of techniques. First, all three family members use innovative bit-map memory organizations that allow the pixels to be accessed easily. Second, performance assistance hardware on the cards provides help to the ROMP in manipulating the pixels. For the two smaller members of the family, this consists of special data paths on the card that, among other things, work with the system processor to move pixels in the bit map. The largest family member has a display command processor that can process lists of display commands with a minimum of help from the ROMP. The third way that the display adapter family solves the screen update problem is by using a range of technologies that includes CMOS standard cells, bipolar gate arrays, and surface-mounted components.

The display adapter family drives three monitors: a 12-inch black and white monitor with a 92-Hertz interlace refresh rate displaying 720 by 512 pixels; a 14-inch color monitor with a 0.31 millimeter shadow mask that has the same refresh rate and pixel format and displays 16 colors simultaneously out of a palette of 64 colors; and a 15-inch black and white monitor with a 60-Hertz noninterlaced refresh rate displaying 1024 by 768 pixels. The 12-inch and 14-inch monitors can display 25 lines of 80 characters using the standard 9-pixel by 20-pixel character box. With the 15-inch monitor, 38 lines of 113 characters can be displayed using the same character box. Because the display adapters that drive the monitors are all-points-addressable, character boxes of any size may be used. All three monitors provide tilt and rotate pedestals, have an antiglare treatment, and have equal spacing between horizontal and vertical pixels (square pixels).

Each of the three display adapters that drive the monitors provide the ROMP with performance assistance hardware that allows the bit map to be updated quickly. The entry level display adapter is the All-Points-Addressable-8 (APA8). It drives the 12-inch black and white monitor. The APA8C is a direct extension of the APA8 to allow it to drive a color monitor. The All-Points-Addressable-16 (APA16) display adapter drives the 15-inch black and white monitor. The three adapters provide successively greater levels of assistance to the microprocessors in updating the bit map. Each of the adapters takes up one slot in the RT PC.

APA8 and APA8 Color Adapter Functions

The APA8 display adapter provides a single plane of 64K bytes of memory. The color version, the APA8C, provides four planes of memory identical to that found on the monochrome adapter, for a total of 256K bytes of memory. Both cards have a unique byte-overlapped memory organization.

The APA8 and APA8C display adapters' unique memory organization allows the ROMP to access the bit map without encountering any word boundaries. This is done by having successive 16-bit words overlap each other by one byte. Thus, any 16-bit word can be written into the bit map with a single memory operation. This memory organization is particularly efficient when displaying text composed of character boxes that are the usual nine pixels wide. Without this unique memory, half the character boxes would bridge two words and so would require special processing by the ROMP. With this organization, all character images on the screen are treated the same.

Mode bits in the control registers on the APA8 and APA8C display adapters can be set to configure the bit map memory so that 16-bit words written into the bit map appear on the screen either as two successive horizontal bytes or with one byte atop the other. Also, an overlay mode is available so that new information written into the bit map is "OR"ed with the information already there.

This avoids many of the read-modify-write cycles usually needed when updating display bit maps.

Both adapters also help the ROMP move blocks of pixels around in the bit map and between the main store and the bit map. To help move pixels within the bit map, the cards provide a set of registers, barrel shifters, and logic units that realign pixels within bytes. Using this hardware assistance, the ROMP can move and realign 16 pixels at a time by reading from a source address and writing to a destination address. Hardware on the card takes care of realigning, masking and merging the source data before writing it to the destination. Optionally, the data can be inverted before being written back into the bit map. The color adapter operates on all bit map planes simultaneously and so can perform data moves at the same speed as the black and white adapter even though it must move four times as many bits. Both adapters have an autoincrementing pointer that can be used to address the array when moving blocks of data between the bit map and system memory.

Figure 1 shows a block diagram of the APA8 monochrome display adapter. The APA8C is identical except that it has four copies of the bit map and logic unit. Also, the video output passes through a look-up table before it goes to the monitor.

These two adapters use CMOS standard cell integrated circuits to achieve a high level of function at a low cost and with minimum power dissipation. Standard cell technology occupies a place between gate arrays and custom integrated circuits in terms of both cost and development time. Like custom ICs, it requires that a full set of masks be generated for each circuit. However, the development time and cost is considerably less because the technique puts constraints on the IC designer. A standard cell circuit is built from a library of cells all having the same height. These cells are arranged into rows with variable spacing between the rows to allow for wiring. The constraints on the circuit's physical design allow automatic placement of the cells and interconnect wiring. The automation reduces both design costs and turnaround time compared to custom ICs. However, because the cells themselves are like small custom ICs, typically implementing such functions as flip-flops and multiplexers, they are smaller and have better performance than their gate array equivalents.

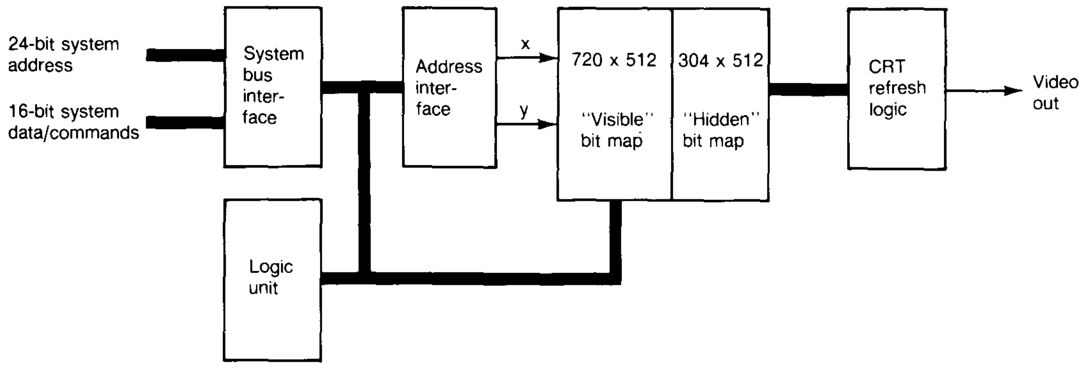


Figure 1 APA 8 Display Adapter Functional Diagram

APA8 Color

On the color display adapter, a plane protection mask register can be used to prevent any of the four bit-map planes from being updated. This adapter also provides a color expansion function that allows all unprotected planes to be written simultaneously. When using this function, a bit written by the system processor to a value of "ONE" is converted into a "FOREGROUND" pixel color before being written into the bit map planes, and a "ZERO" bit is converted into a "BACKGROUND" pixel color. The "FOREGROUND" and "BACKGROUND" pixel colors are defined by a value written into a control register. The color display adapter also has a color look-up table that maps the 16 possible pixel values into the 64 possible color values that can be displayed by the monitor.

In order to fit the extra bit map planes and function onto the color display adapter, surface mount technology is used. Small outline (SO) packages and plastic leaded chip carriers (PLCC) are used for most of the components on the card. These packages take up less card area than more conventional dual-inline-pin (DIP) packages. The PLCC package is particularly attractive as an alternate to 64-pin DIP packages and pin grid arrays. Such large DIPs make inefficient use of card space and present reliability problems because of differences in the thermal expansion coefficients between the DIP and the card on which it is mounted.

PLCC packages have considerable cost advantage over pin grid array packages.

APA 16

The APA16 display adapter drives the 1024 by 768 pixel 15-inch black and white monitor. Because of the larger number of pixels this adapter must manipulate compared to the APA8 and APA8C adapters, it provides more extensive hardware on the card to help the ROMP. This additional hardware includes a unique bit-map addressing scheme, a command processor, and a hardware cursor.

Figure 2 is a block diagram of the APA16. This display adapter provides 128K bytes of memory. The first 96K bytes of this are scanned out to the monitor to form the image. The remaining 32K bytes are used by the display controller located on the card.

The bit-map addressing scheme used on the APA16 is called a Bit Addressable, Multidimensional Array (BAMDA). The BAMDA memory organization allows up to 16 contiguous pixels to be accessed in a single memory cycle along either the horizontal or vertical screen axis and to begin at any arbitrary pixel boundary. This BAMDA allows vertical lines to be drawn at the same speed as horizontal lines. It also allows tall and thin areas, such as character boxes, to be manipulated efficiently.

The system processor can either update the bit map directly by writing to the BAMDA memory, or it can load commands into a display controller located on the adapter and let it update the bit map. Lists of display controller commands are executed out of the non-displayed portion of the bit map. These command lists may be loaded directly into the non-displayed bit map by the system processor or the display controller itself can

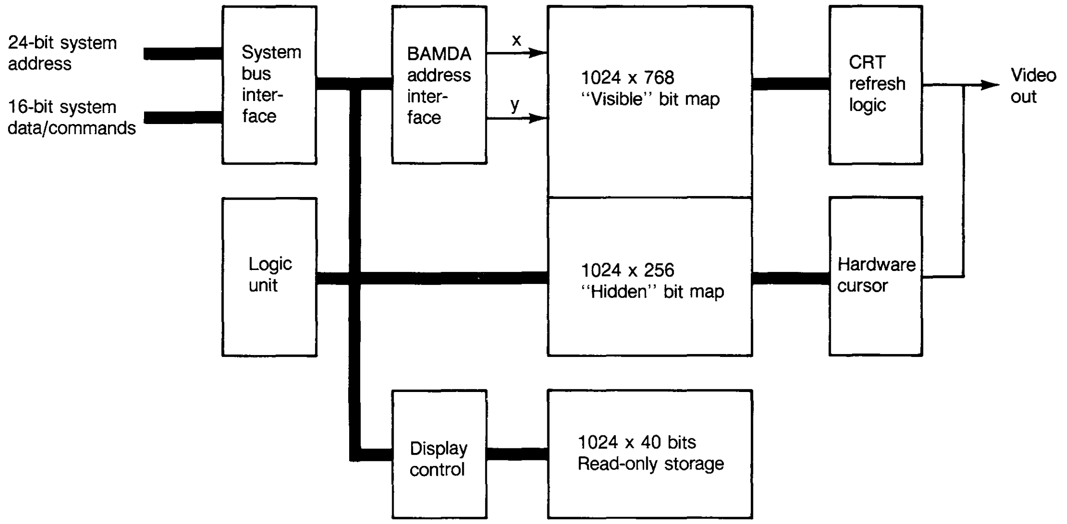


Figure 2 APA16 Display Adapter Functional Diagram

down-load the command list from main store. In this mode of operation, the non-displayed portion of the bit map acts as a cache for the display controller and the main store is used to hold the extended display command list.

The set of commands for the display controller includes operations to draw lines, manipulate blocks of pixels (BITBLT), read from main store (DMA), and control the flow of instructions. The coordinates for line drawing can be in either absolute or relative screen coordinates. The basic operations that manipulate blocks of pixels include rectangular area fill, copy, and merge source area with destination area using any of several logical operations. The copy and merge operations can simultaneously rotate the image ($-90°$, $-180°$) or mirror it about the X or Y axis. These basic operations can be chained together to perform more complex BITBLTs.

The reads from main store can also operate on two-dimensional arrays of bits. Blocks of bits read from main store can replace blocks of pixels in the bit map or they can be merged with bit map data just as they can when doing merge operations entirely within the bit map. These operations can also rotate and mirror data just as the intra-bit-map operations can.

The flow control instructions to the display controller include a branch, a subroutine call, a return from subroutine, a wait for scan line count, and several instructions that will interrupt the ROMP under various conditions. These flow control instructions allow the display controller to execute complex lists of display commands with little or no intervention from the ROMP.

The APA16 also provides a hardware cursor. This is a 384-byte section of hidden bit map that is displayed on the screen overlaying the primary bit map image. It appears as a 48 by 64 pixel block with the position of the block determined by values loaded into registers on the card.

Like the other members of the family, this adapter uses LSI technology to allow a high-function display to fit onto a single PC card. Because of the high speeds needed on large screen display adapters, bipolar gate arrays are used on this card as well as CMOS gate arrays. The display controller located on this card is implemented as a programmable sequencer using a bipolar gate array and horizontal microcode. Other bipolar arrays provide a data path for the pixels. A CMOS gate array and video DRAMs that include an internal address incrementer provide the BAMDA memory.

Conclusion
This family of display adapters uses a range of bit-map architectures, performance assistance hardware, and technologies to provide solutions to a spectrum of applications that include graphics and high-function text. The two adapters that drive the smaller monitors use byte overlapped bit map arrays and CMOS standard cell ICs to assist the ROMP. The color display adapter uses surface-mount technology and provides extra assistance to the ROMP by allowing all planes to be manipulated simultaneously. The adapter for the 15-inch monitor uses CMOS and bipolar gate arrays and video DRAMs with address incrementers to provide extensive performance assistance hardware. This includes a bit-addressable, multi-dimensional array bit map, a microprogrammed display controller, and a hardware cursor. All three adapters ease the burden of the ROMP in updating the bit maps.

Use of Artificial Intelligence to Diagnose Hardware

Nancy A. Burns and C. Edward Williams

The Design Problem

The diagnostic software package for the IBM RT PC is designed to provide both operators and service representatives alike with the ability to diagnose the hardware, isolate faulty parts, and replace them with new parts. This strategy provided many interesting challenges in the design of the RT PC diagnostic system. The program would be running on a machine suspected of having faulty components, controlling and analyzing the results of hardware tests run on itself. Memory space was limited to a minimum of 1 megabyte. Everything necessary for bringing up the machine and running the diagnostics should be diskette resident. Finally, the configuration of the machine can be complex, variable, and expandable.

Solution

The implementation of an expert system was chosen as the solution to the design problem of RT PC diagnostics. Expert systems are programs that take knowledge encoded as rules and make conclusions on the basis of the rules in much the same way as human experts. Encoding knowledge in rule form is generally easier for human experts than programming in a high level language. In addition, a rule base can be easily modified to diagnose new components or even completely different machines. This feature should allow the same inference engine to be used in future development efforts. Finally, since the limits of current diagnostic technology had been reached, expert systems and artificial intelligence offered the opportunity to extend the state of the art.

The General Purpose System for Inferencing (GPSI) was chosen as a basis for the expert system needed to meet these challenges because of its characteristics and structure [1]. GPSI is an expert system shell developed at the University of Illinois under the funding of the IBM Scientific Center in Palo Alto, CA. It was designed for diagnostic and interpretive applications. Written in Pascal and running on an IBM Personal Computer, GPSI uses a rule base which consists of a forest of trees [2]. Each tree contains a goal at its root, and evidences needed to substantiate this goal at its leaves. This structure reflects quite well the complex, interrelated nature of hardware diagnostic rules.

The original GPSI system lacked many features necessary to the diagnostics of a small machine. Most significantly, there was no method of obtaining information except through asking questions of the user. Information found in the machine itself, such as error logs and status words, is much more reliable than users in isolating hardware problems. In addition, tests can be performed on hardware, and the results of those tests provide a clear indication of the problem encountered. A method of invoking these tests and analyzing their results was implemented.

In an effort to limit user interaction, it is desirable to have the expert system learn the answers to its own questions, if possible. This required that a method be created for indicating that a series of specific test results is equivalent to the answer to a single question posed to the user.

Since real memory size is limited, and hardware configurations are variable and volatile, a method of segmenting the knowledge base is needed. When the expert system concludes that a particular component is present and needs testing, the rules necessary for the diagnostics of that component can be read in and used in isolating machine faults.

Architecture

The resulting RT PC diagnostic system is structured as illustrated in Figure 1. There are several layers of hardware and software which make up the system. At the lowest level is the RT PC hardware itself, controlled by device driver software. The hardware is hidden from the application level by the Virtual Resource Manager (see Lang, Greenberg, and Sauer) which provides a standard I/O interface to all devices and manages the memory allocation. The diagnostic application has its own operating system, the Diagnostic Control Program (DCP). The DCP is a virtual machine which provides an AIX file system and a direct interface to the hardware device drivers.

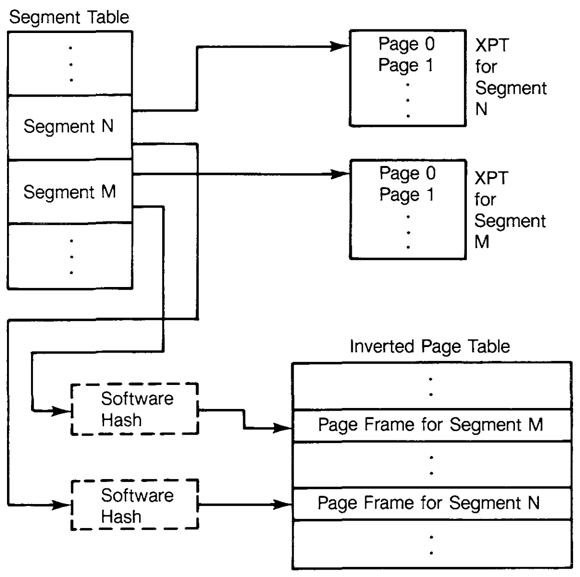


Figure 1 Virtual Memory Data Structures

A segment is divided up into 2048-byte virtual pages. A virtual page can be located in real memory or on the disk. Each segment has an external page table (XPT) with one 4-byte entry for each of its virtual pages. The XPT entries for a given segment are in contiguous virtual memory and are therefore directly addressable. An XPT entry describes the characteristics of its corresponding virtual page, such as its protection characteristics and its location on disk. The XPT is pageable.

There is a pool of external page table entries defined in the VRM [1] segment. The size of this pool limits the size of the virtual address space. The XPT for each defined segment is contained within this pool. The XPT for the VRM segment defines each page in the VRM segment, including the pool of XPT entries. The subset of the VRM segment's XPT that defines the pool of XPT entries is referred to as the XPT of the XPT. It is not pageable.

Real memory is divided up into 2048-byte page frames. A page frame can be thought of as a container for a virtual page. The Inverted Page Table (IPT) defines the virtual page that is currently associated with each page frame. The MMU uses the information in the IPT when translating a virtual address into a real address and when determining if a protection violation has occurred[2]. The MMU will respond with a page fault for any virtual memory reference that cannot be translated using the information in the IPT. The IPT contains one 32-byte entry for each page frame and is not pageable.

Support a Large Virtual Address Space
Virtual memory extends the power of computer memory by expanding the number of memory addresses that can be represented in a system while relaxing the limitation that all addressable memory must be present in real memory. The address translation hardware requires page tables fixed in real memory to perform its function. The size of a conventional page table is proportional to the size of the virtual address space, placing a practical limit on the address space size.

Paged segmentation is a means of reducing this overhead. It takes advantage of the grouping of related data in virtual memory by representing page table data separately for each segment. This allows space savings for short or unused segments of the address space.

An inverted page table further expands the range of addressability by reducing the real memory overhead required to support a very large virtual address space. Since an inverted page table contains an entry for each page of real memory, its overhead is proportional to real rather than virtual memory size. This makes it feasible to map a system's entire data base using a single set of virtual addresses (the "one-level" store). With a one-level store each segment can be large enough to represent an entire file or collection of data.

This is possible because the address translation hardware only needs the location of pages that are present in real memory. If a page is not present, the hardware must detect this fact, but it does not require the secondary storage address. The VMM does need this information, however. Hence, the VMM must keep this information in some data structure that is associated with the page. In the VRM this data structure is the external page table. Unless this external page table is pageable, the advantage of the inverted page table is lost, because the pinned real memory requirements become proportional to virtual memory size.

Large and Sparse Segment Support
The VMM supports segments of up to 256 megabytes. The VMM defines any segment that is one megabyte or larger to be a "large" segment. A large segment can be totally filled with data, assuming sufficient disk space. A large segment may also be lightly filled with data that is scattered throughout the segment. This is known as a sparse segment.

The external page table for a large segment can itself be fairly large. An XPT entry defines 2048 bytes of virtual memory. A page of XPT entries contains 512 of the 4-byte entries and defines 1 megabyte of virtual memory. Therefore, 256 pages of XPT entries are required to define a 256-megabyte segment.

Since the XPTs are pageable and reside in virtual memory, a subset of them describe the XPT area itself. These are the XPT of the

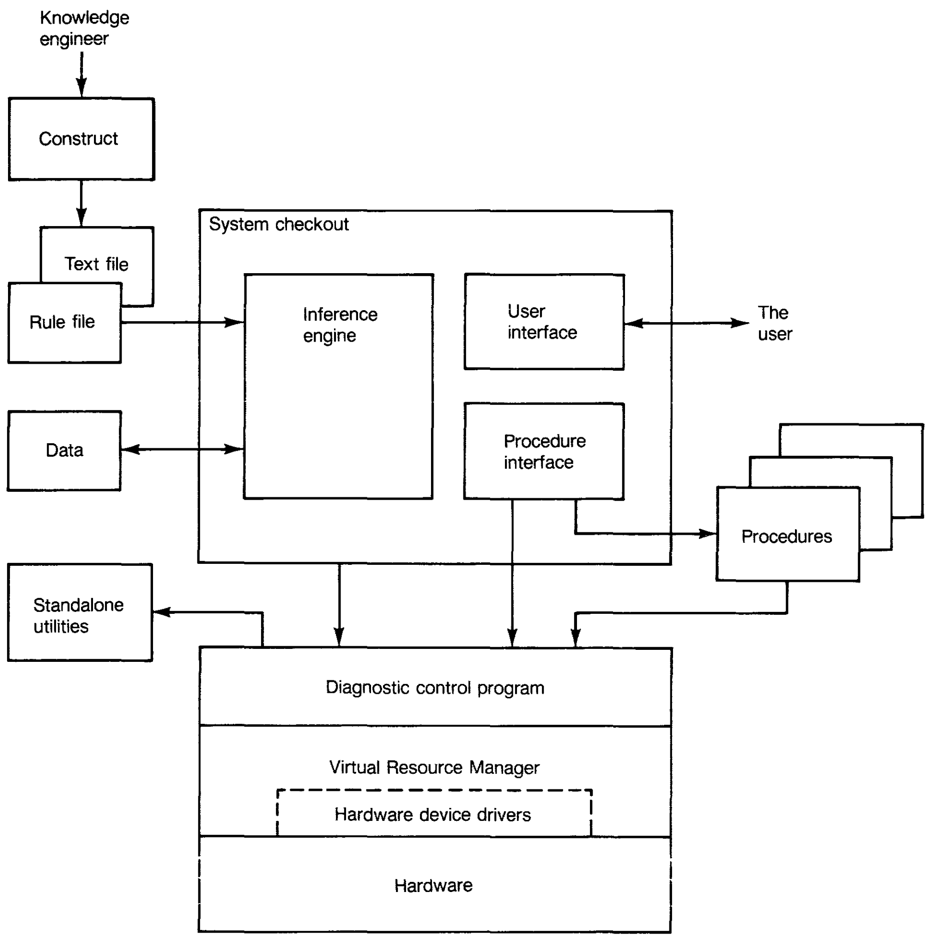


Figure 1 Diagnostic System Architecture

The expert system application is the uppermost level of software in the diagnostic system. The system checkout shell presents menus to the user and obtains information about the type of testing to be performed. It calls the inference engine as a subroutine, passing it a list of values to be used in controlling the inference process. The inference engine reads in the rule base and gathers evidence by asking questions or running procedures. It then concludes goals indicating which parts, if any, are faulty, and returns the list of concluded goals to the system checkout program.

All rules used in the expert system are precompiled by the CONSTRUCT program. The output of the compilation process is two files. One file contains the information necessary for building the data structures to be used during consultation sessions. The other file contains the text for asking questions and reporting goals. Records from this file are read in one at a time when needed.

The main CONSULT module consists of a supervisor which is invoked by the system checkout shell. It communicates with the operating system throughout consultation whenever new rule base segments need to be invoked or power turned off.

The inference algorithm, which performs such tasks as the selection of goals, the chaining through trees, and the investigation of evidence structures, is also contained in the main CONSULT module. This inference algorithm is explained in detail in the following section.

Independent modules, bound along with the inference engine, contain the User Interface Routines and the Procedure Call Coordinator. These two modules contain all machine-dependent code. The user interface routines provide the code necessary for formatting, displaying, and retrieving of information to and from the terminal screen. For the diagnostic application, a full-screen, menu-driven interface is implemented.

The Procedure Call Coordinator for the diagnostics locates the code for a particular procedure on the diagnostic diskette and loads this code into memory, dynamically binding the code to provide addressability for the expert system when so requested. In addition, it builds input and output buffers for parameters passed between the two modules and manages the invocation of the procedure on a procedure call request from the rule base being investigated.

The Inference Process

The knowledge base which drives the diagnostics is represented as a forest of one or more *n*-ary trees. Each tree contains a goal to be concluded or rejected at the root of that tree. The leaves of the tree contain evidence of several different kinds that can be acquired by querying the user, executing external procedures, or referencing other nodes, trees, or subtrees. Between the root goal and the leaf evidences are internal nodes representing a variety of functions. AND, OR, and NOT nodes can be used to relate evidences and form rules of high descriptive ability. Other types of nodes, such as the ALTERNATIVE, WAND, PAND, IF, and the PREEMPT node, introduce special structures used in controlling the flow of execution. The addition of new node types, as required by the application, is an easy enhancement because of the open architecture of the knowledge base and the modularity of the code.

The consultation session of the inference engine primarily uses backward chaining to make conclusions. Once a potential goal is selected, the underlying tree is traced in a post-order traversal which prunes off unnecessary branches and gathers any necessary evidence. The type of evidence to be gathered is indicated by the type of leaf nodes on the tree. An EVIDENCE node indicates that data should be obtained by asking the user a specified question. An EXTERNAL node indicates that data will be obtained through the execution of a specified procedure. A REFERENCE node indicates that a (sub)structure in some other rule must be investigated to obtain the required information. The structure of a sample rule tree is illustrated in Figure 2.

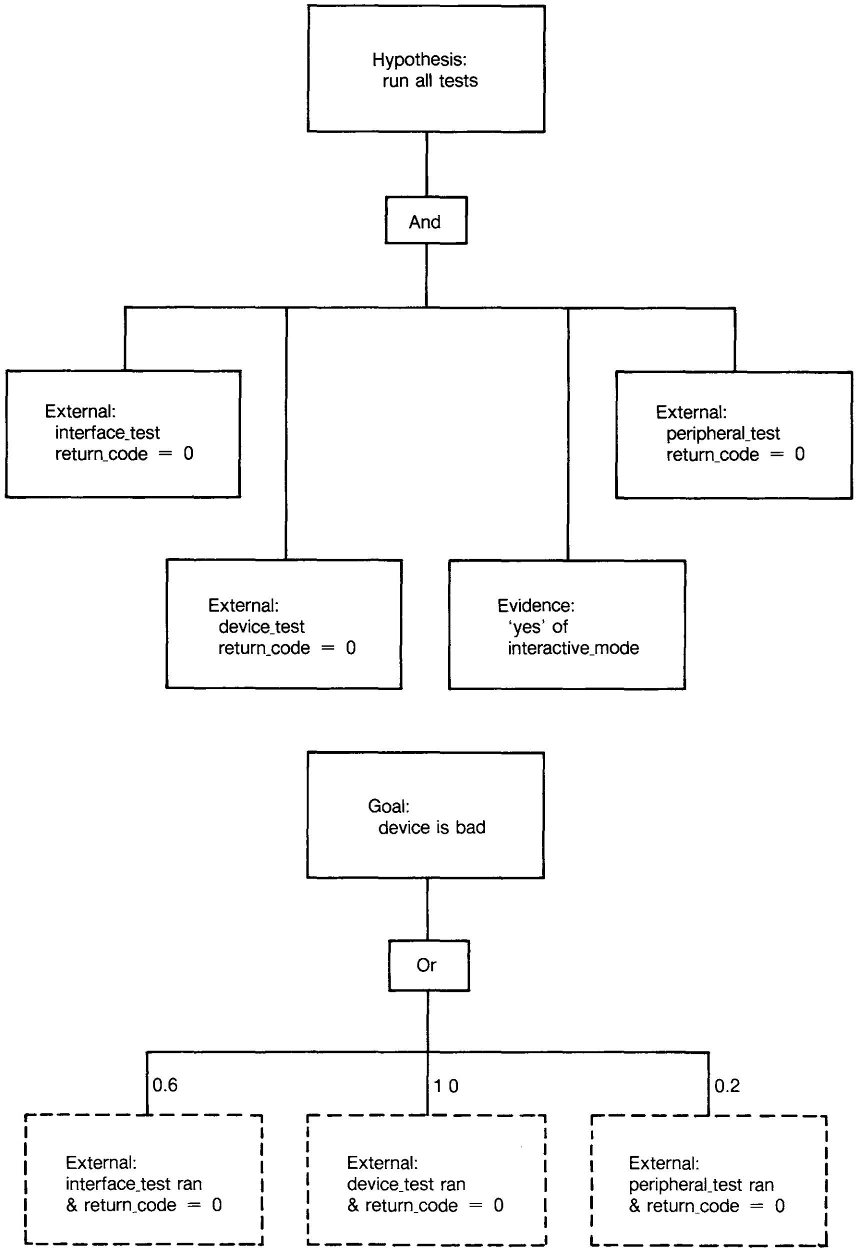

Hypothesis: run all tests
And
External: interface_test return_code = 0
External: device_test return_code = 0
Evidence: 'yes' of interactive_mode
External: peripheral_test return_code = 0
Goal: device is bad
Or
0.6
1 0
0.2
External: interface_test ran & return_code = 0
External: device_test ran & return_code = 0
External: peripheral_test ran & return_code = 0

Figure 2 Sample Rule Tree in Diagnostic Expert System

Each node of a rule tree has associated with it a confidence factor. For an EVIDENCE or an EXTERNAL node this confidence is based on an association factor given to the node by the knowledge engineer and the answer to the question asked or the value returned from the procedure. A REFERENCE node assumes the confidence value of the structure which it references. The confidence values of other nodes are calculated from the confidence of its children. The computations used for these nodes are dependent on the type of the node.

Multiple Rule Base Segmentation
In order for expert systems to simulate the reasoning processes of humans, they have historically required vast amounts of knowledge. In the past, this knowledge has been stored in a single large knowledge base. As a result, most expert systems run on large computer systems and require large amounts of memory space for holding the complete knowledge base. This organization is

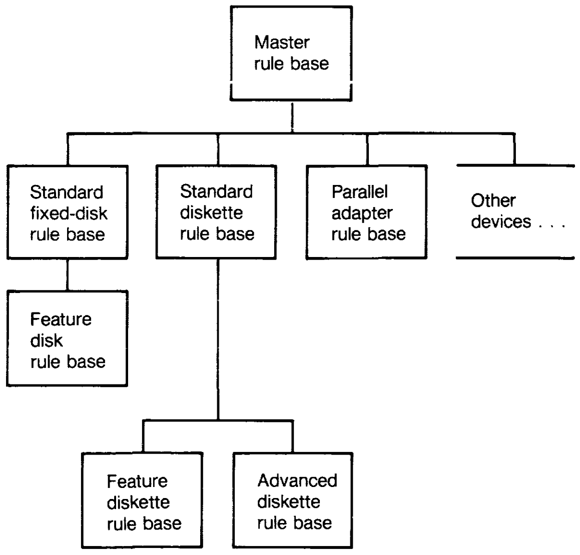

Master rule base
Standard fixed-disk rule base
Standard diskette rule base
Parallel adapter rule base
Other devices . . .
Feature disk rule base
Feature diskette rule base
Advanced diskette rule base

Figure 3 Organization of Diagnostic Rule Bases

unacceptable for an expert system residing and supervising hardware diagnostics on a small machine with a variable configuration.

During the diagnostic process, the amount of memory available for the knowledge base was restricted to approximately 170K bytes of memory. Therefore, it is desirable to store only those rules relating to components actually present and being tested in the consulting machine. At the same time, however, the knowledge base should not need to be changed for any configuration. To accomplish this goal, rules in the diagnostic application are segmented into contextual units. Each segment contains all rules necessary for the diagnosis of a single hardware component and is read into memory only when needed.

An action can be associated with any node in a rule base to indicate that the state of the current rule base should be saved, and a new rule base should be paged in. This action will only be taken if the confidence of the current node is evaluated to be greater than the high threshold associated with that node. The new rule base is then traced until all rules have been evaluated.

When all rules in the called rule base are exhausted, the original rule base is reloaded, and the tracing of it is resumed from the point at which it was suspended. Answers gathered in the called rule base and referred to in the original rule base are passed into the original rule base. Goals concluded in the called rule base are appended to the list of goals concluded in the calling rule base. Any number of rule base calls can be made from any rule base, and a called rule base in turn may call another rule base.

Any evidence which is common to more than one unit can be labelled 'GLOBAL'. This information is copied onto a global list which is passed between the calling rule base and the rule base it calls. This allows information to be passed between the individual rule bases.

Besides its usefulness for memory management, the rule base call has also proven useful for other reasons. Division of the rule base allows separating the rule base into segments of coherent knowledge structures. This makes rule writing and debugging easier and results in a more understandable rule base. Rule base calls also allow the same rule base to be used several times to conclude goals about similar, yet distinct, items. For example, only one rule base is needed for diagnosing diskette drives, although a hardware configuration may consist of multiple diskette drives.

Knowledge Base Features
The knowledge base used by the expert system for diagnosing hardware problems on an RT PC consists of multiple rule base segments. The first rule base segment is the master rule base. The master rule base executes procedures to determine the configuration of the machine and then calls device rule base segments to test the devices which are present. Each device rule base segment tests one or more of the device options. The entire diagnostic system resides on diskette and includes all rules necessary to test the work station in any mode.

The System Checkout program presents menus to the user to determine the testing environment. From the user's responses, the diagnostic shell sets three global values which are passed to the master rule base.

The first of these values indicates whether the system is being run in system checkout mode by the operator, or in advanced mode by a service representative. If a problem is not detected by the operator in system checkout mode, the system can be tested in advanced mode. Tests are longer, and there is more interaction with the user in advanced mode. In addition, advanced testing will often result in a more precise list of possible failing parts.

A second global value is used to indicate which device is selected for testing or whether the entire system should be checked out. If the user has an indication of where a problem lies, they may test only the device that is suspected.

A third global value is used to indicate that an intermittent problem is occurring, so tests may be run in loop mode until the problem is found or the user wishes to quit testing. If a machine is not tested in loop mode, then the rule base is completely traced one time.

Goal text in the diagnostic system is in the form of a service request number (SRN). This SRN is a number that the customer can report to a service organization or use themselves to replace the faulty part. When a hardware part is identified as bad by the expert system, its number is returned to the diagnostic shell which displays up to four items in descending order of their confidence values.

Rule Segment Design

Rule segments are designed by creating rules of two types: control rules and resolution rules. The control rules decide which tests are to be executed. Execution of the hardware tests are order dependent and influenced by such things as the diagnostic mode or loop mode values. The logic for control of execution of tests was simplified when the resolution of the failure was kept separate.

Resolution rules determine which parts are likely to be faulty by analyzing the results of the test units. Typically, a single goal indicates a failing Field Replaceable Unit (FRU). References to procedures in resolution rules do not cause the procedures to be executed, but allow conclusions to be made because of tests that have previously been run.

Separating the rule base into two types of rules in this manner reduced the depth of the rule trees and made the rules easier to develop, understand, and maintain.

Conclusion

Expert systems have gained wide popularity to a large extent because the reasoning mechanism of the expert system is separate from the knowledge on which the inferences are drawn. In this way, the inference logic becomes a general-purpose tool for use on other problems, requiring only a new set of codified knowledge. The use of an expert system has allowed the creation of a diagnostic system that diagnoses a complex machine with a varying and complex configuration. This system works equally well for a naive operator or an experienced service representative.

One of the problems which has kept inference tools from being truly portable, both from problem to problem and from machine to machine, has been the dependence of the expert system program on the idiosyncrasies of the underlying operating system and the requirement of vast memory resources in current expert system technology. The use of segmented rule bases allows the diagnostic system to run in limited memory and have a large and complex knowledge base. The ability to call an unrestricted, dynamic list of external procedures makes the capabilities of this system almost unlimited.

Acknowledgments

We wish to thank the entire diagnostic development team for their creativity and persistence. This paper would not have been written had it not been for the dedicated technical leadership of Richard Flagg; the efforts of Chris Iwaskiw and Roberta Starbird on the Inference Engine; Mike McBride, Luke Browning, and Scott Porter on the Diagnostic Control Program; and Doug Benignus, Chuck Cruse, and Al Laurence on the knowledge base. We are also grateful to our managers Jay Ashford and Niel Wiegand for their willingness to try something new and for their untiring support.

References

1. M.T. Harandi, "The Architecture of an Expert System Environment," *Proceedings of the Fifth International Workshop on Expert Systems,* Avignon, France, May 1985.
2. M.T. Harandi, "A Tree Based Knowledge Representation Scheme for Diagnostic Expert Systems." *Proceedings of the 1984 Conference on Intelligent Systems and Machines,* Rochester, MN, 1984.
3. F.D. Highland, "Design of an Expert System for Shuttle Ground Control," Master's Thesis, School of Sciences and Technologies, University of Houston, Clear Lake City, TX, 1985.
4. P. Nielsen, "A User's Manual for Construct and Consult in the GPSI Environment," Department of Computer Science, University of Illinois at Urbana-Champaign, 1984.

Manufacturing Innovations to Increase Quality and Reduce Cost

Charles W. Bartlett, A.V. Burghart, George M. Yanker

The IBM RT PC 6150 and 6151 are manufactured by a continuous flow process. The demand for finished units, instead of the quantities of available raw parts, paces the assembly line. Supply and demand determine the need for finished units and control the manufacturing flow according to this fluctuating need. This process minimizes inventories of raw parts and of finished products and, thus, reduces manufacturing cost.

The automation of the process further reduces cost, decreases production time, and increases product quality. Automated manufacturing methods give consistency and precision to material handling, product assembly, quality control, and data management.

Material Handling
Automated material handling minimizes the people and the time involved in storing parts, in delivering parts to the assembly line, and in moving the RT PC system units between assembly, quality control, and packaging stations. All objects that the material handling system conveys are packaged and labelled for automation.

Vendors must package and label parts according to specifications. Each carton contains only one type of part and one layer of parts. All parts are arranged for robots to pick up. A bar code label located on each carton identifies the vendor providing the part, the part number, and the engineering change level of the part. Cartons of a part are arranged on a standard 40 by 48 inch pallet. Each pallet has a bar code label.

When parts arrive by truck, each pallet filled with cartons is brought to a *pallet sizing station*. This station measures the filled pallet to verify the pallet will fit in the flow racks, scans and records the pallet's bar code label, and places the pallet on a metal skid to await storage in the flow racks.

An operator using a semiautomatic fork truck moves the filled pallet from the metal skid to the appropriate *flow rack*. All flow racks use a first-in, first-out system. When parts are required in the assembly line, an operator using a semiautomatic fork truck moves a filled pallet from the flow racks to the depalletizing station. An operator *debands* the pallet and removes the over-pack from the pallet.

The *depalletizing robot* scans the pallet bar code to determine the size and arrangement of cartons on the pallet. The robot then removes the top from each carton, places each carton in an empty plastic tote, and places each carton top on a trash belt. The empty pallets are returned to the flow racks and the plastic totes containing the cartons of parts are moved to the sizing and exception handling station.

The *sizing and exception handling station* links the bar code label on the tote with the carton of parts and verifies that the bar code label on the carton accurately describes the contents of the carton. It does this by weighing the carton and measuring the height of the carton. If either the weight or height does not match the part indicated by the carton bar code label, the tote is sent to an operator for correction. After correcting the problem, the operator can return the tote to the sizing and exception handling station.

Totes that pass the sizing tests move to the *stackers* for storage. When parts are needed in the assembly line, a conveyer system moves a tote from the stackers to the assembly line.

When a tote is emptied of all parts, the conveyer system moves the tote to the *empty tote verification station*. This station scans the tote bar code label and weighs the tote. If the tote is empty, the tote returns to the depalletizer. If the tote contains a reusable card carton, the carton is removed and directed to an in-house return conveyor and the tote returns to the depalletizer. Otherwise, the tote moves to the sizing and exception handling station for operator disposition.

Each RT PC system unit remains in a plastic tote while it is assembled, checked for quality, and transported to the packaging station. Each tote has a bar code label and is

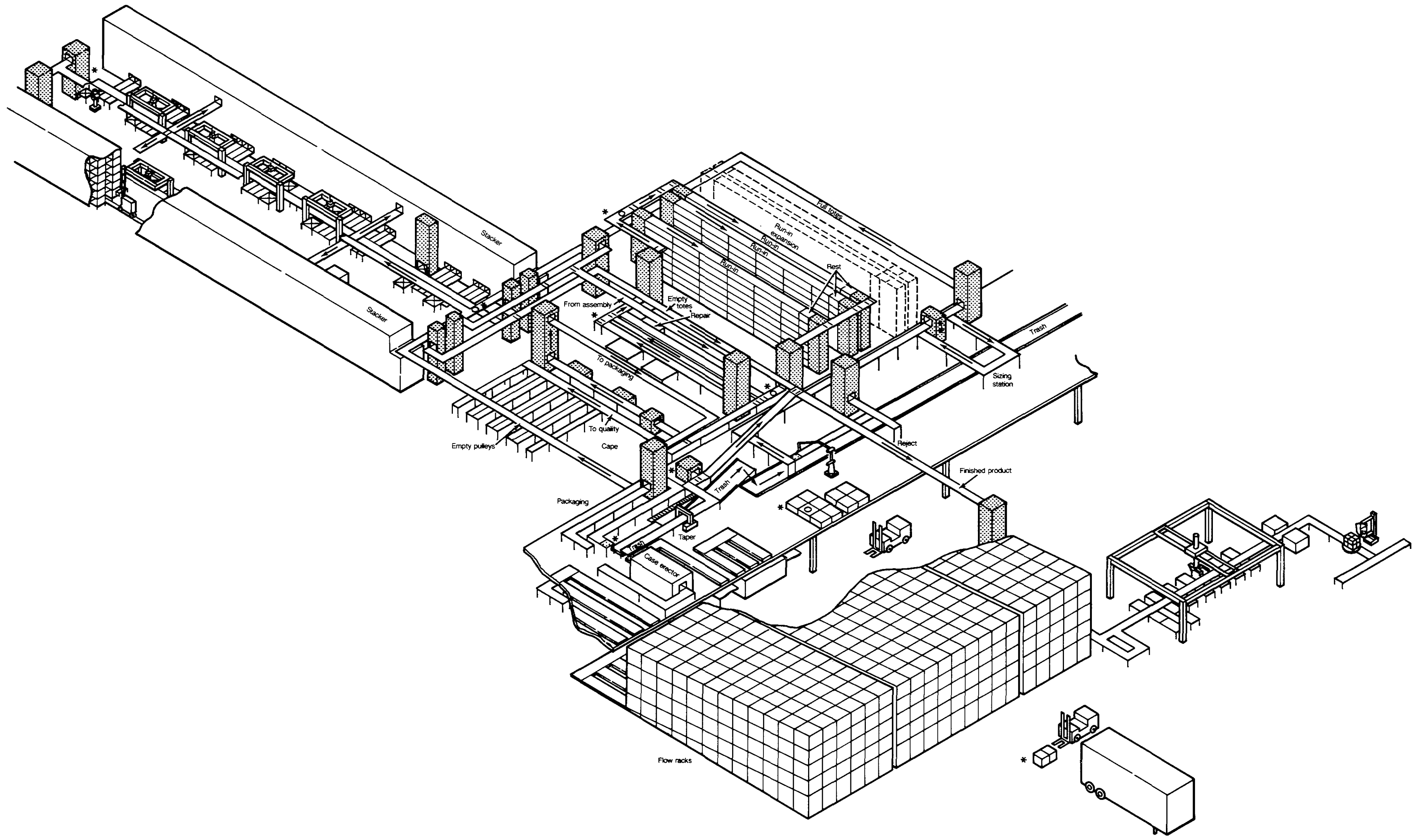


Figure 1 The Manufacturing Area

transported through the manufacturing process by a system of conveyors.

Product Assembly
Automation increases the efficiency of the assembly process by reducing assembly time and cost and increasing assembly accuracy. Because many automated assembly processes make certain demands on product design, manufacturing engineers and development engineers worked together throughout the design process.

The engineers designed the RT PC system units to be assembled in layers using a "bottom up" method, a side being the assembly "bottom". Each part fits into or onto a previously inserted part, and where needed, adequate space is available for a robotic gripper. Parts snap together and are "trapped" by other parts. (Screws are used only where needed for grounding.) Special features enable parts to funnel into place, increasing alignment precision. Parts that require an area that a robot can reliably grip are designed with such an area.

Manufacturing systems monitor each robotic manipulation. Some systems use vision techniques to determine the placement of parts. Other systems analyze strain gauges in the robotic gripper to compare the actual force required to insert a part with the expected force. Such monitoring prevents damage, increases assembly precision, and thus improves product quality.

Quality Control
All RT PC system units undergo a general quality control test, and some RT PC system units undergo special stress tests and communications interference tests. The internal self-test features of the RT PC system units, the software resident in the test controllers, and the software designed to perform unattended testing enable the quality control testing to be automated.

The general quality control test involves a power-on test, an operation test, and a verification test. A unit that fails a test moves to a repair operator. The unit is powered-on (the power-on test) and directed to perform repeated self tests (the operation test). The unit runs without a keyboard or a display and records test data on a resident diskette. A test controller checks the test data and repeats the operation test (the verification test).

Several types of stress tests further monitor quality. Short term and long term reliability tests assess the ability of the product to perform satisfactorily in the customer environment. These reliability tests are performed at an elevated temperature using typical customer software. The elevated temperature accelerates hardware failures.

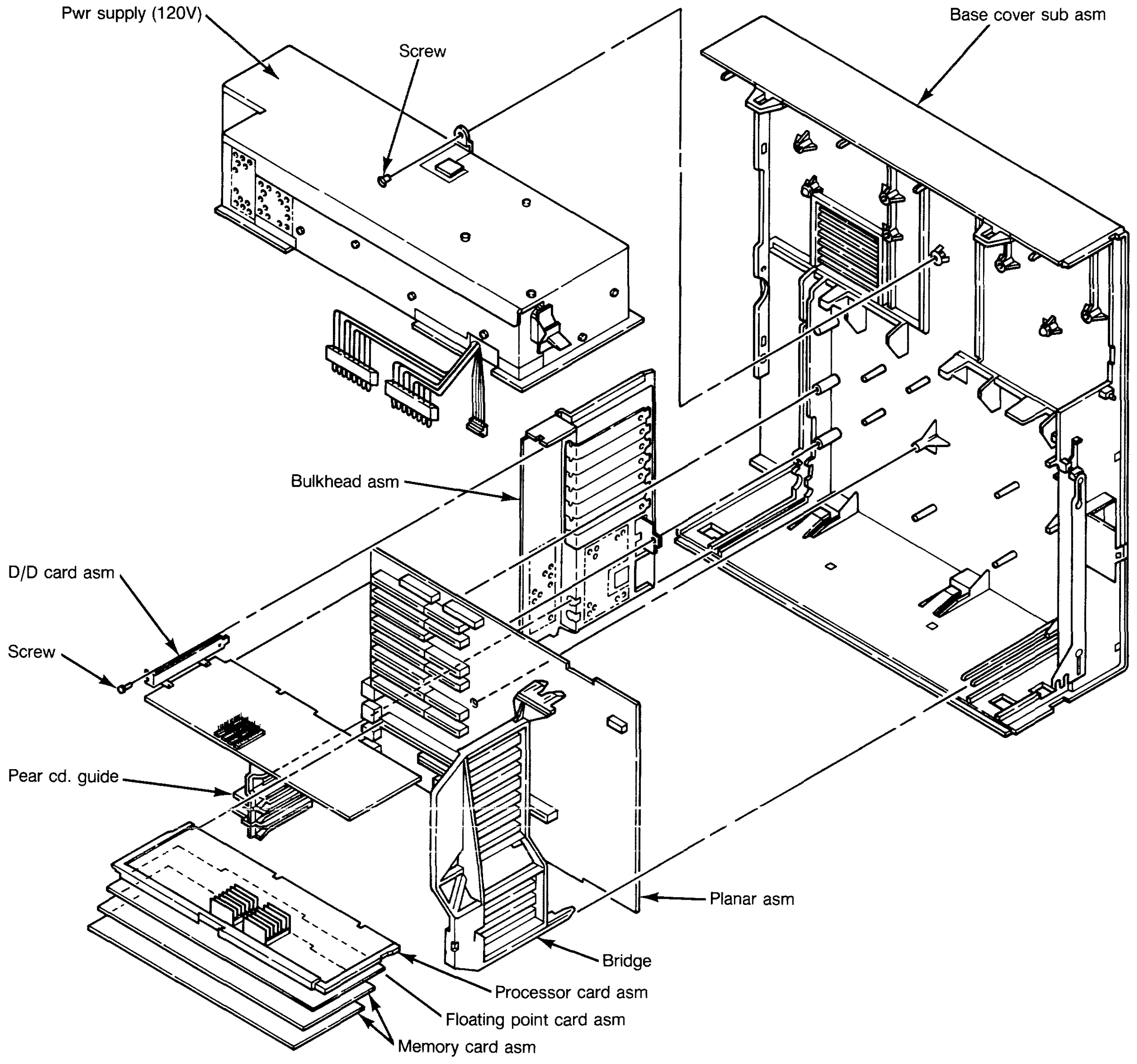


Figure 2 An IBM RT PC System Unit

Data Management
The RT PC manufacturing process uses a distributed computer control architecture to integrate computer systems and to simplify error recovery. This allows one system in the data management network to malfunction without affecting total network performance. The computer systems in the data management network perform at three functional levels. The highest level contains the main system, the middle level contains the area systems, and the lowest level contains

the station systems. Figure 3 shows the functional arrangement of the systems that control the manufacturing process.

The *manufacturing area control system*, which serves as the main system, controls the entire manufacturing process. This system communicates with the area systems. The following manufacturing areas are controlled by area systems: material handling, product assembly, quality control, label engraving, and product packaging. Each area system controls the transfer of information and tracks the status of parts, products, and station systems within its area. Each station system controls a specific manufacturing task.

The *distribution area control system* controls the distribution of materials supplied to the manufacturing process. This system manages parts inventory, tracks parts and totes, and provides data to the manufacturing area control system. The distribution area control system communicates with the following station systems: *stacker interface*, *stacker control*, *flow rack control*, *outer loop interface*, *outer loop control*, *depalletizer control*, and *exception handling control*. Thus, the distribution area control system oversees all workflow dealing with pallet storage, depalletizing, sizing, and tote storage.

The *assembly module control system* and the *assembly conveyer control system* work together to direct the assembly process. As the manufacturing area control system sends build requests to the assembly module control system, the assembly module control system checks the status of each assembly station and sends instructions to the assembly stations. The assembly conveyer control system directs the parts totes, the RT PC system unit totes, and the empty totes through the system of conveyers in the

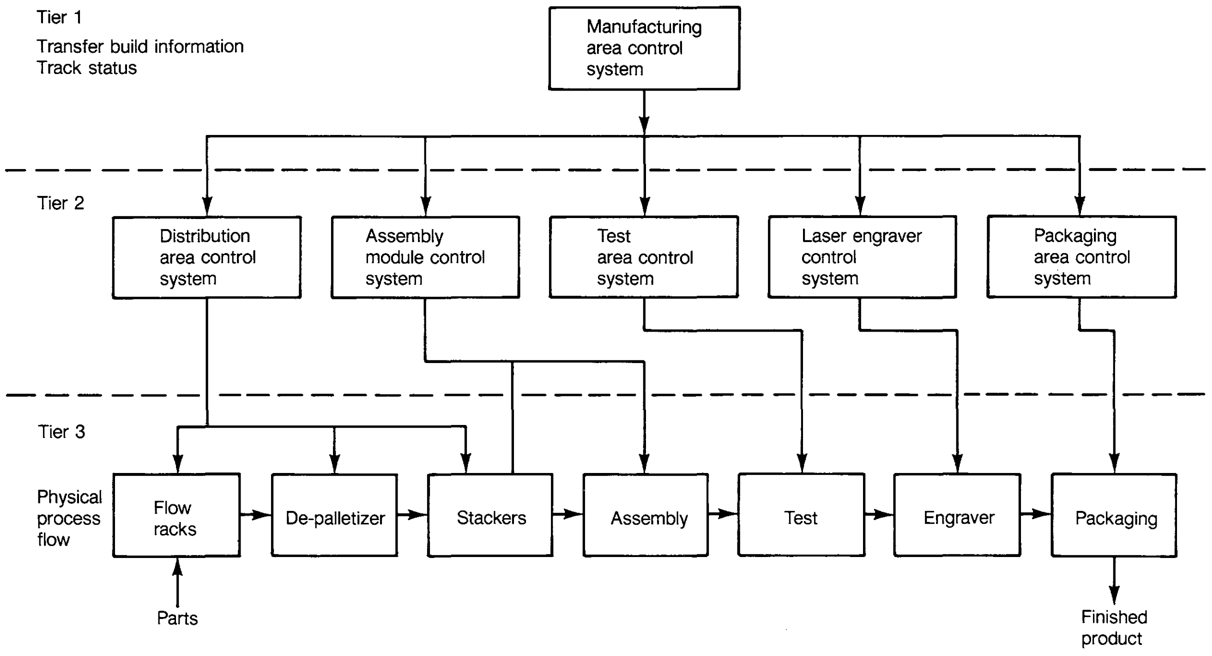

Tier 1
Transfer build information
Track status
Manufacturing area control system
Tier 2
Distribution area control system
Assembly module control system
Test area control system
Laser engraver control system
Packaging area control system
Tier 3
Physical process flow
Flow racks
De-palletizer
Stackers
Assembly
Test
Engraver
Packaging
Parts
Finished product

Figure 3 The Data Management Systems Hierarchy

assembly area. The assembly conveyer control system also sends information about each system unit being assembled to the manufacturing area control system.

As assembled units enter the quality control area, the manufacturing area control system sends product profiles to the *test area control system*. The test area control system tracks each product through the test cycles by communicating with the following test station systems: *rework control*, *run-in control*, *test conveyer control*, *screen test*, and *verify test*. Upon the successful completion of the quality tests, the test area control system conveys the results to the manufacturing area control system and routes the units to the laser serializing area.

The *laser engraver control system* directs the engraving of serial numbers, verifies the accuracy and quality of the engraving, and sends the results to the manufacturing area control system. Units leaving the engraving area are ready to be packaged and shipped.

The *packaging area control system* directs the packaging process and notifies the manufacturing area control system of boxes being released to outside distributors. The packaging area contains the following station systems: *modicon monitor station*, *packing station #1*, *packing station #2*, and *palletizing station*.

ROMP/MMU Technology Introduction

D.E. Waldecker and P.Y. Woon

The ROMP/PL.8 project was initiated by the IBM Office Products Division (OPD) in mid-1977 in Austin. OPD architects were motivated to develop a high-performance microprocessor which could be efficiently programmed using a high-level language. The "801 Project" at IBM Research in Yorktown Heights, New York had many of the same goals. (This project is described by George Radin in [1] and [2].) It was decided to take the 801 architecture and modify it as appropriate for OPD objectives. This cooperative effort became known as the Research - OPD - MicroProcessor and was given the acronym ROMP.

ROMP/801 Objectives

Objectives of both the 801 and ROMP projects were to provide high performance, Reduced Instruction Set Computer (RISC) architectures which were especially well-suited as the target for an advanced, optimizing compiler (the PL.8 compiler). The RISC architectures are characterized by use of general-purpose registers, use of only Load and Store instructions for referencing memory, and execution of most instructions in a single processor cycle. The PL.8 compiler was under development at IBM Research in Yorktown Heights in conjunction with the 801 project. The goal of the PL.8 compiler was to produce code which is almost as efficient as code developed in assembly language. Attention was given to ensure that both the 801 and ROMP machines were good compiler targets. Instruction set definition was driven by compiler requirements as opposed to performance on bench marks or optimization for a particular software kernel.

The ROMP definition was influenced by many factors. Maintaining a strong relationship with the 801 activities in Research was important in order to take advantage of compiler advances which continued throughout the development phase. Cost was a key consideration and influenced both architecture and technology selection. Storage economy was a main factor that led to differences between the ROMP and the 801 instruction sets. A technology goal was to fit the processor (ROMP) and the Memory Management Unit (MMU) on a single chip each. Another goal was to fully exercise the Burlington Silicon Gate Process (SGP) technology while maintaining chip sizes that would produce reasonable manufacturing yields.

An initial TTL model of the ROMP was operational in Austin at the end of 1978. Differences between this first ROMP and the current chip were driven by technology and, to a greater extent, by changes in the Research 801 definition. The original machines performed 24-bit arithmetic and had both 16- and 32-bit instructions. The 801 evolved to 32-bit arithmetic and addressing and the ROMP followed this lead, primarily because the need for a 32-bit address was recognized and maintaining the desired PL.8 compiler compatibility required that this change be made to both machines.

ROMP/801 Differences

Although the 801 and ROMP have a common heritage, some important differences exist between the two. The 801 assumed the use of two cache memories, one for instructions and one for data. A requirement for caches was not incorporated into the ROMP design for cost and complexity reasons. Since the ROMP can execute an instruction almost every processor cycle, an efficient memory interface capable of high bandwidth was a requirement. Two key features of the ROMP design which greatly reduce memory bandwidth limitations are: the Instruction Prefetch Buffer and the use of 16-bit, in addition to 32-bit, instructions. The ROMP contains a 16-byte instruction prefetch buffer which practically guarantees that all sequentially accessed instructions are available for execution when they are needed.

The 801 migrated to all 32-bit instructions while the ROMP maintained both 16- and 32-bit instructions. The judicious use of 16-bit instructions decreases memory code space and allows more code per real-page frame in a virtual memory system, resulting in fewer page faults and improved system performance. More importantly, the shorter average instruction length of the ROMP decreases the memory bandwidth required for instruction fetches. For example, an instruction mix containing 30% Load and

Store instructions (which require 32 bits of memory reference each for data) would require 41.6 bits of memory bandwidth per instruction if all instructions are 32 bits long. The same instruction mix executed in the ROMP, where the average instruction length (weighted average of 16- and 32-bit instructions) is about 20 bits (2.5 bytes), only requires an average of 29.6 bits for each instruction. This is a reduction in memory bandwidth requirement of almost 30% per instruction for the ROMP over a design which contains only 32-bit instructions. Since memory bandwidth is usually the performance-limiting factor, a 30% reduction in the bandwidth requirement will certainly improve performance in a non-cache system.

It must be recognized that a machine with all 32-bit instructions should do more "work" for each instruction executed than a machine with some instructions that can only be executed in a 16-bit format. That is, an equivalent MIP (Million Instructions Processed per second) rate for a machine with only 32-bit instructions should represent more processing capability than the same MIP rate for a machine with both 16- and 32-bit instructions. One of the limitations of 16-bit instructions is the limited number of bits available to specify operation codes, registers, displacements, etc. This limitation is one of the reasons that the 801 uses 32-bit instructions exclusively. Use of only 32-bit instructions permits the register specification fields to contain the 5 bits required to select one of 32 general-purpose registers (GPRs). The limit of 16 registers for the ROMP results in only a modest increase in Load and Store frequency, since the PL.8 compiler performs an efficient register optimization. A primary motivation for having 32 registers is efficient emulation of other architectures which have 16 general-purpose registers (i.e., System/370). The ROMP does an excellent job of emulating other machines which have a more limited register set. The 801 is significantly better at 370 emulation. Aside from emulation, the use of all 32-bit instructions is estimated to make the 801 MIP rate about 15% to 20% more powerful than the ROMP MIP rate. That is, software path lengths for 801 programs are about 15% to 20% shorter than they are for equivalent ROMP programs.

The use of both 16- and 32-bit instructions adds some design complexity. Instruction handling and decoding must account for instruction location on both 16- and 32-bit boundaries. The 16-byte Instruction Buffer and its management also adds complexity. However, studies have shown that the 16-byte Instruction Buffer provides about the same performance advantage as a 256-byte instruction cache, with a significant savings in the silicon required for implementation.

The design point chosen for the ROMP is well suited for a microprocessor VLSI design. Good performance is achieved with readily available memories and the silicon area requirements are a good fit for our SGP technology. The ROMP's dual 16- and 32-bit instruction format provides about a 10% net performance advantage over an equivalent 801 microprocessor in non-cache systems.

Compiler Development for ROMP & 801

The PL.8 compiler was initially developed for the 801 project in Research as part of the exploration of the interaction of computer architecture, system design, programming language, and compiler techniques. The adaptation of this compiler to the ROMP architecture was done in Austin. A single compiler was maintained with the addition of another "backend" for the ROMP. This involved a complex working relationship between Research and Austin. This excellent relationship has continued over the years with enhancements and modifications being made by both groups. The compiler is currently owned by Austin with enhancements being made by both groups.

The PL.8 compiler currently supports three source languages, Pascal, C, and PL.8, a PL/I variant designed to be suitable for generation of efficient object code for systems programming. Object code is produced for the 801, ROMP, System/370, and MC68000.

The ROMP PL.8 compiler development influenced the design of the ROMP instructions in a number of significant ways. The goal of program storage (byte) efficiency caused the following modifications to be made:

1. Short (16 bits) forms of several instructions were introduced to provide for the special case of an immediate operand with value less than 16. For example, Add Immediate, Subtract Immediate, Compare Immediate, and Load Immediate were provided.

2. A short-form relative jump instruction was added with maximum displacement of plus or minus 256 bytes.

3. The long (32-bit) Branch instructions were defined to be relative rather than absolute in order to reduce the storage necessary for relocation information from modules.

4. A Load Character instruction was added in order to handle character data with fewer bytes.

In addition, Load Multiple and Store Multiple instructions were provided to improve the speed of subroutine linkage.

The resultant ROMP architecture proved to require about 30% fewer bytes than 801 for a selected set of bench marks.

In addition, the ROMP instruction set design includes only instructions which can be used effectively by the compiler. The ROMP does not contain complex instructions and addressing modes which a compiler finds difficult to generate. The ROMP does not have complex loop closing instructions which require several free registers in order to operate. It does not contain instructions like repeat, rotate, and edit—instructions which are not primitives for PL.8 constructs. Register allocation is simplified by the requirement that variables be loaded into registers before being operated upon.

The PL.8 compiler employs state-of-the-art compiler technology [3] utilizing several independent advances in the theory of compiler design. John Cocke and Fran Allen [4] published a procedure of data flow analysis—a technique for analyzing the interval of execution over which a variable is used, and using that information for optimization and assignment of variables to registers. The technique allows efficient use of registers and enhances the reliability of generated code.

The compiler's scheduling algorithms make use of the data flow analysis results to produce a program which takes advantage of the pipelined implementation of the ROMP. Since only Load and Store operations reference memory, the compiler can very effectively intersperse memory references and register-to-register (RR) operations in the instruction stream so that processing of the RR operation can overlap the memory reference. The compiler also makes effective use of the Branch with Execute instruction. This instruction allows execution of an instruction following the Branch while the branch target instruction is being fetched. This overlap of instruction execution with the fetching of the new instruction stream results in better CPU utilization.

In addition the PL.8 compiler uses LALR parser generator [5] techniques. Syntax-directed translation enables the compiler to associate the intermediate code generation directly with the syntactic structure of the source language. Furthermore, it uses a map-coloring algorithm from topology for register allocation [6]. Most programs of reasonable size color in 16 GPR without spilling. 32 GPR would reduce spilling on larger programs but would require 5 bits for register specification which would require 32-bit instructions. The trade-off was made in favor of the use of 16-bit instructions (with the 25% to 30% performance advantage) at the performance detriment of large programs.

The compiler incorporates the primary theoretical advances in compiler design achieved over the past decade. The proof of the theory lies in its effectiveness. The approach of developing the language and the instruction set as a joint effort has paid off in language efficiency and in ease of code generation. Benchmarks have shown that the compiler generates code that approaches the performance and storage requirements of assembly code produced by a good hand coder. These results are a testimony to the success of the design approach and the compiler technology used.

Silicon Technology

As stated earlier, the initial ROMP TTL Prototype was operational in Austin at the end of '78. The success of this Prototype in demonstrating the 801 concepts applied to the ROMP, motivated us to proceed with a ROMP VLSI design. In early '79, the IBM General Technology Division in Burlington, Vermont was interested in applying their SGP (Silicon Gate Process) technology to a logic part (as opposed to a memory part). One of their objectives for such a project was that the logic part selected should be complex enough to stress the technology ground rules. The ROMP appeared to fit the requirements for a "technology-proving" development. It contains a custom register file, ROM, custom logic in data registers, multiplexers, and the ALU, Off-Chip Drivers and Receivers, plus random logic designed with a master image approach.

The division of design tasks between Austin and Burlington was a rather complex arrangement. Austin was responsible for the Functional Specification and logic design. Burlington was responsible for the final chip layout but many macros and large portions of the chip were designed in Austin. Austin performed the logic simulation and also built a nodal model of the ROMP chip to verify functionality. Burlington designed the memory for this model and also wrote many of the Architectural Verifications Programs (AVPs) used to test the model, drive the logic simulation, and ultimately test the chip functionally. Manufacturing test patterns were generated in Austin but special test patterns to resolve unique problems early in the program were generated in Burlington.

Early ROMP parts did indeed stress the technology. We were required to change the design several times as technology ground rules evolved. Changes were made to improve yields and chip reliability well after we had achieved functional parts. As we progressed, the chip was also made smaller. The initial pass was 8.35 mm square and the final version is 7.65 mm square.

Projected performance of the ROMP chip was significantly improved over time. Initial projections were for a cycle in the 250 - 300-nanosecond range. As we gained more data and modified the design to eliminate critical paths, the projections were reduced to the 200 - 250-nanosecond range. We also projected that 50% of the functional parts could be selected to execute at a 170-nanosecond cycle. The design which is in manufacture has virtually no fall-out of functional parts due to selection for 170-nanosecond operation. The typical ROMP will run at about a 135-nanosecond cycle. System considerations of memory and I/O interfacing, system clock skews, voltage variations, and tester tolerances limit our CPU cycle from being faster than 170 nanoseconds.

MMU Memory Management

The MMU is a 9 mm square SGP chip which performs the RT PC system memory management function.

The MMU chip used the same technology and design approach as ROMP. Since the ROMP had served as the vehicle to solidify the technology and design methods, the MMU design was more straightforward in many respects. However, the MMU functional requirements resulted in a larger chip than the ROMP. The MMU definition was initiated in late '81. The basis for the functional definition was System/38 and work done at Research on memory management approaches consistent with the 801 architecture. Some of the more prominent features are use of inverted page tables to minimize memory page table space, special segments to provide protection with 128-byte resolution, and ability to accomodate variable speed memories.

Favorable experience with ROMP logic simulation convinced us that there was no need to build a TTL nodal model of the MMU chip. However, in order to support early RT PC prototypes, a three-card TTL equivalent of the MMU function was developed. The early prototypes were completed in early '83. When MMU chips were received in late '83, the three-card TTL version was replaced by less than one-half card containing the VLSI MMU.

Summary

The ROMP project is an excellent example of several IBM divisions at different sites working together to produce a successful program. The project ultimately resulted in the RT PC product design by the Engineering Systems Products group in Austin. It embodied the 801 RISC / PL.8 compiler concepts developed in IBM Research and served as an important vehicle to mature the Silicon Gate Process (SGP) technology of the IBM General Technology Division.

The close relationship of the PL.8 compiler development and the hardware design is a rare occurrence and, we believe, was one of the key elements in achieving an excellent and balanced design.

References

1. George Radin, "The 801 Minicomputer," in ACM, 0-89791-066-4 82/03/0039
2. George Radin, "The 801 Minicomputer," *IBM Journal of Research & Development,* 27, pp. 237-246, May 1983.
3. Marc Auslander and Martin Hopkins, "An Overview of the PL.8 Compiler," in ACM, 0-89791-074-5/82/006/0022.
4. F.E. Allen and J.A. Cocke, "A Program Data Flow Analysis Procedure," in *CACM,* 19, 3 (March 1976).
5. W.R. La Londe, "An Efficient LALR Parser Generator," University of Toronto, Technical Report CSRG-2 (April 1971).
6. Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein, "Register Allocation via Coloring," *Computer Languages,* 6, No. 1, pp. 47-57, 1981.

The IBM RT PC ROMP and Memory Management Unit Architecture

P.D. Hester, Richard O. Simpson, Albert Chang

Introduction
This paper describes the ROMP microprocessor and companion Memory Management Unit (MMU) used in the IBM RT PC. The ROMP and MMU grew out of IBM's requirements in the late 1970s for a modern microprocessor for use in office equipment and small computers. Several major goals were identified at the start of the project.

- **High-Level Language Programming.** With software costs rising, it was decided that almost all programming for the new processor should be done in a high-level language because of its greater efficiency of programming. This meant that a good compiler was needed in conjunction with the processor. In fact, an excellent compiler was needed—one that would produce the tightest possible object code, to reduce the size of ROM and RAM storage required for office machines.

- **Addressability.** Sixteen-bit computers are limited to addressing 64K bytes or words unless some additional hardware, such as segment registers, is introduced in the addressing path. The difficulty of handling objects larger than 64K even with segment registers led to the decision to make the ROMP an all 32-bit machine, with 32-bit registers, 32-bit addresses, and 32-bit data quantities.

  The ROMP and MMU have segment registers, but they are used for different purposes than in typical 16-bit computers. Each segment can span 256 megabytes; the segment registers are used to provide addressability to a number of different objects rather than to extend addressability beyond the first 64K bytes of an object.

- **Two Chips.** For cost reasons the number of VLSI chips in a small system must be minimized. Existing technology did not allow functions as complex as the ROMP and MMU to be combined into a single chip, so one chip was used for the processor and one for the Memory Management Unit. The split is about even; the two chips are of comparable complexity (the MMU is somewhat larger than the ROMP).

- **High Performance with Inexpensive Memory.** The 801 minicomputer [1], a Reduced Instruction Set Computer (RISC) then under development at the IBM Thomas J. Watson Research Center in Yorktown Heights, New York, had exceptionally high performance. However, much of its performance depends on its two caches, which can deliver an instruction word and a data word on each CPU cycle. Since such caches were prohibitively costly for small systems, pipelining techniques normally found in larger machines were adapted to the ROMP so that useful work could be done during the (comparatively) long time needed for memory operations. The techniques include asynchronous prefetching and partial decoding of instructions, a packet-switched channel between the ROMP and the MMU, execution of instructions beyond a "load" until the loaded data is actually needed, and delayed branches which overlap the execution of another instruction with the fetching of the branch target.

- **Virtual Memory.** This requirement was identified later than the others, after it was realized that the ROMP had the potential of being used in much more elaborate systems than just office machines. The virtual addressing mechanism provides 2^{40} bytes of virtual addressability and supports real memory sizes of up to 16 megabytes. It uses concepts from the System/38 [2], and additionally provides a means of controlling access to sections of virtual memory smaller than a page for assistance in database locking schemes.

The PL.8 compiler [3], which was developed at IBM Research in conjunction with the 801 architecture, offered the potential of generating extremely efficient code for a machine which matched its paradigm of a computer. Thus, the ROMP programming model and instruction set are derived from the 801 processor for which the PL.8 compiler was originally designed, but the ROMP is designed for greater byte efficiency (the programs are smaller) than the 801. That the ROMP instruction set is a good target for a compiler is demonstrated by the fact that the PL.8 compiler generates code that is within about 10% of the size of good hand code.

Together, the ROMP and MMU implement a system with the following major characteristics:

- A large uniformly-addressed virtual memory (2^{40} bytes)
- A large number of general-purpose registers (16)
- A simple, uniform instruction set with most instructions executing in a single cycle.

As with other RISC designs [4, 5], the ROMP instruction set performs all operations on data within general registers; the only memory operations provided are Load and Store. The compiler "pipelines" the Load operations by separating them from the use of the loaded data as far as possible.

Although most ROMP instructions execute in only one cycle, additional cycles are taken when it is necessary to wait for data to be returned from memory for Loads and Branches. As a result, the ROMP takes about three cycles on the average for each instruction. At the cycle time of 170 nanoseconds used in the RT PC, the ROMP runs at about 2 MIPs.

Details of the ROMP and MMU architecture are described in the following sections.

ROMP Processor
The RT PC ROMP processor was designed to:

- Provide an architected address and data width of 32 bits
- Provide an efficient target for an optimizing compiler
- Support virtual memory
- Provide system integrity through separate user and supervisor states
- Provide improved error detection and reporting facilities
- Provide high performance with low-cost memory.

The first requirement dictated an architecture providing both 32-bit address and data quantities. As a result, it was decided that all registers and computations would support 32-bit quantities. However, the architecture provides for specific support of 8-bit and 16-bit quantities in addition to 32-bit quantities.

The ROMP processor architecture was defined with the assumption that most software would be developed in a high-level language. At this same time, an optimizing compiler was being developed at IBM Research which supported a variant of the PL/I programming language. A joint study was conducted to evaluate this compiler and the architectural requirements to take advantage of the compiler optimization techniques. This study indicated the need for a large number (16 or 32) of 32-bit general-purpose registers, and an instruction set closely matched to the compiler intermediate language. Specifics of the resulting instruction set are provided later in this paper.

During the architecture definition, it became clear that systems using processors of this class must provide virtual memory. In order to support virtual memory, precise interrupts were defined for the ROMP so that the cause of a page fault can be identified easily. All instructions are restartable; an instruction causing a page fault can simply be re-executed after the fault is resolved. This sort of virtual memory support is common on mainframes and some minicomputers, but had not appeared in a microprocessor prior to the design of the ROMP.

The need to provide protection of user programs and isolation of control program functions resulted in the definition of separate user and supervisor states. Only instructions which cannot be used to affect system integrity are valid in user state. Instructions associated with control program functions are valid in supervisor state only.

In order to guarantee data integrity, certain requirements and facilities are provided for error detection and reporting, including:

- Parity checking on all external buses
- Bus timeout detection
- Non-maskable hardware error detection interrupts.

Good system performance with low-cost memory was an early requirement. Although cache memories were considered, they were quickly discarded due to their cost and complexity. A compromise was made between cost and performance that resulted in the decoupling of memory operations from CPU operations, and in the definition of an innovative high bandwidth packet switching storage channel that supports multiple outstanding operations. The MMU was designed to allow overlap of the address translation process with memory access. The MMU also supports two-way interleaved memory which provides a throughput of one memory operation every CPU cycle.

Programming Model
The ROMP provides 16 32-bit General Purpose Registers (GPRs) that can be used for either address or data quantities. There are no restrictions on which registers can be used for addresses or data. Figure 1 shows the 16 GPRs.

Note that the 16 GPRs are also grouped in eight pairs. These pairs (0-1, 2-3, etc.) are used with the paired shift instructions to provide nondestructive shift capability. Details of the paired shifts are provided in the instruction set section of this paper.

A 32-bit register quantity can be treated as either a full 32-bit quantity, two 16-bit quantities, or four 8-bit quantities. Instructions are provided to manipulate data in any of these forms.

In addition to the 16 32-bit GPRs, a separate set of System Control Registers (SCRs) is provided. The SCRs include the following:

- Three registers associated with a 32-bit system timer facility
- The Multiplier Quotient (MQ) register used with the multiply and divide step instructions

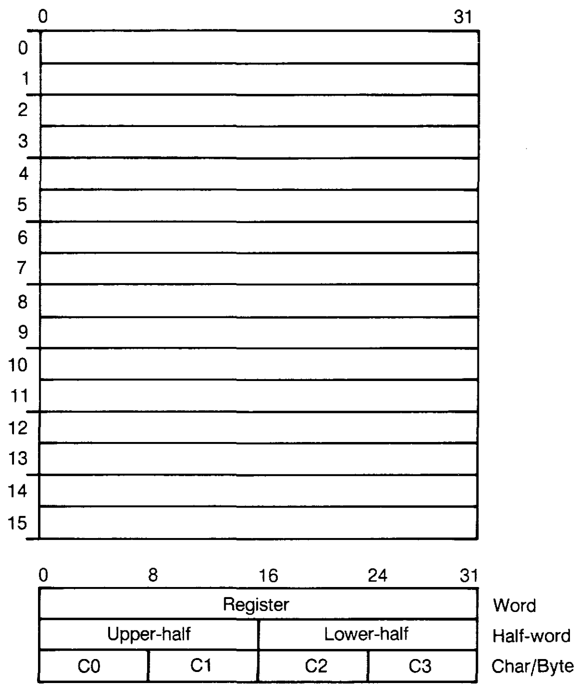


Figure 1 General-Purpose Registers

- The Machine Check Status (MCS) and Program Check Status (PCS) which are used to report hardware errors and software errors and exceptions respectively
- The Interrupt Request Buffer (IRB) used for posting interrupts
- The 32-bit Instruction Address Register (IAR)
- The Interrupt Control Status (ICS) register used for controlling interrupts and interrupt levels, address translation, memory protect, and other miscellaneous control functions
- The Condition Status (CS) which contains the condition code bits.

Instruction Set
The ROMP is generally a two-address architecture, with both 2- and 4-byte instructions of seven formats as shown in Figure 2. The various formats provide an opcode field, register fields (RA, RB, and RC) and an immediate field (I, JI, BI, and BA). RA, RB, and RC are each 4-bit fields which specify one of the 16 GPRs.

Although most ROMP instructions are two-address, the X format provides three register addresses. In all other formats, the RB and RC fields specify the source registers, with RB also specifying the destination register. A single instruction called Compute Address Short (CAS) is implemented in the X format, where the contents of registers RB and RC are added together and the sum placed in

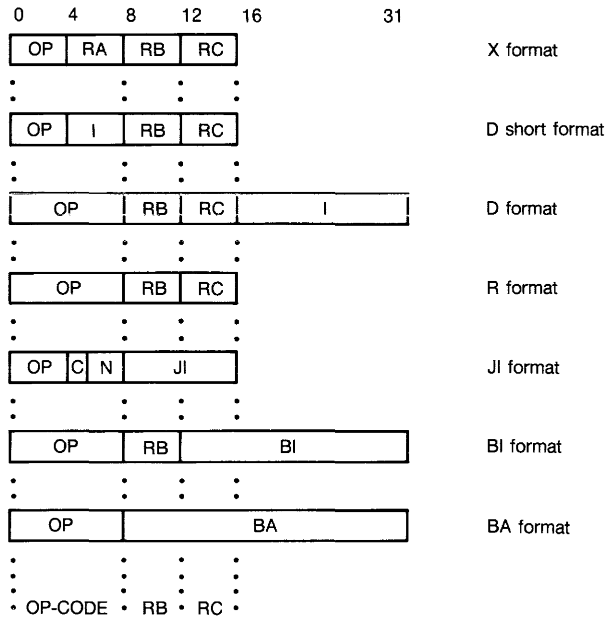


Figure 2 Instruction Formats

register RA. Extensive studies indicated the need for a three-address add instruction for address computations so that both source register quantities could be preserved.

The various instruction formats were defined so that the opcode and two register fields (RB and RC) are always in the same bit positions within each instruction format. This allows these fields to be used as defaults to unconditionally control fetching of instruction microcode and register operands without instruction pre-decoding. This is necessary to support a goal of single-cycle execution of each instruction. Note that in certain formats (JI, BI, BA for example) one or both register fields are not used. However, these fields are still used to fetch register operands. During the execute phase of instruction processing, a decision is made to use the immediate information rather than the register quantities. Since this decision is not made until the execute phase, the register information can be fetched by default and later discarded with no undesirable results. This approach is required to achieve the goal of single-cycle instruction execution, without creating implementation constraints.

During the definition of the ROMP instruction set, several studies were conducted to determine the frequency of use of each proposed instruction. These studies indicated that certain instructions (Increment, Immediate Shift, Short Branch, Loads and Stores with small displacements, etc.) were very heavily used. Some of these were defined as 2-byte instructions in order to achieve the desired byte efficiency and to reduce the memory bandwidth requirements of the processor to less than one word per cycle. Four-byte versions of certain of the 2-byte instructions were also defined for completeness that allowed a 16-bit immediate field instead of the 4-bit immediate field provided by the 2-byte format. Several evaluations were made trading off the byte efficiency of the 2-byte instructions versus their limited displacement capability. The final instruction set definition included 79 2-byte instructions and 39 4-byte instructions. Ongoing analysis of compiler-generated code indicates an average instruction length of 2.4 to 2.7 bytes, indicating good use of the 2-byte formats.

In certain formats (X, D Short, and JI) a 4-bit opcode is used. Opcodes were chosen so that these particular formats could be easily determined with a minimum of pre-decoding.

The ROMP provides a total of 118 instructions in the following ten classes:

Instruction Class	Number of Instructions
1. Memory Access	17
2. Address Computation	8
3. Branch and Jump	16
4. Traps	3
5. Moves and Inserts	13
6. Arithmetic	21
7. Logical	16
8. Shift	15
9. System Control	7
10. Input and Output	2
Total	118

The Memory Access instructions permit loading and storing data between the 16 GPRs and main memory. These instructions support four types of data:

- 8-bit (character) quantities
- 16-bit (halfword) quantities
- 16-bit algebraic (sign extended halfword) quantities
- 32-bit (fullword) quantities.

Load and Store Multiple instructions are also included in this class that permit loading or storing of from one to 16 of the GPRs to memory. A test and set instruction is also provided for multiprocessor synchronization.

All Memory Access instructions compute the effective memory address as the sum of a GPR contents plus an immediate field specified in the instruction (base + displacement addressing). Two-byte memory access instructions provide a 4-bit immediate field, with 4-byte instructions providing a 16-bit immediate field.

The Memory Access instructions operate only between memory and one or more GPRs. No memory-to-memory operations are provided. The architecture allows instruction execution to continue beyond a load instruction if subsequent instructions do not use the load data. This increases system performance by overlapping memory access with subsequent instruction execution.

Address Computation instructions are provided which compute memory addresses without changing the condition codes. These instructions include a three-address add instruction (Compute Address Short), Increment, Decrement, and 2- and 4-byte instructions which permit loading a GPR with a 4-bit or 16-bit immediate value respectively. Separate Compute Address Lower and Compute Address Upper instructions are provided to load a 16-bit immediate value into either the lower half or upper half of a GPR. Two Address Computation instructions are provided specifically to aid in the emulation of 16-bit architectures. They allow computing a 16-bit quantity that replaces the low-order 16 bits of a GPR without altering the upper 16 bits.

Standard Branch and Jump instructions are provided for decision making. Two-byte Jump instructions are provided that provide a relative range of plus or minus 254 bytes. Four-byte Branch instructions provide a range of up to plus or minus 1 megabyte. A group of Branch and Link (BAL) instructions is also provided for subroutine linkage.

A delayed branch (called "Branch with Execute") is provided to allow overlap of the branch target fetch with execution of one instruction following the branch (called the subject instruction). Execution of the subject occurs in parallel with fetching of the target instruction, thereby eliminating dead cycles that would normally occur during fetching of the target instruction.

Three Trap instructions are provided for run-time address checking. These instructions compare a register quantity against a limit, and cause a program check interrupt if the limit is exceeded.

The Move and Insert class of instructions support testing the value of any bit in a GPR, and the movement of any of the four 1-byte fields in a GPR. A Move instruction is provided that allows moving any one of the 32 bits in a GPR to a test bit in the condition status register, with a corresponding instruction that moves the test bit value to any of the 32 bits in a GPR. A series of Move Character instructions are included that move any of the four 1-byte fields in a GPR to another 1-byte field in a GPR.

The Arithmetic class supports standard Add and Subtract operations in both single and extended precision modes. Other instructions in this class include Absolute Value, Ones and Twos Complement, Compare, and Sign Extend. Also, Multiply Step and Divide Step instructions are provided. The Multiply Step instruction produces a 2-bit result per step, and can be used to construct variable length multiply operations. The Divide Step instruction produces a single bit result per step, and can be used to construct variable length divide operations.

The Logical class provides AND, OR, XOR, and negation operations using two register quantities or one register and an immediate value. A group of Set and Clear Bit instructions is also included in this class that allows any bit in any GPR to be set to one or zero.

The Shift class provides Algebraic Shift Right, Shift Right, Shift Left, and left and right paired shifts. Shift amounts from 0 to 31 bits can be specified as either an immediate quantity in the instruction, or as an indirect amount using the value in a GPR. The concept of paired shifts was introduced to provide non-destructive shifts that shift a specified GPR a given amount, and place the result in a different register (the "twin" of the source register) without altering the source register. The twin of a given register is determined by complementing the low-order bit of the register number (i.e., the twin of register 4 is 5, the twin of 11 is 10, etc.).

Instructions in the System Control class are generally privileged instructions that are valid only in supervisor state. Included in this class are instructions that move GPRs to and from SCRs, set and clear SCR bits, Load Program Status, and Wait for interrupt. Also included is a nonprivileged Supervisor Call instruction.

Two instructions that load and store GPRs to I/O devices are included in the Input and Output class. These instructions are normally used to access control registers in the MMU or other system elements.

Interrupt Facility

The ROMP implements a priority-based interrupt scheme supporting seven external interrupt levels. In addition, two error reporting interrupt levels are also provided. The program check level is used for reporting software errors and exceptions such as page fault, protection violations, and attempted execution of a reserved opcode. The machine check level is used to report hardware failures such as bus parity errors, uncorrectable memory ECC errors, and bus timeouts.

The interrupt facility includes old and new program status words (PSWs) similar to those of System/370. Each PSW pair contains the IAR, condition status, and interrupt control information. Hardware automatically performs a PSW swap when an interrupt occurs. GPRs are not automatically saved by hardware, with system software using a Store Multiple instruction to save required GPRs. A Load Program Status (LPS) instruction is provided that automatically restores the machine state from the old PSW once interrupt servicing is complete.

Memory Management Unit

The MMU combines the functions of virtual addressing support and memory control. From the system point of view, it translates virtual addresses to real addresses, implements the memory protection model, performs "lock-bit" processing (explained below), and provides interrupts to the ROMP for exceptional conditions such as page faults.

As a memory controller, the MMU is responsible for the hardware-level control of up to 16M bytes of RAM and ROM. Separate controls are provided for RAM and ROM that support different speed memories and allow interleaving of RAM for improved memory bandwidth. Internal logic is provided to support Error Correcting Code (ECC) for RAM and parity for ROM. The MMU also provides control signals for the external Reference-and-Change Array (R/C).

The support for ECC on the RAM is new in the microprocessor field and is a reflection of the large memory sizes expected to be used with the ROMP. The lockbit mechanism provides a sub-page-level protection and locking mechanism and is new with the MMU.

Virtual Address Translation
Figure 3 shows the memory model which the MMU implements. When the ROMP is operating in Real mode (Translate bit in Interrupt Control Status off), the MMU functions simply as a hardware memory controller. In this mode, up to 16M bytes of real memory can be addressed. When the ROMP is in Translate mode (Translate bit in ICS on), the MMU translates each address from the ROMP from virtual to real and then uses the real address to access the memory arrays. Memory access from adapters on the I/O Channel is also supported by the MMU, with a control bit for each access indicating whether the access is real or translated.

Program virtual addresses generated by the ROMP are 32 bits. These are expanded in the MMU to 40 bits by using the high-order 4 bits of the program virtual address to select one of 16 Segment Registers (SRs), and then concatenating the 12-bit Segment Identifier (SID) contained within the SR to the remaining 28 bits of the incoming address. To the executing program, memory appears to be 4 gigabytes of virtual memory broken into 16 segments of 256M bytes each.

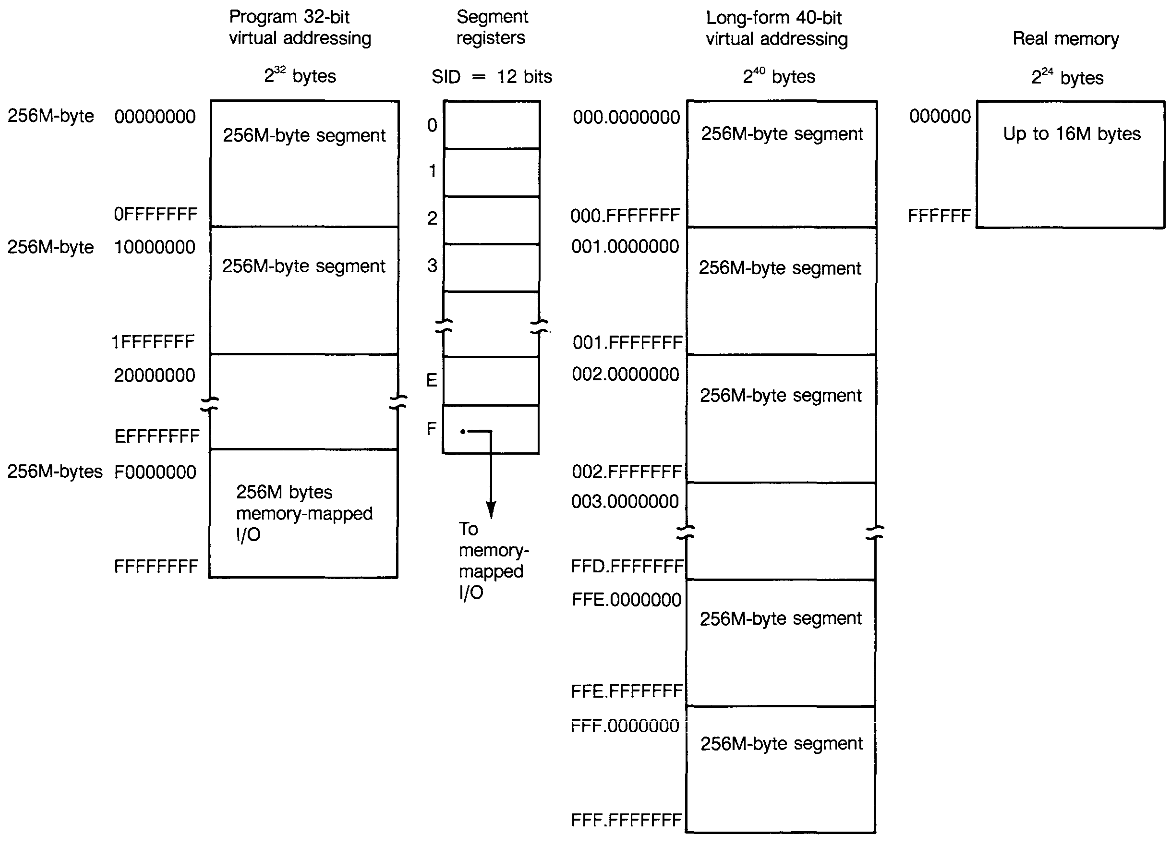


Figure 3 Storage Model

The largest addressable entity is normally the 256M-byte segment, but system software can construct larger objects by (for example) assigning consecutively-numbered SIDs to adjacent SRs, creating an object whose maximum size is any multiple of 256M bytes up to 4 gigabytes. Objects larger than 4 gigabytes will require special techniques.

The Segment Registers reside in the MMU, and can be read and written from the ROMP by supervisor-state programs, using I/O instructions. Each SR contains the following:

- Segment Present bit
- ROMP Access Protect bit
- I/O Access Protect bit
- 12-bit Segment ID
- Special Segment bit
- Key bit.

Because the virtual addresses generated by programs are only 32 bits long, while translation is performed on 40-bit virtual addresses, each program is restricted to addressing only those segments supplied to it (i.e., SIDs loaded into SRs) by the operating system. Segments can be shared between processes by placing the same SID value into an SR for each process (not necessarily the same SR).

The 40-bit virtual addresses are translated to real by looking them up in an Inverted Page Table (IPT) as shown in Figure 4. The table is "inverted" because it contains one entry for each real memory page rather than one per virtual page. Thus a fixed proportion of real memory is required for the IPT regardless of the number of processes or virtual segments supported. To translate an address, a hashing function is applied to the virtual page number (high-order part of the 40-bit virtual address, less the byte offset) to obtain an index to the Hash Anchor Table (HAT). Each HAT entry points to a chain of IPT entries with the same hash value. A linear search of the hash chain yields the IPT entry (and thus the real page number) which corresponds to the original 40-bit virtual address. If no such entry is found, then the virtual page is not mapped and a page fault interrupt is taken.

The hashing technique results in chains which are typically short—between one and two entries. Even so, translating a virtual address using only the HAT and IPT would require several memory accesses for each translation. In order to eliminate most of the IPT searches, the MMU maintains a cache of recently-translated addresses in a Translate Look-aside Buffer (TLB). The TLB is two-way set-associative, with 32 entries. If the required entry is in the TLB, then the MMU can complete its translation in one cycle. If a TLB

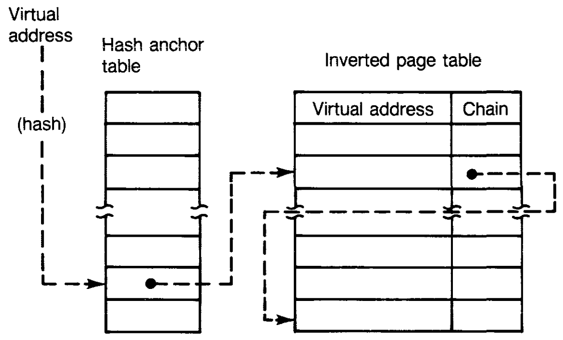


Figure 4 Hash Anchor Table and Inverted Page Table (Conceptual)

"miss" occurs, then the MMU automatically searches the IPT and reloads the least recently used entry for the appropriate congruence class. This typically adds 8 to 11 cycles to the translation time. An IPT "miss" is a page fault.

The MMU provides functions for use by supervisor-state software which cause selected entries in the TLB to be purged and thus reloaded from the IPT the next time they are needed. Such purging is required at certain times to keep the TLB contents synchronized with changes to the IPT. Purging can be done for an individual page, for an entire segment, or for the entire TLB. A task switch by itself does not require that any of the TLB be purged; only the segment registers need be loaded. A "load real address" function is also provided for supervisor-state software to allow determining the real address corresponding to a given virtual address.

Memory Protection

Several functions performed by the MMU combine to provide memory protection for programs running with address translation on.

In order for the MMU to respond to a virtual address at all, the Segment Present bit in the appropriate SR must be set. This feature is used not only for protection, but to provide memory-mapped I/O by arranging for the address range covered by one or more SRs to be ignored by the MMU but responded to by the I/O Channel Controller.

For SRs which have the Segment Present bit set, access through the MMU is controlled by the settings of the ROMP Access Protect and I/O Access Protect bits. A segment register can thus be assigned to the ROMP processor, to I/O devices via the I/O channel controller, to both, or to neither.

For virtual accesses which are allowed by the control bits described above, one of two types of memory protection is applied. Which one to use is determined by the Special Segment bit in the SR. If this bit is 0, then a key matching scheme adapted from System/370 is used. Processes are given key 0 or key 1 access to segments via the Key bit in each SR. Individual pages have 2-bit keys in their IPT (and thus TLB) entries. The types of access allowed are defined by the following table:

Page Key	Type of page	SR Key	Load	Store
00	Key 0 fetch-protected	0	Yes	Yes
		1	No	No
01	Key 0 read/write	0	Yes	Yes
		1	Yes	No
10	Public read/write	0	Yes	Yes
		1	Yes	Yes
11	Public read-only	0	Yes	No
		1	Yes	No

If the Special Segment bit in the SR is 1, then a finer granularity of protection is applied. Each page is considered to be made up of 16 "lines" of 128 bytes each (with 2K-byte pages) or 256 bytes each (with 4K-byte pages). Access to a particular line within a page is controlled by the value of a "lockbit" associated with each line. Each IPT (and TLB) entry contains 16 lockbits for each page, a "write" bit that determines how the lockbits are interpreted, and an 8-bit Transaction Identifier (TID). In addition, the MMU contains a Current TID register which is loaded by the operating system when a process is dispatched. Access to the lines within pages of "special segments" is determined by the following table:

Current TID compared to TID in IPT	Write Bit in IPT	Lockbit value for selected line	Access permitted	
			Load	Store
Equal	1	1	Yes	Yes
		0	Yes	No
Equal	0	1	Yes	No
		0	No	No
Not Equal	—	—	No	No

Use of the Special Segment facilities allows interprocess locking of items smaller than pages. Interrupts caused by disallowed accesses can be used to grant locks, to cause processes to wait, or to indicate actual protection violations.

Conclusions

Together, the ROMP and MMU provide many of the characteristics of a mainframe computer—large virtual memory, 32-bit addressing and data flow, high performance—while requiring only a few chips to implement. The Reduced Instruction Set Computer concept allows the hardware architecture to be relatively simple, while the combination of ROMP and MMU hardware and modern system software can produce a very powerful computer system which is small enough to reside on a desk top.

The realm of personal computers need not be restricted to those applications which can fit on 8- or 16-bit machines with limited addressability and CPU speed. Programs typically considered mainframe applications can take advantage of the addressability, virtual memory, and speed of the ROMP.

References

1. George Radin, "The 801 Minicomputer," *Proc. of Symposium on Architectural Support for Programming Languages and Operating Systems,* Palo Alto, CA, March 1-3, 1982. Published in *ACM SIGARCH Computer Architecture News* Vol. 10, No. 2, March 1982. Also published in *IBM Journal of Research and Development* Vol. 27, pp. 237-246 (1983).
2. G.G. Henry, "Introduction to IBM System/38 Architecture," *IBM System/38 Technical Developments,* IBM Corporation, Atlanta, GA, 1978.
3. M. Auslander and M.E. Hopkins, "An Overview of the PL.8 compiler," *Proc. of the SigPlan '82 Symposium on Compiler Writing,* Boston, MA, June 23-25, 1982.
4. J. Hennessy, N. Jouppi, S. Przybylski, C. Rowen, T. Gross, F. Baskett, and J. Gill, "MIPS: A Microprocessor Architecture", *Proc. of the SigMicro 15th Annual Microprogramming Workshop,* 1982.
5. D.A. Patterson, "Reduced Instruction Set Computers", *CACM 28,* 1 (January 1985).

ROMP/MMU Implementation

D.E. Waldecker, C.G. Wright, M.S. Schmookler, T.G. Whiteside, R.D. Groves, C.P. Freeman, A. Torres

The IBM RT PC central processor and memory management functions are implemented in the ROMP and MMU chips. These VLSI parts are manufactured by the IBM General Technology Division in Burlington, Vermont and are contained on the RT PC processor card. The technology is discussed by Dupont et al.[1]. See also Waldecker et al.[2] for a description of the RT PC processor card. The basic challenge in the ROMP/MMU functional design was to organize the chips and interfaces to execute one instruction every 170-nanosecond processor cycle, except for a few types of instructions. This execution rate is maintained with relatively slow memory by interleaving memory accesses (see also Rowland [3]). ECC checking is performed on each memory access without impacting the ability to execute an instruction every cycle. Functional highlights of the ROMP and MMU chips, as well as their interaction and the logic design process used, are presented in this article.

The ROMP chip fetches and executes instructions. It also manages instruction flow for interrupts, program checks, and machine checks. The MMU accepts memory requests from the ROMP or from I/O, translates memory addresses from virtual to real, manages memory page faults, performs memory access authorization checking, and interfaces to the RT PC system memory cards. Both parts contain checking to generate machine checks and program checks in case of parity errors, invalid instruction operation codes, invalid real-memory addresses, and time-out conditions caused by non-response of a system element.

ROMP-MMU Interconnection

The ROMP and MMU chips are connected via a high-performance channel. The ability of the ROMP RISC architecture to execute an instruction almost every CPU cycle highlights the need for this channel to support high bandwidth. Two primary considerations for a high-performance memory interface are: 1) the ability to support a high transfer rate, and 2) low latency on replies to requests.

The ROMP Storage Channel (RSC) is a packet-switched, 32-bit channel designed to match the pipelined operation of the ROMP processor. Its most distinctive aspect is the ability to support overlapping of memory accesses. Memory reads are split into two distinct actions, a request and a reply, and multiple outstanding requests are allowed. An address packet and a data packet can be transmitted each 170-nanosecond processor cycle, giving a channel bandwidth of 23.5M bytes per second.

Address and data transfers on the RSC are done on a synchronous basis. The 170-nanosecond CPU cycle is divided into two transfer periods. The first half of each CPU cycle is dedicated to address transfers and the second half to data transfers. The address cycle and the data cycle each have independent arbitration since the address transfer and the data transfer may be for independent operations. Independent acknowledgment is also provided for the address and data cycles. A transfer may not be accepted if the recipient is "busy" or if the recipient detected a parity error on the transfer. The acknowledgment sequence will indicate these conditions and the source will re-send the address or data in the case of "busy" or "error." Data replies from memory or I/O have priority over data being sent to memory or I/O. This priority is necessary to prevent potential overflow of the buffering available in the receiving device. The ROMP takes advantage of the system memory interleave capability in two areas. First, it prefetches instructions and is capable of sending out four instruction fetch requests before requiring a reply. Because channel bandwidth is inherently greater than necessary for instruction execution, the 4-word prefetch buffer is usually at least partially full, providing some immunity to interference from program data references and DMA traffic.

Secondly, for data references to memory or I/O the ROMP allows two outstanding requests. Program execution is stopped only if the data being referenced is required to continue. Instructions following a Load are executed in parallel with the actual memory reference unless they require the data being fetched by the Load instruction. The PL.8 compiler places Load instructions in the instruction stream as far ahead of the instruction using the data as it possibly can. Also, Load data is placed into the ROMP register file through a dedicated port to eliminate interference.

Of course, the MMU and memory must also support fully overlapped operation if the performance potential is to be achieved. In fact, the MMU can simultaneously be processing two memory requests while the result of a third is being transmitted to the ROMP. The memory in the RT PC system is implemented as two interleaved 32-bit wide banks, allowing overlapped operation. The RT PC system memory card design supports interleaved operation on a single memory card.

Management of requests and replies on the RSC is accomplished by having a unique tag accompany each request and each reply. Whenever the ROMP issues a memory request, it sends a unique 5-bit tag with the request. The MMU remembers the tag and returns it with the requested data or instruction. The tag received with each reply allows the ROMP to handle the received data or instruction correctly. The same tagged approach is used for requests from the ROMP to the I/O or from I/O to the MMU.

It can be seen that ROMP instruction execution is relatively "decoupled" from memory requests. Instructions are prefetched and the ROMP usually has about two instructions in its prefetch buffer. Instruction execution is not suspended on Stores. That is, after the ROMP sends the address and data to the MMU, it is free to continue executing instructions. If the compiler is successful in placing Loads "upstream" in the program, the ROMP also need not wait for requested data in order to continue execution. In the current implementation of

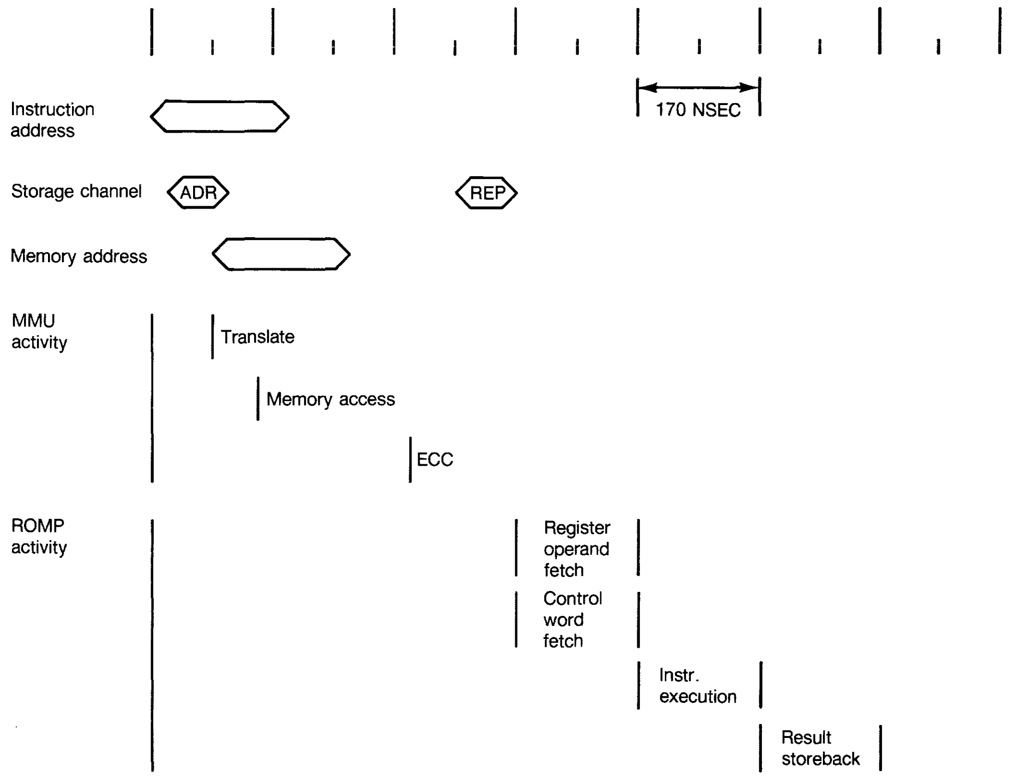


Figure 1 ROMP/MMU Instruction Execution Flow

the ROMP, the execution overlap of Loads and Stores is suspended when operating with virtual addresses, but overlap continues for instruction fetches. The ROMP generally requires 60% to 70% of the RSC bandwidth to support its inherent performance. The remainder of the RSC/Memory bandwidth is available for I/O DMA traffic. Even though DMA traffic may interfere with ROMP memory requests, the decoupled nature of the ROMP and memory can permit such interference without performance degradation in many cases. That is, in many cases it does not matter if a ROMP instruction fetch or a data load is delayed a cycle. Of course, as I/O traffic causes the RSC usage to approach 100%, ROMP performance starts to be degraded linearly by increased I/O activity.

Figure 1 shows the typical operations required to execute a single ROMP instruction. The sequence begins with the fetching of an instruction from memory. The request is sent from the ROMP and accepted by the MMU one-half cycle later. The MMU translates the address, uses the result to address memory, checks the returning data for ECC errors, and latches the result. The 2-cycle memory access time includes address translation, address and data buffering, and ECC error detection. Error correction is not included in this time, however. (If an error is detected, the reply to the ROMP is cancelled and retransmitted to the ROMP on a subsequent cycle. This practice reduces the impact of ECC on access time when no error is detected from 80 nanoseconds to about 30 nanoseconds.) After access, this instruction is returned to the ROMP, and is latched into the instruction prefetch buffer. Following latching of the instruction, a three-cycle process occurs. First, the ROMP extracts the opcode, register addresses, and any immediate operand. The opcode is used to address a small microcode ROM and the register addresses reference two independent read ports in the register file. At the end of this cycle, register and immediate operands and control information have all been latched. The following cycle, operands are manipulated in the ALU according to the control information, with the result being latched at the end of the cycle. In the third cycle the result of the operation is stored back into the register file.

Although there are several cycles in the complete instruction sequence, the process is overlapped to a high degree. Figure 2 shows how the sequence described above would be overlapped for three identical instructions. Shown are three 32-bit instructions which execute in one cycle each.

Rapid response (i.e., low latency) to memory requests is most important for instruction fetches which directly follow a branch-taken execution. The contents of the instruction prefetch buffer must be discarded in this case. Although latency of starting the new instruction stream is fixed, the ROMP reduces its impact by implementing a "Branch with Execute" instruction. This Branch form is designed to keep the CPU busy with instruction execution while the contents of the Branch address is being fetched from memory.

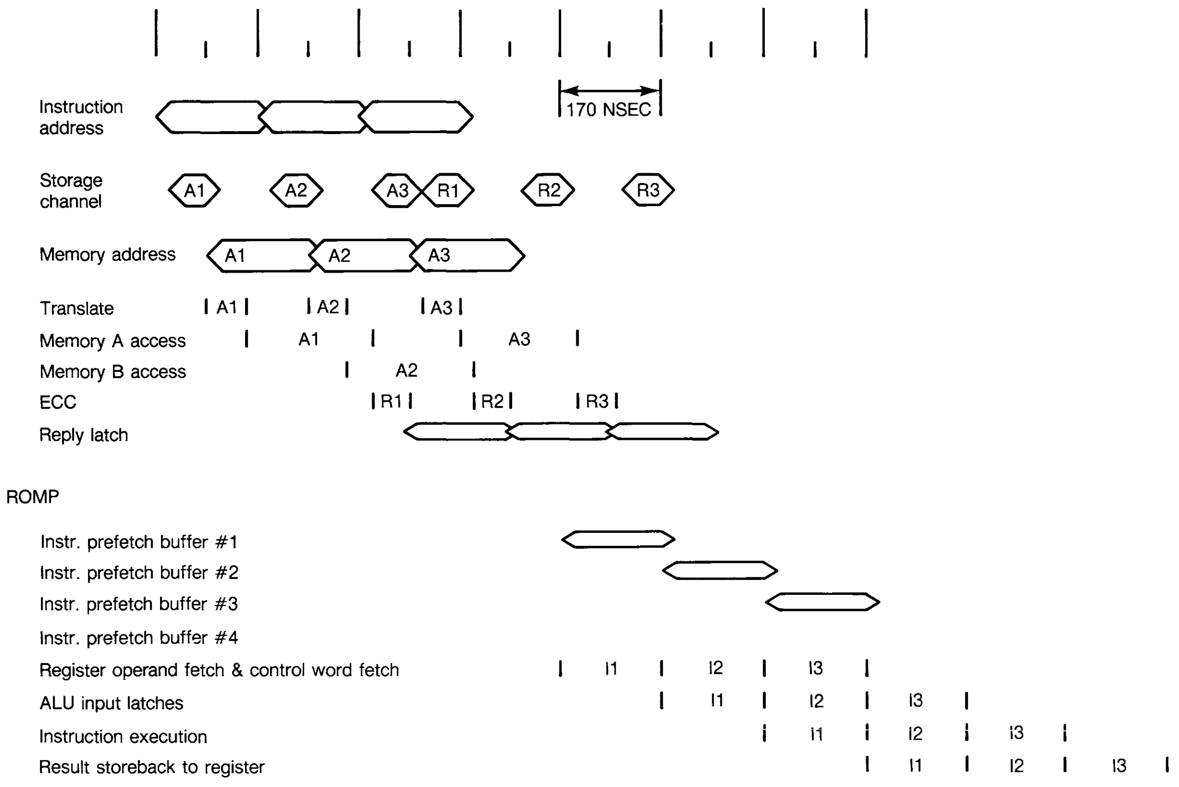

170 NSEC
Instruction address
Storage channel
A1
A2
A3
R1
R2
R3
Memory address
Translate
Memory A access
Memory B access
ECC
Reply latch
ROMP
Instr. prefetch buffer #1
Instr. prefetch buffer #2
Instr. prefetch buffer #3
Instr. prefetch buffer #4
Register operand fetch & control word fetch
ALU input latches
Instruction execution
Result storeback to register
I1
I2
I3

Figure 2 Timing Example --- Three 32 Bit Instructions

The following sections highlight some internal functions performed by the ROMP and MMU chips as well as the logic design and verification process used.

ROMP Chip

The ROMP chip is a pipelined processor capable of executing a new instruction every cycle. While one instruction is being executed, the next instruction can be decoded, and following instructions can be fetched from memory and buffered. A one-cycle execution rate is prevented when either an instruction requiring multiple execution cycles is executed, or a hold-off occurs (i.e., when the processor waits for an instruction or data from memory). Most instructions take only one cycle, but Store and I/O Write instructions each take two cycles. The ROMP instruction set is described in Hester, Simpson, and Chang [4].

The ROMP processor is partially microprogrammed. It uses ROM for control during the execution cycles, but hardwired control logic is used for instruction prefetching and for memory data requests, since those operations are usually overlapped with the execution of other instructions. Figure 3 shows a block diagram of the ROMP processor data flow.

Instruction Fetching

The instruction fetch area includes the Instruction Prefetch Buffers (IPBs), the IPB Multiplexer (MUX), and the Instruction Address Register (IAR) and its incrementers. Four IPBs are provided to keep the processor supplied with instructions. Instructions are prefetched whenever an IPB is available and there is no other request for use of the RSC. Every cycle, each IPB that is waiting for an instruction ingates the data from the RSC. During the following cycle, the tag associated with that data is examined to determine if it was addressed to any of the IPBs. If so, then that IPB will hold its contents until that instruction word is gated by the MUX to the decode circuits.

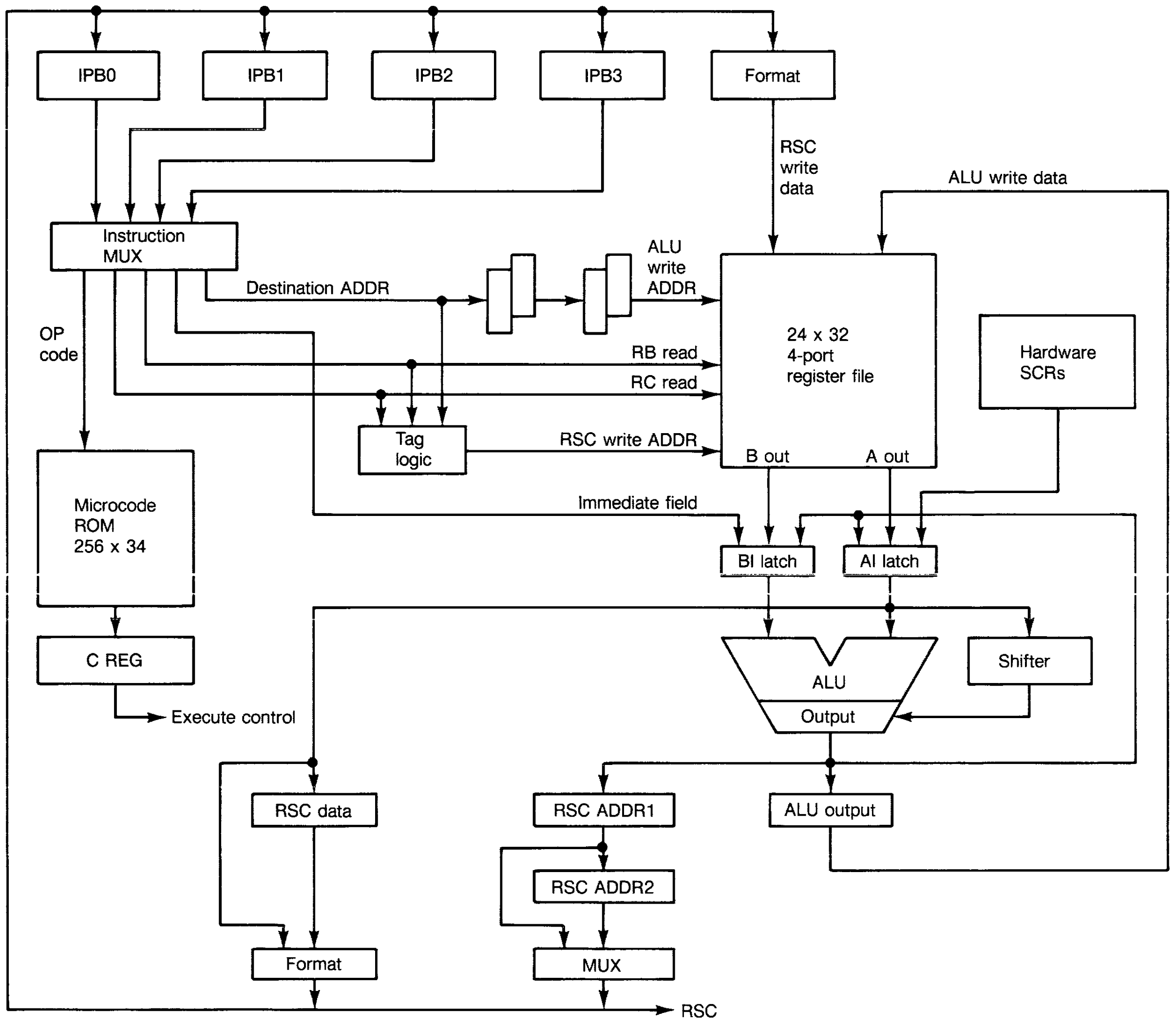


Figure 3 ROMP Data Flow

The IAR, which in many systems is called a Program Counter, is included among the hardware System Control Registers (SCRs) in the diagram, and has two incrementers. One is used to update the IAR after each instruction is executed, while the other is used to calculate the address of the next instruction to be prefetched.

Execution Unit

The execution unit includes the register file, the AI and BI latches, the ALU and shifter, and the ALU output latch. It also includes the MQ register and the Condition Status register, which are both SCRs. To support a one-cycle execution rate, a 4-port register file is used. The register file is a RAM containing the 16 general registers, some of the SCRs, and other registers which hold temporary values during the execution of some of the operations. Two of the ports of the register file are used to read two operands simultaneously into the AI and BI latches. Another port is used to write the result back from the ALU output latch, and the fourth port is used to write data from memory or I/O. Data is written into the register file during the first half of a cycle, and read out during the second half of a cycle. Therefore, if the same word is addressed by both a read port and a write port, the new data will be read out. The ALU is used for the execution of all arithmetic and logical instructions, and for address calculations when executing Load, Store, I/O and Branch instructions. The addresses are sent to the RSC request area. For Branches that are taken, the address is also placed in the IAR. During the second execution cycle of

Store and I/O Write instructions, the execution unit sends the data to the RSC request area.

A bypass is provided which allows the ALU output to be gated directly into either the AI latch or the BI latch. This bypass is used when the result from one execution cycle is needed during the next execution cycle. The bypass is activated by circuits which compare both read port addresses of the register file with the address that will be used for the result write port during the following cycle.

RSC Interface
The RSC interface consists of the request area and the receive area. The request area arbitrates for use of the RSC and it receives the acknowledgment signals after requests are sent. It also has buffer registers for two addresses and tags for requests to memory and I/O. It has one buffer register for outgoing data, and it aligns the data for byte and half-word Store and I/O Write instructions. Instruction addresses arising from successful Branches come from the execution unit, while instruction addresses which are in sequence come from the IAR prefetch incrementer. Requests will be re-sent if the acknowledgment signals indicate either a busy condition or a parity error.

The RSC receive area contains buffer registers for one incoming data word and tag. Each cycle, they capture whatever data word and tag is on the RSC. The data word is also gated into each IPB for which an instruction fetch request has been made. During the following cycle, the tag is examined to determine if the word is addressed to the ROMP, and if so, whether it is an instruction or data. If it is an instruction, the tag will also identify which IPB has been assigned to it. If it is data, the tag will point to one of two descriptors which will control the alignment by the formatter and store it into the proper location in the register file. The RSC receive area also checks the parity of the data word and sends back the proper acknowledgment signals on the RSC.

Control Unit
The control unit includes the microcode ROM, the Control Register (C Reg), the instruction decoders, and the ROM and register file address latches and control circuits. It also includes circuits for detecting and handling interrupts, machine checks, and program checks, as well as the SCRs associated with these events.

The ROM contains 256 control words of 34 bits each. It owes its small size to several factors. The reduced instruction set contains fewer than 128 instructions and most require only one execution cycle. The control words are needed only for the execution cycles of the instructions, since instruction prefetching is controlled by hardwired circuits, and the last control word of each instruction controls the decoding of the next instruction. Therefore, nearly half of the words are available for other operations, which include Program Status Word (PSW) swaps for interrupts, System Timer updating, and Power-on-Reset (POR) initialization. The initialization routine checks many of the internal facilities, including the Register File, ALU, and RSC, and goes into an endless loop if an error is detected. Another POR will cause it to try the initialization again.

Several techniques were used to keep the length of the control words short. Each word contains several encoded fields, with each field controlling a separate function or group of signals. Four different formats are used, so that some fields have different interpretations for each of the formats. The four formats are:

- ALU — Used for arithmetic and logical operations
- Shift — Used for shift and byte move operations
- Channel — Used for address calculations and RSC requests
- Control — Used for microcode branching and miscellaneous operations.

Use of a separate format for microcode branching eliminates having an address field in every control word. Since nearly all instructions use only one or two control words, microcode branching is seldom needed.

The control words are fetched from the ROM into the C Reg during the cycle prior to the one when they are executed. During the last execution cycle of any instruction, the next instruction is selected from one of the IPBs by the MUX and decoded. This decode cycle is used to simultaneously fetch a control word from the ROM and fetch the two operands from the register file into the AI and BI latches. The operation code is taken directly from the output of the MUX and used as the ROM address. This puts nearly all of the first control words into the upper half of the ROM. If additional words are needed, the ROM address is usually just incremented and switched to the lower half of the ROM. The control signals are obtained by decoding the outputs from the C Reg. Also during the decode cycle, the register address for the result, called the destination address, is put into a two-stage pipeline to be used two cycles later for storing the result from the ALU output latch into the register file.

System Control and Support Processor Facilities
The system control facility includes an SCR that controls the processor mode, the types of interrupts that are enabled, and interrupt priority. It has an SCR for buffering interrupt requests, and a System Timer facility for real time applications. There is also an SCR for identifying program check and machine check errors. With this, the ROMP can recover from most software and intermittent hardware errors.

The ROMP is a Level-Sensitive Scan Design (LSSD) processor which allows the internal registers and latches to be reconfigured as serial shift registers. A support processor, such as a PC, can then be used to examine and alter the internal state for system or program debugging. With other facilities, the processor can be stopped at a specified instruction or microcode address, or single stepped so that the state can be examined after each instruction. The processor can also be reset and initialized.

MMU Chip

The MMU implements the memory management features described in Hester, Simpson, and Chang[4] with a number of innovations. This article will focus on a few of these innovations. Figure 4, showing the MMU data flow, is included for reference. Refer to O'Quin et al.[5] for a software perspective of the MMU.

Translation Look-aside Buffer
Automatic hardware reload is included for Translation Look-aside Buffer (TLB) entries when virtual addresses are not present there.

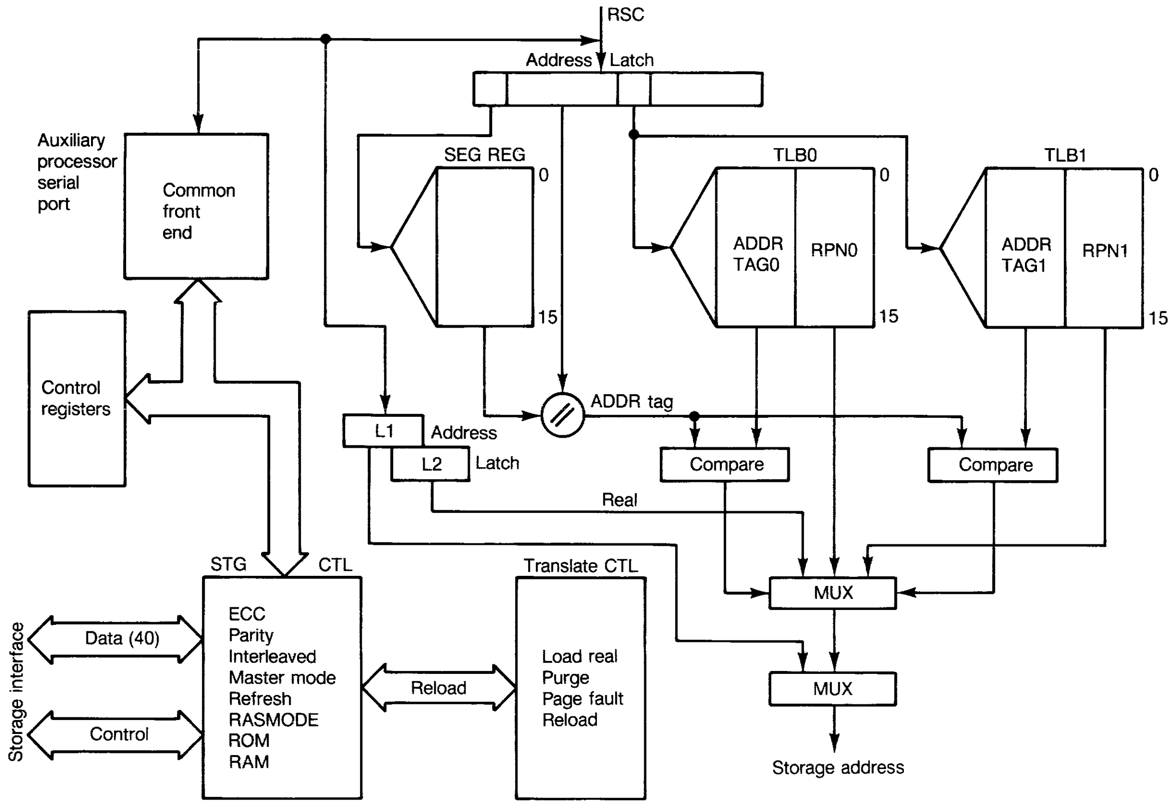


Figure 4 MMU Data Flow

Memory management units that depend on processor intervention are much slower due to the overhead in passing control to the processor which must then save and restore registers and return control to the memory management unit in addition to the page table memory lookup function. Figure 5 illustrates the significant performance advantage afforded by hardware TLB reloads as a function of the TLB hit ratio (the ratio of accesses without reload to total accesses).

MMU Memory Interface
Another key MMU feature is a flexible memory interface capable of supporting a wide variety of ROM and RAM. Memory interleaving for maximum bandwidth and error correction or parity checking are among the options supported by the hardware. External logic supplies all timing-critical memory controls to allow attachment of widely varying classes of memory. The interface employs a simple handshaking scheme that allows attachment of different access time memories to the MMU in a manner that overlaps data transfers with the dynamic RAM precharge. See also Waldecker et al.[2] and Rowland [3].

The flexibility built into the memory interface was not achieved without difficulties. For

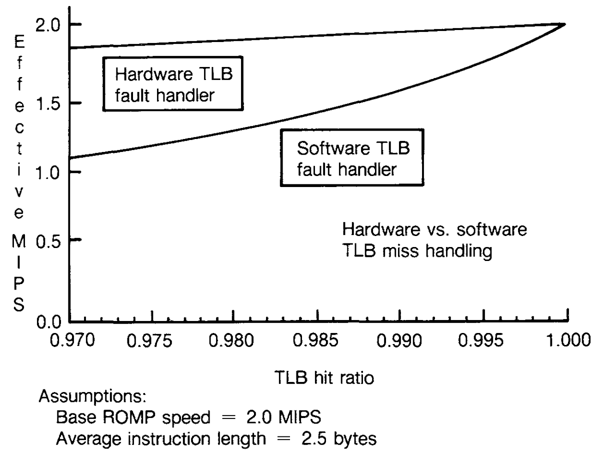


Figure 5 Hardware versus Software TLB Performance Handling

example, provisions for specifying RAM and ROM ranges in software created a "chicken or the egg" dilemma of how to run this range-setting software without the ranges being set. This led to the invention of "master mode" in which the MMU will accept any memory request before the range registers are initialized. If the first request after IPL is a memory read (such as a ROMP initial IAR fetch), this and all subsequent requests are directed to ROM until master mode is disabled by initializing the ROM and RAM range registers. The logic level on one of the pins at power-on determines whether master mode is effective to allow using more than one MMU on the RSC.

With any VLSI design, pins are always at a premium and generally force tradeoffs in performance. The MMU 24-bit memory address was implemented by multiplexing low and high address bits without incurring a performance penalty. The low order address bits are quickly passed through from the RSC and latched externally, in parallel with the address translation for the upper address bits. Thus pins are saved with no performance impact.

Many memory interfaces "hang" when an out-of-range address elicits no response from the memory controller. The MMU solves this problem with a special "Address Recognized" signal that is used to communicate invalid address conditions without machine error conditions.

Hidden refresh for dynamic RAMs is another MMU implementation feature. By performing refreshes to idle banks when possible, refresh interference is typically reduced by 50 percent.

Error Correction
The MMU provides an efficiently pipelined ECC function that separates error checking from error correction. Data transfer is done in parallel with the checking phase and is not delayed in the normal case (i.e., when no error is detected). If an error does occur, the data transfer is cancelled and extra time is allowed to complete correction. Thus, only accesses with errors are penalized by the extra cycle required for correction.

Logic Design Process
The ROMP logic design proceeded in parallel with the logic-circuit library definition and the SGP manufacturing process evolution in Burlington. The potential for significant changes required having flexible and automated tools for performance tuning, simulation, hardware verification, chip test generation, and chip problem isolation. The physical process is described in Dupont et al.[1].

Design Process
Figure 6 shows the general design flow used in the design of the ROMP and MMU.

Each logic designer simulated his logic macros interactively followed by full-chip simulation using the AUSSIM simulator. The full-chip simulation was driven by test programs written to cover as many variations of functional operation as we could reasonably exercise. New test programs were continually added as problems were discovered or new test conditions were identified. The same test programs were also used to test the nodal model as well as chip hardware when it arrived. (The nodal model

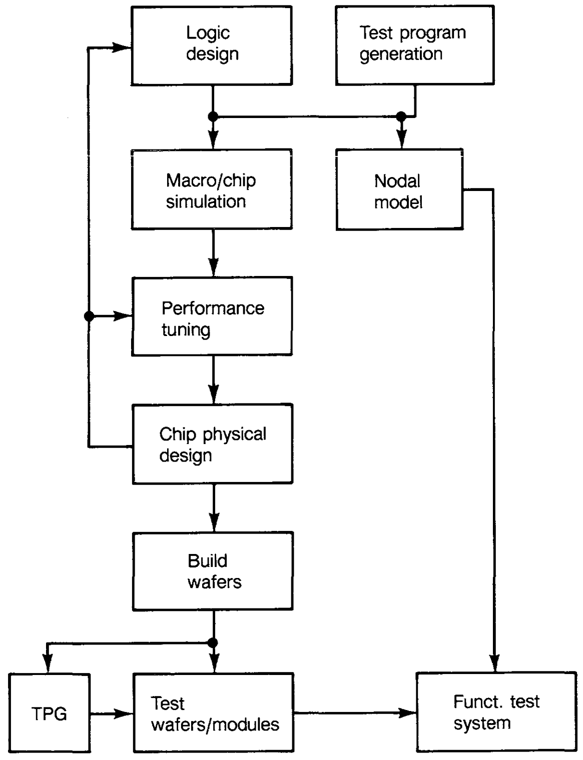


Figure 6 Design Flow

was a TTL equivalent of the ROMP chip and is discussed later in this article.)

Performance optimization was performed by using design automation tools which identified long paths which were then corrected by a combination of logic and physical design changes.

Manufacturing test was driven by test patterns generated in the design process. Failure isolation to areas of the chip was provided by test procedures and special programs written to analyze test results and identify sections of likely failure.

Early parts from Burlington with a potential for working were brought to Austin and functionally tested in our nodal model test bed by replacing the CPU portion of the model with the ROMP chip.

Performance Tuning

Early in the ROMP design, the need to estimate path delays automatically was recognized by Austin logic designers faced with thousands of nets to manage. To meet this need, a timing analysis tool was developed to estimate latch-to-latch paths based on block rise and fall delays. Block delay requirements were specified by the logic designer, and the physical designers adjusted the block power levels based on loading and wiring capacitance to meet the specified values. If the requested delay values could not be met, the logic designers would modify the design to improve the problem paths and the process would iterate.

Design Verification

The potential for significant changes led us to pursue two independent design verification methods — a nodal model and software simulation. A nodal model of the chip was built by automatically converting our SGP circuit library to TTL equivalent circuits. (Nodal means that every SGP signal was duplicated in the TTL design.) Simulation was performed using the Austin Interactive Simulator (AUSSIM).

After all of the logic macros were simulated individually, they were connected together and chip simulation was performed. High level simulation models were written for embedded arrays and for support logic to simulate a system environment. Architecture Verification Programs (AVPs) written in ROMP assembler language were used to test the functions described in our functional specification. Figure 7 shows the interconnection of external high-level simulation models with the ROMP low-level logic. The master and slave behaviorals simulate additional RSC traffic. The AVPs were loaded into our array model and executed by the ROMP. The first AVPs were written in Burlington, and Austin created more as they were required. The ROMP AVPs grew to over 400 and required eight CPU days to run on a 3081.

The nodal model helped us discover various subtle design bugs not found in logic simulation. The model was quite large, containing 14 boards of logic with an average of 280 TTL modules each. CPU support functions, including the memory, an I/O test board, and a test processor interface were part of the model. Debug of these areas permitted rapid functional test of newly-received chips replacing the CPU portion of the model with the ROMP chip.

Manufacturing Test Generation

A manufacturing test is performed on each chip on a wafer to determine which chips should receive further processing. Testers capable of probing over a hundred pads

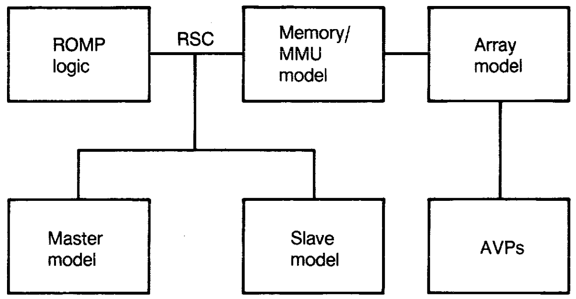


Figure 7 Design Verification

simultaneously and automatically applying hundreds of test patterns are used to perform this testing. To determine in manufacturing which chips in a wafer are defect free, test patterns are applied to each chip. As the number of circuits on a chip increases, the number of test patterns required to test all possible defects increases dramatically. The large number of circuits (>50,000) in the ROMP and MMU necessitated generating test data automatically. To do that, all latches in the ROMP and MMU were made Level Sensitive Scan Design (LSSD) latches. LSSD latches can be transformed into Shift Registers called scanstrings which allow test data to be shifted into and out of the chip by appropriate clocking. This approach supports automated testing by shifting in test data and then shifting out the test results. The tester compares test results with correct results previously generated during test preparation. Figure 8 and Figure 9 show the test results for the ROMP and MMU.

Chip Defect Isolation

Chip defect isolation involves quickly pin-pointing which of the 50,000 plus devices on a chip are defective.

Failure diagnosis started out as a laborious manual process of listing the most likely failing signals or circuits whenever a failure

ROMP Test Pattern Generation	
Number of scanstrings	5
Longest scanstring	157 latches
Number of CPU hours to generate	76 hours (3081)
Test coverage on random logic	98.6 %
Test coverage on OCDs	100 %
Test coverage on RAM	100 %
Test coverage on ROM	100 %

Figure 8 ROMP Test Pattern Generation

MMU Test Pattern Generation	
Number of scanstrings	6
Longest scanstring	169 latches
Number of CPU hours to generate	15 hours (3081)
Test coverage on random logic	99.4 %
Test coverage on OCDs	100 %
Test coverage on RAMs	100 %

Figure 9 Design Verification

was detected. The process was improved by using the AUSSIM simulator to resimulate the failing test patterns to reduce the list of possible defects. Later the process was improved further with a program that analyzes all possible defects that could cause a failure, and filters all but the most likely causes. This program proved very successful in accurately pin-pointing defects.

Bench-test setups for probing directly on signal lines which were only microns wide helped find defective circuits.

State-of-the-art tools such as lasers for cutting shorted nets and voltage contrast tools for observing chip behavior were used to pin-point and repair defects isolated with the diagnostic program.

Chips which passed all manufacturing tests in Burlington were next tested in the Austin maufacturing card test by having the processor card execute a comprehensive set of test software. Discovery of defective parts on the processor card is infrequent, demonstrating the effectiveness of our chip test methods.

Conclusions

The organization of the ROMP and MMU chips, together with the design tools which were used, have resulted in a high-performance and flexible processor complex for the RT PC family. The 32-bit RISC CPU (ROMP) is designed to execute an instruction every cycle and the memory system is capable of supporting this high execution rate. The high memory bandwidth requirement of the ROMP is satisfied by the combination of a packet-switched ROMP-MMU interface (RSC), interleaved memory accesses, and a pipelined MMU design for address translation and ECC. The MMU memory management chip contains performance enhancing features such as automatic hardware TLB miss handling, hidden refresh, and ECC error handling.

The design and test tools evolved during our design have been used to design other chips with greater device counts and the diagnostic tool has grown in importance and usage for resolution of chip defects.

References

1. Raymond A. Dupont, Ed Seewan, Peter McCormick, Charles K. Erdelyi, Mukesh P. Patel, P.T. Patel, "ROMP/MMU Circuit Technology and Chip Design," *IBM RT Personal Computer Technology,* p. 66.
2. D.E. Waldecker, K.G. Wilcox, J.R. Barr, W.T. Glover, C.G. Wright, H. Hoffman, "Processor Card," *IBM RT Personal Computer Technology,* p. 12.
3. Ronald E. Rowland, "System Memory Cards," *IBM RT Personal Computer Technology,* p. 18.
4. P.D. Hester, Richard O. Simpson, Albert Chang, "The RT PC ROMP and Memory Management Unit Architecture," *IBM RT Personal Computer Technology,* p. 48.
5. J.C. O'Quin, J.T. O'Quin, Mark D. Rogers, T.A. Smith, "Design of the IBM RT PC Virtual Memory Manager," *IBM RT Personal Computer Technology,* p. 126.

ROMP/MMU Circuit Technology and Chip Design

Raymond A. DuPont, Ed Seewann, Peter McCormick, Charles K. Erdelyi, Mukesh P. Patel, P.T. Patel

Introduction

A VLSI design of a 32-bit 801-based RISC microprocessor [1,2,3] was a significant challenge in the late seventies. The semiconductor processes that were then fully developed did not provide the necessary density or performance capability for a one-chip implementation of the entire processor function. The 2-micron silicon gate NMOS technology under development in Burlington [4] appeared to offer the higher packing density and the lower power circuits required in the low-cost system environment. The circuit challenge was to use this new technology to design a 200-nanosecond, single-chip, 32-bit microprocessor at low enough power to satisfy cooling and reliability constraints.

Many additional challenges faced the design team throughout the chip development. The chip architecture, device process, and all the design tools used to develop the chip were new. Even with the early device models that were provided to the circuit designers, it was predicted that the chip powers would exceed those necessary for the desired cooling and reliability. For this reason, an investigation into the circuit power/performance was undertaken to find a solution to the on-chip power. This investigation led to the exploration of power supply voltages lower than the customary 5 volts, the analysis of circuit design parameters associated with lower voltage, and generation of a lower voltage from the standard 5-volt supply in the system.

In addition to the circuit challenge, the design methodology was developed as the chip was designed. The available design techniques of master slice (gate arrays) and master image (standard cell) were widely used but neither would provide the necessary density or performance. A fully custom design would have been too costly for a non-commercial chip with volumes less than the normal break-even point. The best design technique was to use custom circuits in high-leverage data flow areas and master image in control (random logic) areas.

Circuit Technology

Base Circuit Technology

The base circuit technology consists of standard NMOS FET pull-down devices with a depletion mode pull-up device as shown in Figure 1.

When this circuit must drive a large load capacitance, a buffer stage is added. In a buffered circuit the pull-down devices are enhancement type, however the pull-up devices can be either enhancement or depletion type. Figure 2a/2b shows the two types of buffered ("push-pull") circuits. Because of its lower power dissipation, the enhancement push-pull circuit was chosen for general use, even though the up-level it

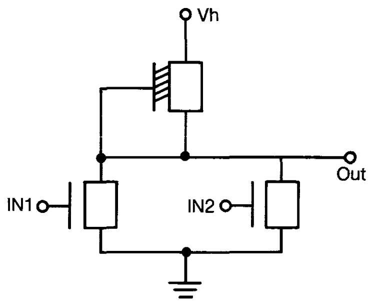


Figure 1 Depletion Load Circuit

provides is a threshold below the supply voltage. The depletion push-pull circuit is used only in areas where a higher up-level is required. As a general design practice, push-pull circuits were used primarily to drive global wiring.

A circuit analysis indicated that the minimum power/delay product of the circuit shown in Figure 1 occurred at a power supply voltage (Vh) of 3 volts. Since 3.4 volts is used by some technologies, the compromise was made to use this value. While the lower voltage provides a significant leverage in many circuits, there are some that perform more optimally at 5 volts. Typically, these were circuits with low capacitance loading in which gate capacitances comprised most of the net loading. These circuits were kept at 5 volts. Figure 3 shows how the 5-volt, 3.4-volt and push-pull circuits might be optimally

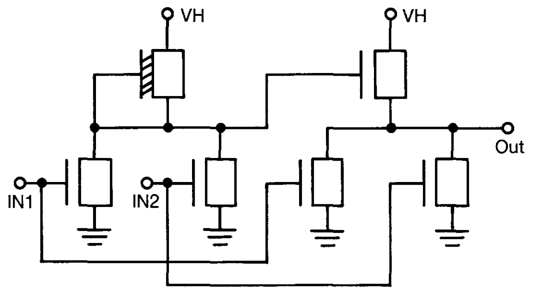


Figure 2a Enhancement Push-pull

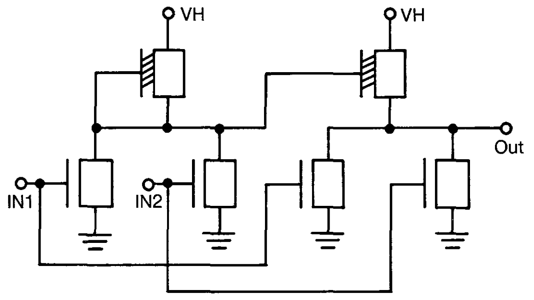


Figure 2b Depletion Push-pull

applied as a function of the wiring capacitance.

In order to provide an adequate noise margin for the transfer of 3-volt signals, the size of the active input devices was double that which would have been used in a 5-volt circuit. This did not have a significant impact on circuit density since the circuits were used primarily in wiring limited areas.

The 3.4-volt supply was not readily available in the system and it had to be generated from the available 5-volt supply. This was done using an on-chip regulator to drive the base of an external PNP power Darlington. The voltage differential is dropped across the PNP

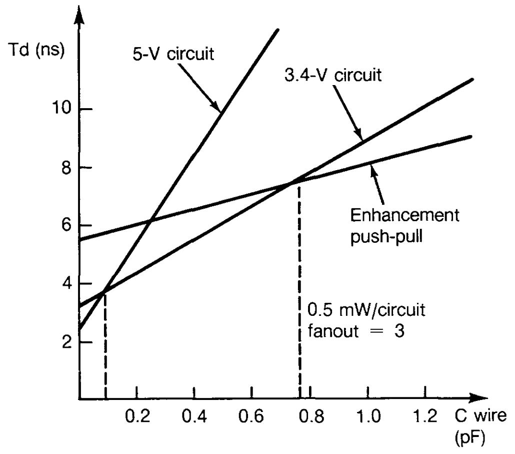


Figure 3 Circuit delay (Td) as a function of wiring capacitance

transistor and regulated by the on-chip regulator.

Off-Chip Drivers
Off-chip driving presented several challenging problems. The delay of many of the off-chip drivers was in the critical paths, requiring delay to be kept at a minimum. Since the capacitance loads are quite large (in the range of 25 to 75 pF), faster switching requires substantial currents. When off-chip drivers switch simultaneously, the peak currents algebraically add and cause substantial shift in the ground or power supply nets, especially in 32-bit designs. This shift can disturb the internal circuits. In order to prevent this, only 12 drivers are connected to a pair of power and ground pads. In addition, the 3.4-volt power supply is used to power the final driver stage. This gives an improvement in the performance as well as a reduction in the switching currents.

Custom Circuit Design
The combination of a two-level metal silicon gate enhancement / depletion NMOS technology and the 32-bit RISC architecture made custom design an attractive alternative for the repetitive logic elements in the data flow. This was particularly evident for the on-chip storage elements which included a 24-word x 32-bit four-port, simultaneous read-write register file implementation of the microprocessor general-purpose registers, a RAM implementation of the two-way associative Table Look-aside Buffer on the address translation chip, as well as numerous 18- and 36-bit multiport data-selector registers used throughout the design of both chips.

Whereas the RAM cell was implemented with a standard six-device symmetrical cell, the register file cell was implemented using an eight-device asymmetrical cell as shown in Figure 4. In this cell, the read devices 5 and 6 are made small relative to device 4 to prevent inadvertent destructive read out. The write devices 7 and 8 are made large compared to the current sourcing capabilities of devices 1 and 2 to guarantee writing.

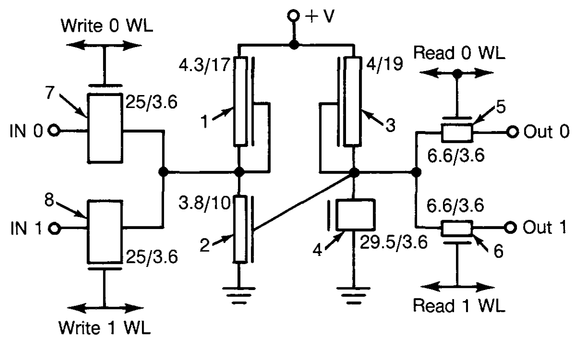


Figure 4 Eight-Device Register File Cell

Transfer devices were also employed to provide area and performance improvements in the design of the data registers. A circuit schematic of a typical latch stage of a data register is shown in Figure 5. The slave latch of this master-slave latch pair, as shown, is composed of nine individual devices. Devices 1 and 2 form an output buffer stage to drive large capacitance loads. Transfer device F, when connected to the buffer output, performs a polarity hold function and device G, when connected to the output of the previous stage, performs an LSSD (Level Sensitive Scan Design, [4]) shift function. The input transfer devices, together with the input capacitance of device 9, comprise a master latch in which the capacitance of terminal 312 acts as a storage element. The additional input devices form a data-selector or multiplexer function allowing different word inputs to be selected.

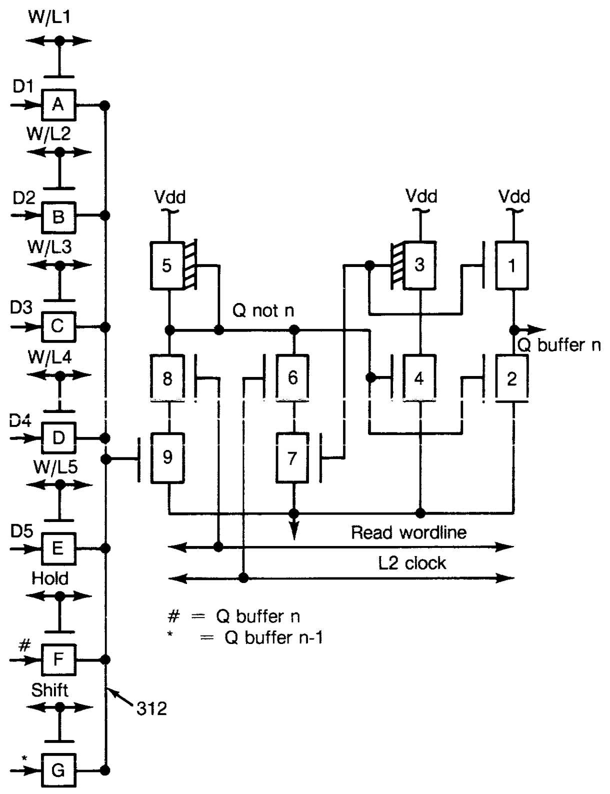


Figure 5 Data Register Bit Cell

Custom design of these respective data storage elements using transfer devices to their natural advantage significantly reduced chip area, improved performance and lowered chip power. In addition, the use of standard predesigned functional blocks, such as registers, facilitated and standardized the logic for these components, reduced data entry and simulation time and finally formed the nucleus of the "bit stack" data flow physical design image for both chips.

In addition to these custom circuits, other specialized circuits were used where they provided an advantage. These circuits include ROM, multiplexers, parity and ECC, ALU, incrementers and comparators. These circuits were manually designed as required.

Custom Physical Chip Design
During the physical chip design, five levels of hierarchy were defined. These consisted of the chip, super macro, macro, circuit and transistor. The use of hierarchical design methodology allows the design to proceed in two areas in parallel, these being the macro and global areas. This reduces the chip design turnaround time and permits the design of reusable macros. This hierarchical top-down approach proceeds in three steps. The first step is the chip floor planning, the second is the macro design, and the third the global design and verification.

Floor planning is done in conjunction with the high-level logic definition and determines the optimum size, aspect ratio and placement of each macro and input/output on the chip image with appropriate space left for global wiring. An abbreviated version of the chip floor plan for the CPU chip is shown in Figure 6. The floor plan is broken into three areas: 1) data flow stacks, 2) random (control) logic form master image (standard cell) and 3) mixed random logic and data flow.

After completion of the chip floor plan the macro design can begin. This is done in two steps. The custom bit-stack macros are done first when the logic data flow is defined, and the master image macros are completed as the control logic is frozen.

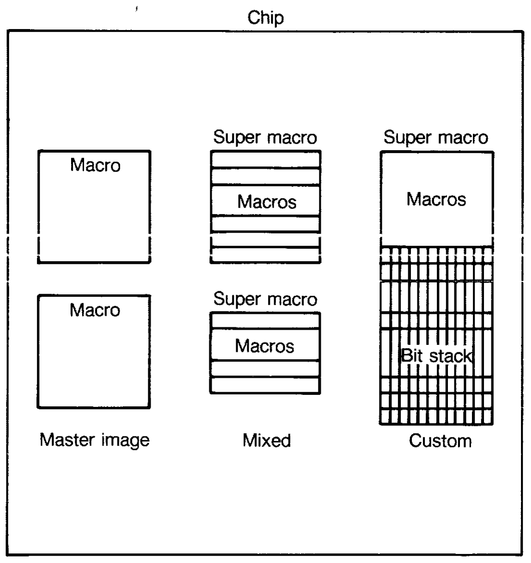


Figure 6 Abbreviated Chip Floor Plan

Data Flow Bit Stacks
A data flow stack is a predefined bus structure to coordinate macro-to-macro communications along predefined wiring channels. The primary characteristic of the macros placed in the bit stack is their bit-slice nature. The macros are designed in a 1 X N configuration, 1 bit in height and N bits wide. Typical macros in the data flow stack are data registers, multiplexers, register file, ALU, shifter, rotators, parity functions and off-chip data bus structures. The macros are then placed in the vertical stack, aligning the bits from macro to macro.

For both the processor and memory management chips the data flow stacks were designed for seven 36-bit data buses across the horizontal cross section as shown in Figure 7. The data bus wiring flows vertically and the control signals flow horizontally. By integrating the data bus with the circuits, both metal layers are fully utilized, reducing chip area, as circuits can be placed under the wiring. The minimum distance between inputs and outputs can be obtained by efficient wire packing of the data buses, reducing wiring capacitances and improving chip performance. Another advantage of bit-stack design is that it allows reuse of general-purpose macros such as multiplexers, parity checkers, data registers, etc. These macros can be predesigned to fit into this structure, thereby preserving the custom advantages while keeping the design time and resource at a minimum.

Master Image Design
The control logic structures are implemented using a master image (standard cell) approach. The circuit library is predefined and consists of 200 combinations of AND, OR, NAND, NOR, OAI, AOI, XOR and latches. Each of these books has five different

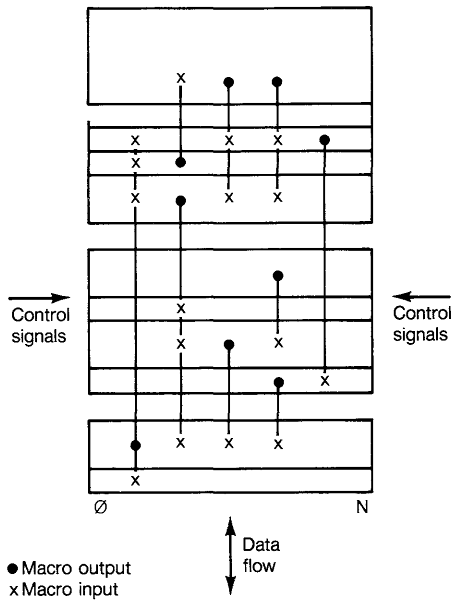


Figure 7 Data Flow Bit Stack

powering levels to obtain a wide range of power performance for each circuit. Appropriate push-pull drivers were also provided to drive the global wires that leave the macro boundary.

The circuits in the control logic macros are automatically placed and wired. This permits the control logic to be designed late in the design cycle, similar to a PLA approach but with the advantage of having control of path performance. The basic layout for a typical master image macro is shown in Figure 8. The circuits are of a fixed height, but have a variable width depending on the function. There are variable wiring bay widths between the circuits for interconnection. Once the circuit bays and wiring bays are defined, placement and wiring programs, based on those described by Donze et al.[5], complete the design of the macro. The power-tuning option permits each circuit to be powered up

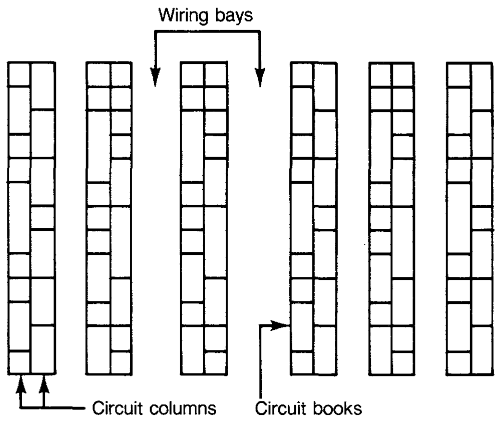


Figure 8 Control Logic Image

or down after placement and wiring are complete. The chip critical paths can then be power-tuned late in the design cycle to improve overall chip performance and the over-powered nets can be adjusted to reduce on-chip power dissipation.

Global Design
After chip floor planning is complete as described above, the global chip design is defined at a high level. The actual macro input/output positions are required for the global chip wiring. These are defined for the data flow stacks early in the design process and the control logic macro input/output pins are fixed prior to their design. With this information, both the macro and global chip designs can proceed in parallel. Capacitance values are continually fed back for performance estimation and verification during the global design phase.

Design Verification
In a complex VLSI design, one of the keys to success of the design is the ability to verify the process ground rules and logic-to-physical correspondence. Due to the complexity of the process and the subsequent lengthy

processing times, it is imperative that the design be fully functional on the first pass. This is achieved through a very extensive checking methodology. In order to contain the data volumes, the macro designs are individually checked for both logical-to-physical correspondence and process ground rules. This data is then suppressed and the same checks are repeated at the global level. The macro level logical-to-physical checks are done down to the device level, while the global checks are done only to the macro level. In addition to the above key checks, other verification is done on the chip performance, DC power drops and power busing, as well as manual audits.

Conclusions

Both the process and the circuit technology were selected to achieve the design of a high-performance, 32-bit, 801-based RISC processor and memory management unit at reasonable system cost. The use of the lower voltage circuits reduced on-chip power and improved circuit performance. To provide an economical power supply for the lower level of supply voltage, an on-chip voltage regulator was used. Innovative custom designs significantly improved the overall processor performance and provided a more efficient use of silicon. The hierarchical design method reduced the overall design time by permitting more than one area of chip design to proceed in parallel. The separation of data flow and random logic permitted the custom circuits to be employed for high-leverage repetitive logic and standard cell design for the control logic, where fast design turnaround time is required. In addition, the use of the data flow concepts permitted a highly-bused architecture to be implemented in a small chip area. Many of the design concepts described in this paper are unique to the IBM design environment as opposed to the commercial environment where large numbers of chips will be built and every last bit of silicon must be used. Some trade-offs have been made between silicon area and design turnaround time, where automatic tools are effective.

Finally, the successful combination of all the design techniques permitted a complex and highly bused architecture to be implemented, providing the major logic components to the system in only two chips.

Photomicrographs of the two chips are shown in Figures 9a and 9b.

Acknowledgments

The circuit and physical design of the processor and memory management chips reflects the contribution of many designers in both Austin and Burlington. Their efforts are gratefully acknowledged.

References

1. George Radin, "The 801 Minicomputer", *IBM Journal of Research and Development,* 27, 237-246, May 1983.
2. George Radin, "The 801 Minicomputer", ACM, 0-89791-066-4 82/03/039.
3. D.E. Waldecker, C.G. Wright, M.S. Schmookler, T.G. Whiteside, R.D. Groves, C.P. Freeman, A. Torres, "ROMP/MMU Implementation," *IBM RT Personal Computer Technology,* p. 57.
4. E.B. Eichelberger and T.W. Williams, "A Logic Design Structure for LSI Testing", *Proc. 14th Design Automation Conference,* June 1977, 77ch1216-1c, pp. 462-468.
5. R. Donze, J. Sanders, M. Jenkins, G. Sporzynski, "PHILO - A VLSI Design System", ACM IEEE 19th Design Automation Conference, June 1982.
6. R. Bechade, M. Concannon, C. Erdelyi, W. Hoffman, "A Comparison of Mixed Gate Array and Custom IC Design Methods", *1984 ISSCC Digest of Technical Papers,* February 1984.
7. P. McCormick, M. Lang, "Hierarchical Design Methodologies, A VLSI Necessity", *Advances in CAD for VLSI,* vol. 6, Design Methodologies, 1985 (in press).
8. IMB Corporation installed user program document SH20-1118-0, "Advanced Statistical Analysis Program (ASTAP), Program Reference Manual", Program Number: 5796-PBH.

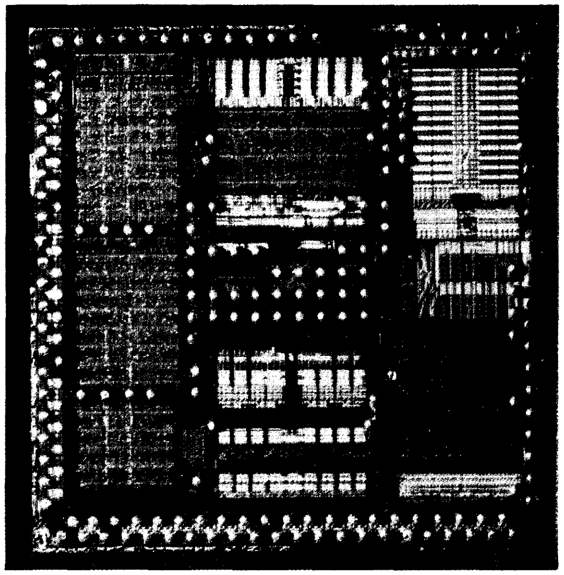


Figure 9a Processor Chip

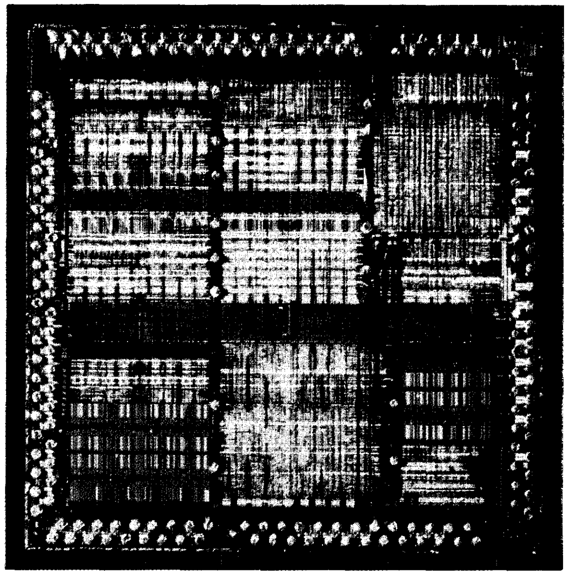


Figure 9b Memory Management Chip

Software Development Tools for ROMP

Alan MacKay and Ahmed Chibib

Introduction

Traditionally, a microprocessor is designed with minimal interaction between the hardware and software engineers. The resulting architecture often contains many complex instructions and addressing modes well suited for the assembly programmer. However, such instructions are seldom utilized by compilers due to their complexity.

This paper describes a set of tools used to assist the hardware engineers in designing, refining and verifying the architecture of a Reduced Instruction Set Computer (RISC) [1] to be a suitable target for optimizing compilers. This IBM System/370-based tools package consists mainly of an optimizing compiler, a binder, a fast simulator, and a host of execs, utilities and libraries. We should point out that these are internal IBM tools and are not part of the software available for the IBM RT PC.

Compiler Tailoring

The ROMP processor architecture [2] and compiler design was a cooperative effort performed by software and hardware engineers in Austin and Yorktown. It was one of few projects where compiler technology influenced many hardware decisions and vice versa. The ROMP instruction set was tailored for speed and space efficiency of the generated code without sacrificing the function of the processor itself. By the same token, the PL.8 compiler [3,4,5] took advantage of such hardware features [6,7] as the execute form of Branches which perform the next ("subject") instruction in parallel with the branch, or the overlapping of a Load instruction with the execution of another instruction that doesn't require the result of the Load. This was accomplished by having a scheduling mechanism in the compiler that analyzes each basic block of generated code and rearranges the instructions to facilitate such an overlap.

In the spirit of the RISC architecture, certain high-level functions on the ROMP, such as multiply, divide, and storage-to-storage move operations, are implemented with subroutines rather than microcode. On the assumption that only one shared copy of these routines will exist, the subroutines, which are called Run-Time Routines are hand tailored and highly specialized for speed efficiency without regard to space.

Most languages, regardless of their level, make use of subroutines to implement primitives and data abstractions. The PL.8 compiler takes advantage of the ROMP architecture by providing an efficient subroutine linkage and parameter passing mechanism. In passing parameters, the first four are loaded into predetermined registers. The invoked procedure may use those registers without ever having to copy them into storage.

Another aspect of subroutine linkage is the procedure's prologue and epilogue. The PL.8 compiler assumes that certain registers (R0-R5) are killed (scratch registers) by a CALL. The compiler, therefore, uses those registers first to avoid saving and restoring any of the CALLer's registers. However, if more registers are needed, they are used in descending order (R15-R6). This is particularly important because the STM (Store Multiple) instruction always saves from the register specified through register 15.

After the initial ROMP design was completed, a few opcode points remained available. To make the best use of them, a statistical gathering feature was added to the compiler to collect information on the frequency of instructions and their operand's value. After numerous bench marks had been compiled, those opcode points were assigned to the short form (2 bytes) of several instructions.

The short-form instructions included memory "Loads" and "Stores". They have a maximum displacement field of 15 units, where the unit is a byte, half-word, or word depending on the particular instruction. The PL.8 compiler tries to maximize the use of these short-form instructions by arranging the program's data so that the small and most frequently used variables are mapped closest to the base address, without regard to the order in which they were declared in the program.

Because the compiler was designed as a state-of-the-art compiler, particular attention was not paid to the resources required for compilation. Thus, the compiler demands a significant amount of computer resources.

Program Binder

Data integrity and early error detection have always been elusive goals in programming. Hardware engineers provided us with protection keys and supervisor state, while language designers raised the level of the source languages. In the PL.8 compiler, as in some other compilers, run-time checking was introduced to protect the programmer from obvious errors like indexing out of an array bound, or beyond the end of a string. The ROMP instruction set has trap instructions that allow such run-time checking to be implemented at a very low cost in both space and execution (approximately 10%). However, a compile time option is provided to eliminate checking code generation.

In addition to run-time checking code, the PL.8 language requires the declaration of all external procedures along with the description of their arguments. This insures that the data types of the actual parameters specified on a procedure call actually match those parameters specified on the procedure definition. Moreover, the PL.8 compiler generates symbol cards in the text deck for both the entry declaration and the subroutine definition. This allows the binder [8] to perform type checking at bind time for separate compilations.

Design and Use of Simulator

As the ROMP version of the compiler was being developed, it was necessary to ensure that the code being generated was correct. This was at a time when the hardware design was not yet complete, and it would be at least a year before a prototype of the ROMP processor would be available. It would be even longer before models would be available in sufficient quantities to allow general availability to programmers. In order to allow early testing of the compiler, a simulator for the ROMP was implemented to run on 370.

The simulator allowed verification that the code generated by the compiler was correct and complete. This gave the software and hardware engineers more confidence that the architecture changes to the ROMP were needed and could be used by the compiler. The knowledge that the code generated by the compiler was correct allowed errors in the first prototype hardware to be identified and corrected without going through the tedious determination of hardware or software fault.

A primary factor in the usability of the simulator was its fast execution. This is accomplished by what we term "compulation." Compulation is a simulation technique of translating the simulated instruction into a sequence of native instructions. These "compiled" sequences of native instructions are kept for subsequent re-execution of that simulated instruction. Each 256 bytes of the simulated ROMP memory is mapped into 4K-byte areas of 370 memory. These 4K-byte areas are divided into "cells" of 32 bytes, each cell corresponding to a half-word of ROMP memory. These cells hold the 370 instructions to simulate the corresponding ROMP instruction. The 4K-byte areas are reusable; more available 4K-byte areas results in more ROMP code maintained in its "compiled" form. Each cell is initialized as a branch to a routine to "compile" a ROMP instruction. All simulation execution is a branch from one cell to another. If the cell contains a compiled instruction, that ROMP instruction is simulated. Otherwise, the corresponding ROMP instruction is compiled into the 370 instructions which are then executed. For example, the ROMP instruction:

```
cas r5,r6,r9
```

will expand into the following 370 instructions:

```
l   r1,disp_r6(b)
al  r1,disp_r9(b)
st  r1,disp_r5(b)
bct time,continue
bal ric,end
```

The Load instruction gets the contents of the simulated ROMP register 6 into a 370 register. The Add Logical gets the contents of ROMP register 9 and adds it to what is in register 6. The Store then sets ROMP register 5 to its new value of register 6 plus register 9. The Branch on Count instruction decrements the number of ROMP instructions left to be executed before halting execution, and then branches to the next simulation instructions. If the number of instructions goes to zero, the Branch and Link instruction branches to the code that re-materializes the ROMP condition codes, program counter, and other system information before returning to the user interface.

This "compulation" requires additional overhead when an instruction is executed for the first time, but most work done by computers is done by looping, so that a significant amount of time is spent re-executing the same instruction, without any further decoding of the ROMP instruction.

Another factor in the simulator that allows very fast execution is that the condition code bits for the ROMP are not calculated until needed, e.g., until a bit is tested by a conditional branch or the simulation is stopped and control is returned to the command processor for interaction with the user. As part of the simulation of instructions that modify a condition status bit, the operation and operands are saved. Then, when a bit is tested by a conditional branch, just that bit is reconstructed for use by the branch. This re-materialization of the needed condition code bits provides a performance benefit because the 370 condition codes do not have to be remapped into the ROMP condition codes which take many 370 instructions. It can also be observed that the condition codes set as a by-product of instruction execution are seldom used, and then usually only a single one of the conditions will be tested.

These simulation techniques allow execution of large amounts of ROMP code at very low cost on the 370. The first time cost of "compiling" the ROMP instruction is on the order of 100 370 instructions. But once the ROMP instruction has been "compiled", the execution time is only 5 to 10 370 instructions executed per ROMP instruction simulated.

Code Development Support

The availability of good software tools is often as important as the processor itself. The ROMP was no exception. The PL.8 compiler which compiles three different languages (PL.8, Pascal, and C) and produces code for four different machines (370, ROMP, M68000, and 801) (see Figure 1), along with a fast simulator, binder, and a host of other tools, provided an excellent code development environment that was to show the advantages of the ROMP over other microprocessors.

The presence of a compiler from the beginning of the processor design allowed most of the code written for the ROMP processor to be in a high-level language. This was especially valuable when early processor samples were received. One sample of the processor had a defective register 2, which is the function return value register and first parameter register. Within a couple of days, the compiler was changed and all applications recompiled to not use register 2. If a significant amount of assembler code existed, such flexibility would not have been possible.

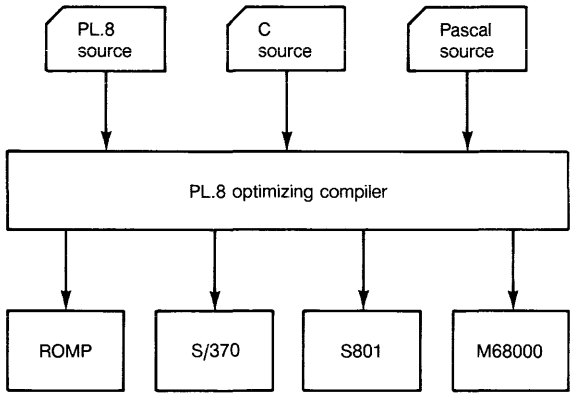


Figure 1 Three Source Languages, Four Target Machines

Using the 370, backend programs could be written in any of the three languages supported by the PL.8 compiler. They were then compiled, modified, and tested natively on 370, taking advantage of all the VM tools, long before the ROMP processor was available (see Figure 2). Once a program was tested, it could then be simulated on the ROMP fast simulator, or executed on the ROMP model itself when the model became available.

Tools are also important in measuring the performance of a microprocessor. When only primitive tools are available, bench marks are coded in assembly language, and bytes and cycles are counted manually to produce summarized results that are often questionable at best, due to the difficulty in verifying that a given program does in fact perform the specified function. However, much easier procedures existed for ROMP. Since we were confident that the PL.8-generated ROMP code was comparable to the best handcode, all coding was done in a HLL. Compilation and testing was all done on 370. The compiler listings provided us with code and data size information, while the simulator gave us the cycle count. The comparison results, though almost always positive, did at times highlight some

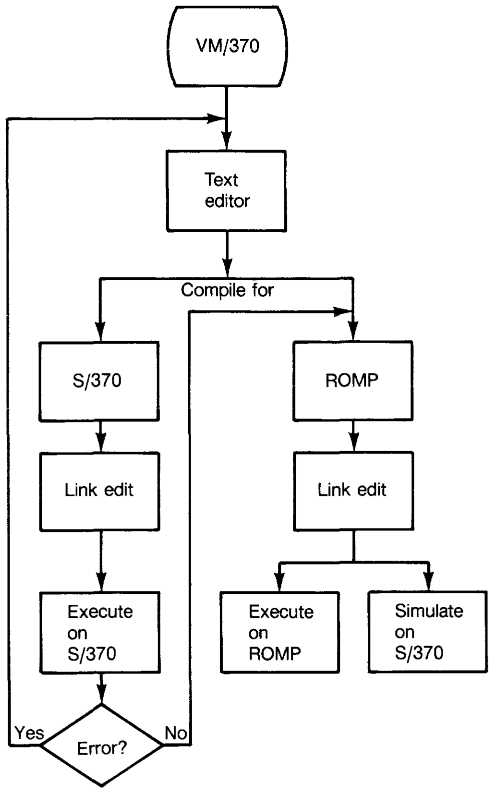


Figure 2 Code Development Process

inefficiencies in the hardware architecture, and/or the compiler-generated code, which were then corrected.

Conclusions

The unique design and early availability of the ROMP simulator allowed development and extensive testing of tools such as the PL.8 compiler and application prototype code before any hardware was available. The hardware and software engineers designing the ROMP processor together produced a full-function, general processor without unnecessary complex instructions that would be unused by software and expensive to implement in hardware. The presence of a compiler from the beginning of the processor design allowed most of the code for the ROMP processor to be written in a high level language. The VRM [9] was mostly written in PL.8. When some assembler code was required for performance, the PL.8 compiler-generated code was used as the starting program for hand tuning.

References

1. D.A. Patterson, "RISC-1: A Reduced Instruction Set VLSI Computer," *Proceedings of the Eighth Annual Symposium on Computer Architecture,* May 1981.
2. D.E. Waldecker and P.Y. Woon, "ROMP/MMU Technology Introduction," *IBM RT Personal Computer Technology,* p. 44.
3. M. Auslander and M.E. Hopkins, "An Overview of the PL.8 Compiler," Proceedings of the SIGPLAN '82 Symposium on Compiler Writing, Boston, MA. June 23-25, 1982.
4. M.E. Hopkins, "A PL.8 Overview," Paper to be published.
5. M.E. Hopkins, "Compiling for the RT PC ROMP," *IBM RT Personal Computer Technology,* p. 76.
6. P.D. Hester, Richard O. Simpson, Albert Chang. "The RT PC ROMP and Memory Management Unit Architecture," *IBM RT Personal Computer Technology,* p. 48.
7. George Radin. "The 801 Minicomputer." *Proceedings of Symposium on Architectural Support for Programming Languages and Operating Systems,* Palo Alto, California, March 1-3, 1982
8. G.J. Chaitin and C.C. Hoagland, "A Compiler Output Format and Its Binder and Loader," Paper to be published.
9. Thomas G. Lang, Mark S. Greenberg, and Charles H. Sauer, "The Virtual Resource Manager," *IBM RT Personal Computer Technology,* p. 119.

Compiling for the RT PC ROMP

M.E. Hopkins

Introduction

The IBM RT PC ROMP architecture is relatively low level and simple. A natural consequence is that the primitive instructions should execute rapidly on most implementations. Does the choice of such a low level interface make sense given that almost all programming today is, or should be, done in a high level language? Could compiler writers do a better job if the CPU was somewhat more elaborate, with additional functions tailored to the constructs commonly found in high-level languages? Of course it is clear that code can be generated for any execution model. Examples of execution models are register transfer, stack, and storage-to-storage. Unlike human coders, compilers will tirelessly and accurately generate long sequences of code to map one model of a language onto a machine with another model. The hard task is to obtain efficient code for a particular machine.

Which style of machine is best? Our preference for a machine like the ROMP is based partly on fundamental engineering constraints and partly on our ability to use well understood compilation techniques to obtain high quality code. An example of a fundamental engineering constraint is that operations that are internal to the CPU, such as register-to-register add, run faster than instructions that reference storage, even on machines with caches. (The fact that some machines slow down basic arithmetic to memory reference speed should not concern us.) Examples of compilation techniques will be given later. We also have a certain bias to simple hardware solutions. Part of this is aesthetic, but we also have a suspicion grounded on experience that the next language just may not match the complex operation which is built into an elaborate architecture.

The discussions that follow are based on the PL.8 compiler, which accepts source programs written in C, Pascal and PL.8, a systems dialect of PL/I. A description of the compiler is given in Auslander and Hopkins[1]. PL.8 produces optimized object code for System/370 and MC68000 as well as ROMP and the 801 minicomputer [2]. The compiler largely relies on global optimization and register allocation to produce good object code. The VRM and various ROMP tools were developed using PL.8. Originally, the compiler was an experimental vehicle used to build software for the 801 minicomputer, but in recent years it has been used in a number of internal IBM projects. It is not presently available to customers.

Hardware/Software Cooperation

Both hardware and software affect system performance. The compiler writer must accept his share of the responsibility. The ROMP architecture divides the task in ways that lead to better performance without excessive burden on either hardware or software. A few examples will indicate how responsibility is shared.

One of the more expensive operations on many computers is branching. As long as instruction execution proceeds sequentially it is possible to prefetch and decode instructions ahead of their actual execution. This overlapping is usually termed pipelining. When a branch is encountered a new instruction stream must be found. Conditionality and computed branch targets complicate the decisions that must be made in hardware. Very-high-performance machines do prefetch on multiple paths and retain branch history tables to avoid "flushing the pipe." Most one-chip processors simply accept expensive branches as a fact of life. The ROMP solution is to define a family of execute branches that perform the next ("subject") instruction in parallel with the branch. Implementing this facility only complicates the hardware a little. It thus becomes the responsibility of the compiler to produce execute branches. Through most of compilation, the compiler only deals with branches in the familiar non-execute format. At one point a scheduling process is run which rearranges code between labels and branches. (This unit is termed a basic block.) One of the goals of scheduling is to place an instruction that could become the subject of an execute branch just in front of the branch. (The main constraint is that the branch cannot depend on the result of the subject instruction.) Other optimizations are unaware of the compilation of execute branches. Final assembly then looks at the instruction that precedes every branch and flips the pair if it is valid to convert a normal to an execute branch. Branches tend to constitute over 20% of all instructions executed. Even if only half of all branches can be transformed to the execute form, a modest increase in hardware and compiler complexity has resulted in the effect of a 10% reduction in the path length or number of instructions executed.

A similar situation exists with loads. Loads tend to take substantially more time than register-to-register (RR) ops, but it is possible for the hardware (in Real mode) to overlap the load with execution of the following instruction if the next instruction does not require the result of the load. The scheduling process also rearranges code to facilitate such overlap. If loads constitute 15% or 20% of all executions, it is easy to see that another 10% or greater reduction in effective path length may be achieved here. Notice that a machine that bundles the fetch of an operand from memory with a computation cannot easily overlap fetching with some other function.

Of course the object code that comes from such a compiler looks strange. In some sense, you are seeing the equivalent of the internal state of a very costly high-performance pipelined processor. Writing optimal assembly language code requires some care on the ROMP. It is rather like microcoding. However, on the ROMP the process is systematic, if tedious, making it fortunate that most programming is done in a high-level language.

Compilation Strategies
The most important optimizations performed by the PL.8 compiler are probably moving code out of loops, the elimination of redundant computations (commoning), and register allocation. The ROMP makes these operations easier and more profitable.

Let us examine these optimizations in the light of machine models and how they evaluate expressions:

- Stack computation
- Memory-to-memory
- Memory-to-register
- Register-to-register

Consider the source code fragment:

```
x = a + b;
(a few statements, which destroy x, leaving
a and b)
y = a + b;
```

If the recomputation of a + b is to be avoided on the stack machine, an explicit copy in storage must be made and then the value must be refetched from storage when assigning to y. The trouble with this strategy is that "remembering" is very costly. On the ROMP an RR Add costs one cycle, while Loads and Stores take between three and five cycles depending on whether the machine is in real or virtual mode and whether or not it is possible to overlap another instruction with the load. Unless an operation is very expensive, it is often as efficient to recompute as to "remember" on a stack machine. On a memory-to-memory machine one must often pay for an explicit copy as in the following code for a hypothetical memory-to-memory machine:

```
temp = a + b
x = temp
  :
y = temp
```

The added storage references may well make commoning counterproductive. We shall say no more about the stack or memory model. Whatever their virtues for simplifying compilation, they seem to guarantee more storage references and thus worse performance than the other two models.

The storage-to-memory model is shared by 370 and MC68000. At first glance a 370-type approach seems attractive.

```
x = a + b; L R1, a
           A R1,b
           ST R1, x

y = a + b; ST R1, y
```

On the ROMP we get:

```
x = a + b; L R1, a
           L R2, b
           CAS R3, R1, R2 Add
           ST R3, x

y = a + b; ST R3,y
```

The ROMP takes one more instruction. (It does have some opportunities to obtain overlap on the Loads by inserting unrelated instructions, but let us ignore that benefit). If the example is changed slightly to:

```
x = a + b;
y = a − b;
```

We then get on 370:

```
x = a + b; L R1, a
           A R1, b
           ST R1, b

y = a − b; L R1, a
           S R1, b
           ST R1, y
```

After the first statement, neither a nor b are available and they are the operands of the next statement. On the ROMP, both are available and so there is no need for an expensive refetch. Of course we can turn the 370 into a ROMP-style, register-to-register machine. The problem is that the 370 Add instruction destroys one of its operands, while CAS, an Add that doesn't set the condition code on the ROMP, is three-address. The PL.8 compiler goes to considerable effort to give 370 code the benefits of both the storage-to-register and register-to-register approaches. It is not clear that the effort is worth it. On some 370 models, two Loads and an Add Register may be as fast as Load, Add from Storage. In any case there are relatively few storage-to-register computational operations in a typical snapshot of 370 execution. One typical mix shows the following most frequently-executed storage-to-register ops.

Instruction	% of execution
C: compare	1.74
N:and	1.26
AL:add logical	1.07
CL:compare logical	.44
A:add	.39
S:subtract	.37
O:or	.36
CH:compare half	.33
AH:add half	.24
SH:subtract half	.10
MH:multiply half	.07

If all such ops are included, the percentage of executions is less than 6.5%. Modest as this figure is, it overstates the advantage to be gained from memory-to-register ops, as many of these instructions are addressing literals. On the ROMP they would be immediate ops. In light of frequency of usage, potential performance improvement, hardware requirements and compiler complexity it is hard to believe that storage-to-register ops are cost effective.

The reader may not be impressed with optimizing a + b, and would be correct if the only benefit of optimization was a rewrite of the user's program at the source level. The most potent effects of an optimizing compiler are derived from reducing the administrative code used to implement high-level constructs. Consider what it takes to implement the following code fragment in PL.8.

```
1 a                 static ext,
2 b                 integer,
2 c (0:10),
    3 d             integer,
    3 e             integer,
    3 f             char (16);
  x = e(i);
```

The reference to e(i) includes the following factors:

- The address of the structure a
- The displacement of e within a
- i times the stride of c.

In PL.8 and Pascal, subscript range testing is normally done even on production code. Thus there is also a trap to ensure that the value of i is between zero and ten. The fetch of e(i) may be commoned or moved out of a loop, but there are many other opportunities for optimization. The load of the address constant to locate the structure need not be repeated when a reference is made to b. Storing into d(i) requires no additional instructions. Programs are filled with opportunities to reuse portions of this administrative type code. The higher the level of the machine, the less chance there will be for reuse, as one factor may change.

An example of this phenomenon is in subscript computations. As the ROMP does not have a built-in multiply instruction, the compiler generates a series of shifts, adds and subtracts when the stride is a constant. Thus a multiply by 24 is implemented as:

```
shiftl(i, 4) + shiftl(i, 3)
```

If somewhere else in the program there is a multiply of i by 8 or 16, one of the shifts already used to compute i*24 will suffice. By systematically exploiting the many small opportunities for optimization that occur in real programs, the PL.8 compiler can produce programs that execute very rapidly on the ROMP.

It is now necessary to discuss register allocation. So far we have tacitly assumed that there would be enough registers to hold all the intermediate results which optimization creates. A large number of registers require, not only more hardware, but more bits in the instruction to name the particular register. Compiler studies showed that, while 32 registers were beneficial, 16 were a reasonable compromise. A PL.8-type compiler approach would probably not be very effective with substantially fewer than 16 registers. The code would tend to look like the memory-to-memory model of computation. The PL.8 compiler uses a graph coloring algorithm [3] to assign the infinite number of symbolic registers used during optimization to the 16 available on the ROMP, but other methods can be used.

It is particularly important that a machine not restrict the register allocation by typing registers or otherwise constraining their use. Implicit usage is also undesirable. Even the ROMP has some minor problems here, but they are easily overcome. Register 0 cannot be used as a base because the CPU assumes this means the value zero. The register allocation phase of the compiler overcomes this problem by introducing an interference in the coloring graph. Each symbolic register used as a base interferes with real register zero; thus, the compiler will not assign such a symbolic register to R0. Branch and Link implicitly uses R15. This was chosen by the compiler writers to match the proposed linkage conventions. The most bothersome constraint is paired shifts. Normal shifts on the ROMP are of the form:

```
Shift                RA, shift amount
```

The value to be shifted is in RA and is returned to the same register. If only this form of instruction were available, implementing a multiply by an arbitrary constant using shifts and adds would often require intermediate copies. Rather than introduce a 4-byte nondestructive shift instruction, the paired shift was introduced. Every register has a twin whose name is obtained by complementing the low bit of the name (e.g., the twin of R2 is R3 and vice versa). The PL.8 register allocator handles this in the following manner. The internal form of the program used by optimization has shifts with separate target, source and shift count fields. Prior to register coloring an attempt is made to coalesce the source and target. If this fails, an attempt is made to coalesce the source and target onto a particular pair of real registers. Other cases, which seem to be rare, result in an extra load register.

On machines like the 370 there are a plethora of problems associated with registers:

- The 370 really has fewer than 16 registers because at least one must be reserved for program addressability.
- The fact that integer multiply destroys a pair of registers introduces complications.
- The PL.8 compiler has never really exploited 370 instructions that use register pairs such as the loop closing BXLE op and double length shifts. (We are not alone in not using BXLE. It has a frequency of less than .01% on most execution samples.)
- The fact that some arithmetic and logical instructions work on less than a full word is a constant problem. It takes a lot of special analysis to decide when a short op can be used.

While the ROMP does have some minor irregularities in its register scheme, it is a substantial improvement on our past architectures, resulting in few problems for an optimizing compiler whose goal is to retain many available quantities in registers.

Checking and Linkage

In recent years programming languages have attempted to guard against programming errors and raise the level of the source language. The ROMP instruction set supports both.

The trap instructions provide an economic method to test for unusual or erroneous conditions during execution. Pascal and PL.8 both customarily run in production with checking enabled. However, it is possible to eliminate these checks. By having separate checking ops and then optimizing code, the compiler writer can ensure that the correctness criteria of a wide variety of languages are efficiently enforced.

The efficient implementation of a language like C, in which primitives are coded as basic functions, clearly depends on linkage. However, higher level languages which implement data abstractions also depend on the subroutine mechanisms. In implementing procedure call, the ROMP convention is to load the first four parameters into registers. The invoked procedure may then use them in place or copy them into other registers. The important point is that they seldom need to be copied into storage, an expensive operation. Longer parameter lists must be put in storage, but these are relatively infrequent. This strategy is much more efficient than the traditional 370 or UNIX type linkage, which passes parameters in storage. When invoking a procedure, it is not normally necessary to load its address. System routines such as multiply or the primitive storage allocator are kept in low memory and the 24-bit absolute branch can be used to access them. Relative branching within a bound module, which is as large as a megabyte, is also possible with a single instruction. On entry to a procedure it is merely necessary to do a Store Multiple to save any registers that will be used and bump the stack pointer. Stack overflow is normally caught by an attempt to reference a protected page. (For procedures with large stack frames an explicit check is made.) Exit from a procedure consists of loading the return value in a register, restoring the saved registers and executing a branch register.

In practice, there are many variations on this theme, depending on the language, system conventions, and the user's program. For example, in Figure 1 we see the object code for a function that performs the typical C storage-to-storage move. Because it is able to work entirely out of registers that are, by convention, not saved over a call, it has no prologue and an epilogue that consists of a branch register. As source programs and languages become more complex, procedure call overhead will increase, but the compiler writer can always choose the minimum code sequence for the task at hand. One interesting consequence of the MMU relocate is that, given inverted page tables, many systems will want to allocate a very large contiguous stack when a process is created. There is no reason to maintain the stack in disjoint sections, as the mere existence of address space does not degrade performance as is the case with conventional page tables.

High-Level Functions
High-level functions on the ROMP are implemented with subroutines rather than microcode. The most obvious examples are

```
/* move to a byte of zeros. */
move (t, s)
char *t, *s;

while (*t+ +  =  *s+ +);
return;
```

Object Code for ROMP

```
2| 000000                                PDEF   move
5| 000000                          %6:
5| 000000 LCS    4003                   LCS    r0,$MEMORY+*s(r3)
5| 000002 INC    9131                   INC    r3,r3,1
5| 000004 INC    9121                   INC    r2,r2,1
5| 000006 CIS    9400                   CIS    cr,r0,0
5| 000008 BNBX   89AF FFFC              BFX    cr,b26/eq,%6
5| 00000C STC    DE02 FFFF              STC    r0,$MEMORY+*t-1(r2)
7| 000010 BNBR   E88F                   BFR    24,r15
```

Figure 1 Example

multiply, divide and storage-to-storage move. On 370 instruction traces, move constitutes about 2% of all executions, making it, by far, the most important complex instruction. It tends to consume close to 10% of the execution time. On large 370 machines 2 bytes are moved per cycle. (A cycle is taken to mean the time for a minimum op such as a register add.) For aligned moves, the ROMP can achieve close to this rate by means of an "unrolled" loop. For unaligned moves, a carefully handcrafted subroutine has been written. It uses ops which are of otherwise marginal utility, such as MCxx. There are even some compensations for not having the 370 MVC op. The ROMP move subroutine has been tailored to make moves of overlapped data nondestructive, thus satisfying the PL.8 rule. Once again we note that high-level instructions never quite do what they are supposed to do, but low-level ops can be specialized to the specific requirements.

The various versions of multiply on 370 constitute about .1% of all instructions executed. High-performance 370 machines tend to have a very expensive multiplier. Low-end machines implement the multiply instruction with a microcode operator that is the functional equivalent of the ROMP Multiply Step instruction. It is hard to see how the ROMP solution results in significantly worse performance. Sometimes there may be better performance. Constant multiplies can be done with adds and shifts. Some applications may not require a full 32-bit multiply. If the multiplier is only 12 bits long, then it is possible to get a product with six rather than 16 multiply step instructions. Once again the basic instruction set permits the user or compiler writer to do exactly what he wants with great efficiency rather than depending on the foresight of some computer architects. (Those of us who participated in the development of the ROMP architecture are constantly grateful that we did not enshrine our more exotic requirements in silicon.)

One of the sadder sequences of code is to see a divide by two in a binary search or heap sort implemented with a divide rather than a shift instruction. Even on high-performance machines, divide can take almost ten times a shift. That is a big loss of performance in a loop that is likely to be very important. This doesn't occur because compiler writers are unaware that a right shift can sometimes replace a divide by a power of two. The problem is negative numbers as dividends. $(-1)/2$ is 0 on 370. -1 shifted right one bit is still -1. The substitution of a shift for a divide only works for positive dividends. For the PL.8 language we decided to implement a true twos complement divide subroutine using the Euclidean algorithm that rounds down rather than toward 0. Thus replacing divides with shifts gives the same result. In this case a low-level instruction set gave us a new view of source language semantics. We simply implemented the divide subroutine that we wanted rather than accepting built-in semantics.

The ROMP does have Load and Store Multiple ops. It would be possible to get along without them. However, this is one case where a high-level instruction improves performance. This is because they permit the CPU to send one address to the memory subsystem and then do a series of Loads or Stores without the interference of fetching a series of instructions and sending effective addresses to the memory subsystem.

The ROMP approach to implementing high-level function frees the compiler writer and user from the tyranny of instruction sets without giving up any significant performance. Furthermore, the engineer can concentrate on making Load, Store and Branch run well. Here is the frequency of execution of the top ten instructions in a typical snapshot of 370 execution.

Instruction	% of Executions
BC:Branch Condition	20.2
L:Load	15.5
TM:Test Under Mask	6.1
ST:Store	5.9
LR:Load Register	4.7
LA:Load Address	4.0
LTR:Test Register	3.8
BCR:Branch Register	2.9
MVC:Move Characters	2.1
LH:Load Half Word	1.8

Together these constitute 67% of all instructions executed. Clearly the vast majority of the over two hundred 370 instructions occur a good deal less than 1% of the time. Most of the above have direct counterparts in the ROMP instruction set. Other than move, it is hard to think of any 370 instruction which might have improved ROMP performance if it had been implemented.

Code Size and Path Length

The 801 minicomputer was designed to have the shortest possible path length, and code size was sacrificed to achieve this. This is highly appropriate on a machine with a cache. In a machine with a storage hierarchy, most of the faults come from referencing data. Doubling the size of the code only marginally increases the number of faults. However, the ROMP does not have a cache. In order to multiplex the 32-bit memory channel with instructions and data, it helps to have short instructions.

For this reason the ROMP has short forms of many commonly occurring full-function instructions. In addition, a compromise was made such that, of the register-to-register operations, only CAS, the form of add that does not set the condition register, is fully three-address. Shifts have the paired form while subtract and the logical ops destroy one operand. This is a compromise. Add occurs so frequently that a 16-bit, three-address format has a big benefit. There are not enough code points to have all the other RR instructions in 16-bit, nondestructive format. Because the register allocator was able to coalesce operands most of the time, a 16-bit, two-operand format was chosen for the other RR ops. Similar reasoning led us to have short-form increment and decrement instructions.

All in all, the ROMP is surprisingly space efficient without undue performance loss. The average length of a ROMP instruction varies from application to application, but is usually well under 3 bytes. In some cases, the ROMP does require an added instruction but it is relatively infrequent and an easy decision for the compiler.

Details

A number of small details contribute to making the ROMP a good target for compilers.

- Condition codes tend to be an awkward match for many systematic methods of compilation. In the ROMP, those instructions that set the relational bits of the condition register set them in the same way as a compare with zero. This permits the compiler to eliminate all compares with zero that are preceded by an instruction that sets the same register as the register comparand. It is also important to not set the condition register on Load, Store, or the basic Adds that compute addresses. This permits code to be inserted, or rearranged without worrying about the condition register. The condition register test bit provides an efficient means to move and compare arbitrary bits even when their position in a word must be computed at run time. This makes it very efficient to implement packed arrays of bits and Pascal-type sets.

- Load instructions that fetch bytes and half-words from storage either set the remainder of the register to zero or fill it with sign bits. This makes it easier to treat partial words as algebraic or logical quantities. On 370 one of the most common idioms is a subtract of a register from itself followed by an insert character. LC does the entire job on the ROMP.

- The Load and Store Multiple instructions can be used to do block moves or zero large areas in an efficient manner.

- Sometimes constant data will not fit into the ROMP 16-bit immediate field. Instructions are provided that treat the immediate field as a left-justified quantity. It is thus possible to follow either of two strategies. Use two ops if either the upper or lower form is insufficient. The alternative is to manufacture the constant in a register, which requires two instructions, but then the constant may be reused many times by short, fast RR ops.

Other Methodologies

Not all source languages will be implemented with optimizing compilers like PL.8.

Where high-quality object code is not crucial it may pay to have a very fast compiler that produces mediocre object code. A number of features make this a reasonably easy task. Even code from a very naive compiler is quite compact. The large number of general-purpose registers means it is easy to reuse values over short stretches. The large displacement means one can reserve large areas for intermediate results without fearing overflow. High-level function can be invoked via subroutines with a reasonable in-line overhead. Finally, code can be addressed and constants can always be manufactured on-the-fly without establishing or maintaining addressability to code segments and a literal pool as is required on 370.

Another method of implementation is interpretation. The ROMP is a very good interpreter. This should not surprise us as it is really a general-purpose micro engine.

Conclusion

The ROMP architecture provides the high-level language compiler writer with the right set of implementation primitives. Its strength is the ability to combine the basic operations in new ways suited to the task at hand. In Figure 1 we have an example of how a common idiom in C is efficiently implemented. It is hard to see how the most specialized instruction could improve very much on this. After all, there will have to be a fetch and store, as well as a test and bumps for each character moved. A high-level instruction would be further complicated by considerations of crossing page boundaries, running too long, etc. If we build in this instruction we are tailoring the machine to C; other languages such as Pascal, ADA, FORTRAN and COBOL that do not share the C convention that character strings are terminated with a zero, will find the op useless. However, even C may not find this the best strategy for character movement all the time. Large buffers should not be moved 1 byte at a time. Then there are other idioms in C, for searching tables, scanning input forward and backward, looking for other characters and an infinite number of other tasks. Is each of these to be a special op? Will the compiler have to look for complicated patterns trying to match a complex function to a complex instruction? The ROMP permits the compiler writer to combine primitives to efficiently solve the particular problem at hand for a wide variety of source languages.

As the programming community moves toward languages that are more powerful than C it becomes even more urgent to rely on basic constructs. Only the simplest languages can be based on complex, high-level messages. Thousands of hieroglyphics are much less powerful than an alphabet of twenty-odd characters. Fast, primitive operations will be required to efficiently implement the high level languages of the future.

References

1. M. Auslander and M.E. Hopkins, "An Overview of the PL.8 compiler," *Proc. of the Sigplan '82 Symposium on Compiler Writing,* Boston, MA, June 23-25, 1982.
2. George Radin, "The 801 Minicomputer," *Proc. of Symposium on Architectural Support for Programming Languages and Operating Systems,* Palo Alto, California, March 1-3, 1982.
3. Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein, "Register Allocation via Coloring," *Computer Languages,* Vol 6, No 1, 1981, 47-57.

Advanced Interactive Executive (AIX™) Operating System Structure

Larry Loucks

Introduction
When we set out to design the IBM RT PC system, we realized that the RT PC needed a full-function operating system with the ability to support continuously-running applications. In the increasingly interconnected world of advanced microcomputers, it is no longer acceptable to dedicate the computer to a single application. There must always be an operating system presence to respond to external requests.

Obviously, an architect who sets out to build a "disciplined" environment, also takes on the responsibility for making that environment functionally complete and flexible enough to satisfy the full range of applications. We took a three-level approach to the problem: make the built-in functions powerful enough to satisfy most applications, provide *controlled* access to the hardware interfaces for occasions when the built-in functions aren't sufficient, and make the operating system open-ended enough to allow for extensions to cover situations such as new types of devices.

We wanted to give users the widest possible choice of applications to run on the RT PC, so we provided ways of moving applications from the IBM Personal Computer, other UNIX environments, and IBM mainframes into the RT PC environment. At the same time, we wanted to give those applications the maximum possible benefit from the capabilities of the advanced hardware, so we incorporated into the Advanced Interactive Executive (AIX) an application development environment suitable for many existing higher level language programs, as well as the ability to process most PC DOS commands and data.

The AIX operating system kernel was derived from AT&T's UNIX System V. In light of our requirements for application diversity, operating system stability, and exploitation of the RT PC's advanced hardware features, we felt that the best approach was to provide enhancements below, within, and above the kernel. This led to the software structure shown in Figure 1. The Virtual Resource Manager (VRM) [1] controls the real hardware and provides a stable, high-level machine interface to the advanced hardware features and devices. The kernel received corresponding enhancements to use the services of the VRM and to provide essential additional facilities. The application development extensions above the kernel were integrated into the existing operating system structure. In some cases, the extensions were packaged and priced separately, but they are designed to operate as integral parts of the operating system after they have been installed.

Because we were dealing with a new hardware architecture and with large quantities of new and modified software in the system, we felt that special efforts were required to ensure excellent performance. We adopted a policy of continuous performance assessment of the operating system, starting with the earliest availability of hardware and software. The performance group had to develop new tools and procedures to assess the performance of the system while it was still immature, but the results of that effort are visible in the performance of the completed product.

Although the VRM and the AIX kernel have been "tuned" for each other, we have not precluded the ability to run other operating systems in the VRM virtual machines. Similarly, the techniques we used to virtualize existing types of devices would work for new device types as well. Both the VRM and the kernel are deliberately open-ended, to allow the straightforward addition of new functions and device support.

Creating the Right Environment for the AIX Kernel
The existing structure of the AIX kernel was not well suited to exploiting the advanced features of the RT PC hardware. Rather than making major changes to the architecture of the kernel, we built the VRM to provide a more comprehensive real-time execution environment (see Lang, et al.[1]). This environment includes multiple preemptable processes, process creation and priority control, dynamic run-time binding of code, direct control of virtual memory, millisecond-level timer control, multiple preemptable interrupt levels, and an efficient interprocess

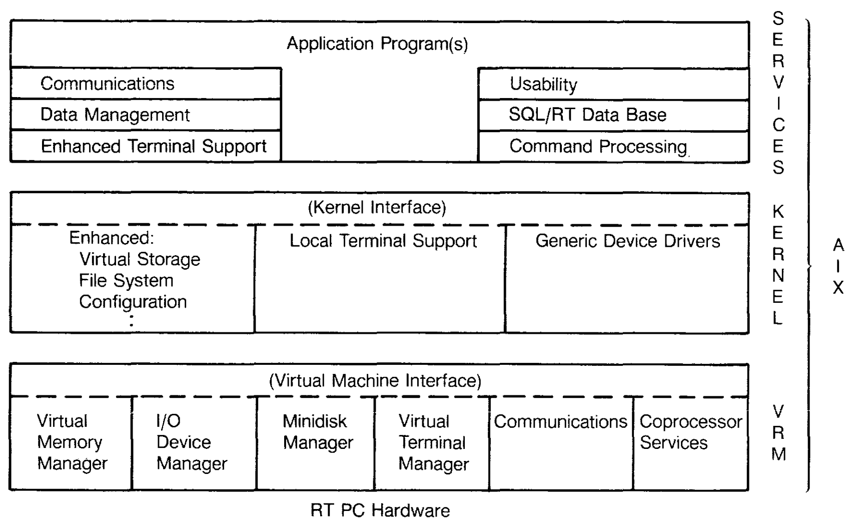


Figure 1 Overall Structure of AIX Operating System

communication mechanism for main and interrupt-level processes. The VRM software uses these features to control the ROMP processor, Memory Management Unit (MMU), and I/O hardware, and provides the kernel with interfaces to these functions.

The key to the ability of AIX to support multiple simultaneous interactive applications is the virtual terminal (see Baker, et al.[2]). A virtual terminal is a virtual counterpart of the real RT PC display(s), keyboard, and mouse. Each application initially gets a single virtual terminal to work with. The application can request creation of additional virtual terminals at will. The virtual terminals time share the use of the real displays and input devices. A virtual terminal can function as either a simulated ASCII terminal or a high-function terminal equivalent in power to the real hardware. The simulated ASCII terminal resembles a typical "glass TTY," enhanced with functions to control sound, multiple fonts, and color. Advanced applications can use the high-function terminal to obtain controlled access to all of the functions of the real display. Most of the basic operating system functions — kernel, command processing, and usability — make use of the simulated ASCII terminal, so that their functions are available on real ASCII terminals as well.

The ROMP/MMU virtual memory architecture, in combination with the VRM, gives the RT PC a demand-paged virtual memory of one terabyte, consisting of 4096 256-megabyte segments. The VRM performs page fault handling and manages the allocation of real memory, paging space, and virtual storage segments (see O'Quin, et al.[3]). It provides the AIX kernel with interfaces to control these functions and to respond to a page fault by dispatching another process. The VRM can also map memory pages within a given segment onto disk file blocks, creating a "single-level store" that makes DASD and memory equivalent.

The VRM provides the operating system with an extensive, queued or synchronous interface to the I/O devices, insulating the kernel from the details of specific devices and the management of shared devices. The correct device handler is selected on the basis of the currently-installed hardware or the configuration files and is dynamically bound into the VRM. The devices that the application sees are generic devices such as generalized fixed-disk drives ("mini-disks") or RS232C ports. In those cases where the generic devices are not appropriate, or where the real time capabilities of the VRM environment are needed by the application, the user or a third-party programmer can write C or Assembler language code to implement the necessary function, and can dynamically add that code to the VRM while the VRM is running (i.e., without re-IPL).

Problem determination in system or user-added code is supported by VRM serviceability facilities that include trace capabilities, dumps, and a debugger.

The VRM supports the PC AT coprocessor option as though it were another, rather specialized, virtual machine (see Krishnamurty and Mothersole[4]). The coprocessor runs concurrently with the execution of programs in the ROMP, but it only has access to the keyboard, locator, and display when the coprocessor virtual terminal is the "active" virtual terminal, that is, when it has control of the display. The input from the keyboard and locator are presented to the coprocessor as though they had been produced by the corresponding PC AT devices. If no display has been dedicated to the coprocessor, the display interface emulates a PC display on the system display. The VRM manages the shared system resources to ensure that the

ROMP and coprocessor operate cooperatively.

The VRM resides on a minidisk of its own in a standard AIX file system. Installation and space management on that minidisk are performed with standard AIX utilities.

Building a Firm AIX Base
The structure of the AIX kernel has been modified to allow it to operate in a VRM execution environment (see Loucks [5]). New device drivers for devices such as disk, diskette, tape, and asynch were written to use the device interfaces of the VRM. The device-driver interfaces have been extended to allow dynamic binding of a driver to a VRM driver.

The kernel has been enhanced to use the VRM virtual memory services. The kernel now provides a demand-paged virtual memory system that fully supports the 1024-gigabyte address space. The kernel occupies one (256-megabyte) segment. Each process is allocated three segments: one for program text, one for data, and one for the stack. Additional segments can be obtained for use with private or shared data, or for mapped files. The VRM map page service is used at run time to dynamically map the program text and initialized data, as well as to provide the application with the ability to map a user file into a virtual memory segment. This provides the effect of a "single-level store." The kernel uses VRM page fault information to control process dispatching, as well as allowing the kernel itself to be paged.

Historically, UNIX-based data base programs have used only the low-level disk I/O services of the kernel because the standard UNIX file system lacked several key features necessary to support them. This resulted in data base programs that were not integrated with the system, unique sets of utility commands to be learned, and a general increase in the complexity of the system. We wanted to provide an integrated environment, so the kernel file system services were extended to provide the necessary facilities to allow us to add data management and relational data base support that is built on top of the file system (see Bissell[6]). The enhancements included the ability to perform space management within a file, buffer cache synchronization on a file basis, and file and record-level locking.

The complex multi-process applications that we envisioned being run on the RT PC required more robust interprocess control facilities, so the signal (asynchronous event notification) package has been superceded by a new package that provides for more signals and cures a number of race conditions that were inherent in the original package. The standard signal package remains available for compatibility with existing application programs.

The local terminal (displays, keyboard, and mouse) now has two modes. To allow existing applications to run unchanged and new character-oriented applications to use the RT PC facilities fully, we extended the ASCII character-oriented terminal model via private escape codes in the data stream and a new set of IOCTLs to access features such as fonts, character sets, color, sound, and mouse input. For applications that need the APA capabilities of the displays, or more direct communication with the keyboard and mouse, we developed a new mode for the terminal driver that makes the full capabilities of the console hardware available in a controlled manner. The application selects the mode in which it will use the virtual terminal.

Because we expected RT PC's to be used both as single-user workstations and as traditional UNIX time-shared systems, we felt that some changes were required to support the workstation user. We have made some alterations to reduce the number of situations in which user has to exercise superuser authority. We added the ability to define more than one group to which a user belongs at any given time. This allowed us to define single users as members of the "system" group. System group members can perform a number of operations that previously could only be performed by superuser. Only the most hazardous commands are still restricted to superuser authority. This technique gives the user of a private workstation a simpler environment to work in, while preserving the existing AIX authority structure for multiuser environments.

Configuring a UNIX system has historically required an understanding of the internal structure and logic of UNIX. We felt that it was unrealistic to impose such a requirement on our prospective users, so we set out to simplify the installation and configuration processes (see Lerom et al.[7]). For those devices that can be identified internally, such as displays, the system performs an automatic configuration process. For devices that require explicit description, such as printers, we built a menu interface that obtains the necessary information from the user and makes the required coordinated changes to all of the affected VRM and AIX system files. These menus use the same VRM facilities that were provided to allow users to add device and real-time application support. The interfaces to this menu structure have been documented so that users or third-party programmers can add devices to be selected and described via the menus.

UNIX has a dual-purpose command language. The commands have been designed from the beginning to be primitives of a command procedure programming language, sometimes at the expense of ease of use when individual commands are submitted from the terminal. This makes the management of files and the performance of common operations unnecessarily complex. Many UNIX installations solve this problem by building sets of procedures that effectively constitute a command meta-language. We chose to combine the solution to this problem with the construction of a full-screen interface to AIX (see Kilpatrick and Greene[8]). The usability package provides a full-screen file management utility and the ability to request the most common AIX commands via a panel interface. The dialog manager that is included in the usability package is general enough to serve application programs as well as AIX commands (see Murphy and Verburg[9]).

To simplify the diagnosis of problems in AIX we added several debugging tools: a trace mechanism, logging of errors and system messages, and a memory dump capability.

Enhanced Application Development Environment
To be able to support the full range of modern applications, AIX needed several functional extensions. One of the most critical was the need for an indexed access method. We added a B-tree based data management program that permits either record-level or field-level access. Although it is packaged separately from the operating system, data management becomes an integral part of the file system when it is installed. Similarly, we added a data base program supporting the Structured Query Language (SQL) to provide both users and application programmers with relational data base facilities.

The higher level language compilers for the RT PC were chosen on the basis of the number and types of programs that have been written in those languages. We selected dialects that would facilitate propagation of programs from the IBM Personal Computer, other IBM mainframes, and other UNIX systems, with language extensions where necessary to support the AIX environment. In some cases the compilers have two modes — one for programs from the PC, and one for programs from minicomputer or mainframe environments. We developed a new subroutine linkage convention (see O'Quin[10]) that supports multi-module programs written in several languages.

AIX also includes a new shell that processes PC DOS commands, conversion programs that transform data from PC to RT PC format, and subroutines that allow applications to read DOS-formatted diskettes and minidisks (see Brissette, et al.[11]).

The Files and Tools applications of the usability package can be extended to cover new types of files, new actions that can be performed against those files can be defined, and new tools — including complete full-screen applications — can be added. The dialog manager in the Usability package can also be used to provide new full-screen applications with an interface that is consistent with the interface presented by Files and Tools.

The "LIBCUR" package, which supports the presentation of full-screen menus on ASYNC terminals, has received performance enhancements and has been compatibly extended to provide access to the extended font and other functions of the RT PC native displays. We also added functions such as screen division and "layering" logic to give the dialog manager a high-level, device-independent interface on which to build.

Programmers developing applications for the local terminal can have the full power of the RT PC APA displays available to them.

As a base for LAN-based applications, we included a set of primitives to support the PC Network.

Conclusion — A Good Beginning
In its Release 1 form, the RT PC software can be installed and used for production work without a large investment in learning the internals of AIX or the peculiarities of a new command language. It is a base to which existing applications can be moved and on which new applications can be built. Perhaps more important, AIX and the VRM provide a system that has been architected to be extendable.

From the beginning of the project, we have known that we could not include in Release 1 all of the functions we wanted, and that users and other developers would have needs that we did not anticipate. The open-endedness of the system results from our awareness of the limitations of prediction.

Clearly, we still need to look at the function of the system in several areas. We are not yet satisfied with the communications capabilities of AIX and its ability to function in a distributed environment. There are aspects of compiler technology that could make more effective use of the capabilities of the ROMP. Additional opportunities will undoubtedly arise as more people use the system.

References

1. Thomas G. Lang, Mark S. Greenberg, and Charles H. Sauer, "The Virtual Resource Manager," *IBM RT Personal Computer Technology,* p. 119.
2. D.C. Baker, G.A. Flurry, and K.D. Nguyen, "Implementation of a Virtual Terminal Subsystem," *IBM RT Personal Computer Technology,* p. 134.
3. J.C. O'Quin, J.T. O'Quin, Mark D. Rogers, T.A. Smith, "Design of the IBM RT PC Virtual Memory Manager," *IBM RT Personal Computer Technology,* p. 126.
4. Rajan Krishnamurty and Terry Mothersole, "Coprocessor Software Support," *IBM RT Personal Computer Technology,* p. 142.
5. Larry Loucks, "IBM RT PC AIX Kernel — Modifications and Extensions" *IBM RT Personal Computer Technology,* p. 96.
6. John M. Bissell, "Extended File Management for AIX" *IBM RT Personal Computer Technology,* p. 114.
7. Shirley Lerom, Lee Terrell, and Hira Advani, "Configuration Methods for a Personal Computer System," *IBM RT Personal Computer Technology,* p. 91.
8. P.J. Kilpatrick and Carolyn Greene, "Restructuring the AIX User Interface," *IBM RT Personal Computer Technology,* p. 88.
9. Tom Murphy and Dick Verburg, "Extendable High-Level AIX User Interface," *IBM RT Personal Computer Technology,* p. 116.
10. J.C. O'Quin, "The IBM RT PC Subroutine Linkage Convention," *IBM RT Personal Computer Technology,* p. 131.
11. Leonard F. Brissette, Roy A. Clauson, and Jack E. Olson, "PC DOS Emulation in a UNIX Environment," *IBM RT Personal Computer Technology,* p. 147.

Restructuring The AIX User Interface

P.J. Kilpatrick and Carolyn Greene

Introduction

When the UNIX kernel was adopted as the core of the IBM RT PC operating system, our User Interface Design Group received an interesting challenge. UNIX was designed as a powerful and flexible tool for computer science experimentation. To users for whom a computer is a means and not an end, however, the UNIX command language can seem complex and unpredictable. The very open-endedness that has allowed the continual expansion of UNIX's functional power over the years, has resulted in a wide variety of syntaxes and semantic characteristics. In a number of cases, too, the need to make a command useful in Shell scripts as well as from the terminal has resulted in quirky responses to unsophisticated terminal users.

The UNIX user interface also shows its age, in that it was originally designed for typewriter-like terminals connected to a minicomputer via low-speed lines. Many of the characteristics that detract from UNIX's usability today, such as extreme terseness of command language, result from design decisions intended to improve the performance of the early UNIX systems.

The process of installing and customizing a UNIX system is also somewhat complex, requiring an understanding of UNIX internal structures and processing.

Naturally, we were exacerbating these problems by porting UNIX to an environment for which it was never intended: a workstation supporting multiple virtual terminals for a single user on an all-points-addressable display. Our objective was to preserve the functional power of UNIX while presenting the user with a consistent and straightforward terminal interface.

Ground Rules

Our user interface design was subject to a number of constraints, some architectural and some practical.

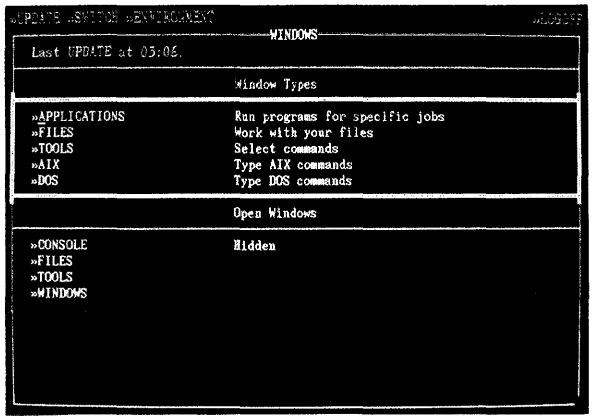
- Users who wanted to use the RT PC to run one or more specific applications should be given an interface that would enable them to install and configure the system and applications, manage their files, and perform routine system functions without being exposed to the complexities of the full command language.
- Except for the installation and configuration interface, all of the *system* user interfaces had to be available to users in substantially the same form on the RT PC display and on attached terminals. Applications, of course, would operate only on those terminals capable of satisfying their functional requirements.
- The user must be able to exploit the full command language when necessary.
- The ability of the RT PC system to run multiple concurrent interactive sessions should be an inherent part of the user interface.

In other words, we wanted to satisfy a user set ranging from UNIX experts to novice users. Rather than adopt a Procrustean, one-size-fits-all solution, we chose to provide a family of several related user interfaces with different objectives.

Windows with Personalities

A virtual terminal running an application constitutes a "window" onto the output of that application. Our user interface currently provides five kinds of windows, each running its own specialized application.

- The Windows window, shown in Figure 1, is the operator's console. It is the first thing displayed when the user logs on, and it is the base from which all new windows are created. The Windows window contains a list of the kinds of windows that can be created, and a list of the windows that already exist.
- A Files window (see Murphy and Verburg[1]) is a full-screen display of a directory in the user's file system. Selecting a file causes the user to be presented with the set of actions that can be performed on that file.



```
»UPDATE »SWITCH »ENVIRONMENT                                  »LOGOFF
                            WINDOWS
  Last UPDATE at 03:06.

                           Window Types

  »APPLICATIONS              Run programs for specific jobs
  »FILES                     Work with your files
  »TOOLS                     Select commands
  »AIX                       Type AIX commands
  »DOS                       Type DOS commands

                           Open Windows

  »CONSOLE                   Hidden
  »FILES
  »TOOLS
  »WINDOWS
```

Figure 1 A WINDOWS Window After Two Other Windows Have Been Created

- A Tools window is a hierarchically arranged list of commands and application programs that can be invoked via a panel rather than a command-language interface.

- An AIX Shell window is the equivalent of a single instance of the AIX Shell running on an ASYNC terminal.

- A DOS Shell window is identical to an AIX Shell window, except that it has been preconditioned to submit commands to the PC DOS compatibility interface of AIX.

After logon, the user can determine which interfaces are most appropriate to the tasks to be performed and then create several windows of suitable types. The windows form a ring, as shown in Figure 2.

The user can move around the ring of windows with the Alt-Action (forward) and Shift-Action (backward) key combinations. If the user has created a large number of windows (up to the maximum of 28), he or she can go directly to the Windows window with the Ctrl-Action key combination and then

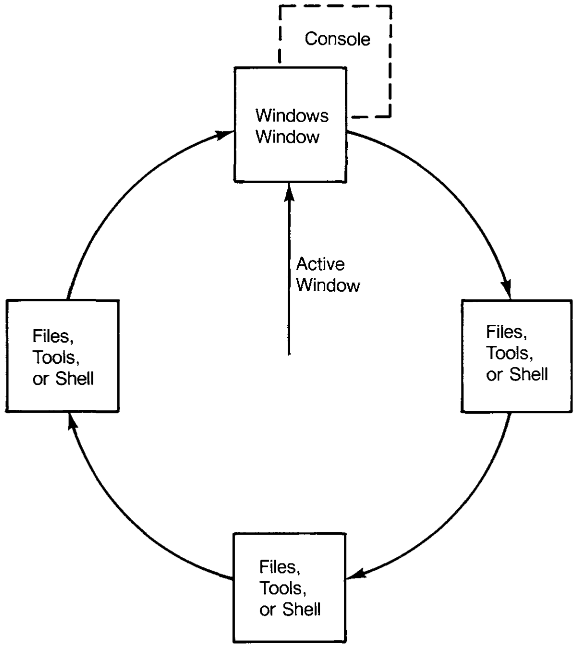


Figure 2 The Ring of Windows

move directly to the desired window by selecting it in the Windows window and selecting the ACTIVATE command.

The "Usability Package" Interface
In architecting the full-screen user interface for AIX, our objective was to increase the productivity, self-confidence, and satisfaction of the user. We also wanted the benefits of the new interface to extend to users of terminals, not just to users of APA displays. This precluded, perhaps fortunately, the use of the cute techniques of some recent products. We divided the screen into two areas: 1) the command bar at the top of the screen, and the application area taking up the remainder of the screen. All other types of information, such as panels requesting command parameters, error messages, and help information, appear in pop-up panels that overlay parts of the application area.

When the user selects something in the application area, the command bar displays all of the commands (or sets of commands) that are valid for use with that selection. Some of the commands are very close to specific command-language counterparts. Others are generic commands that result in the invocation of different AIX commands depending on the type of object being manipulated. The unifying principle is that the user should see the system as consistent. It is the system programmer's problem to deal with the underlying nonuniformities (see Murphy and Verburg [1] for a more detailed description of the Files and Tools applications).

We have not attempted to provide all of the commands via the usability interface. Instead, we have supported the most common tasks in simplified ways. The Files and Tools application programs have deliberately been left open-ended to allow us to support tasks that we may have overlooked and to allow inclusion of new applications being added to AIX.

Conclusions
In attempting to simplify use of UNIX while preserving its power, we have inevitably created a family of user interfaces rather than a single, unified interface. We believe, however, that we have provided a selection of tools that is appropriate to the diverse users of RT PC.

Naturally, the user interface will evolve along with the rest of the RT PC. We hope to see increased integration of applications with the usability package and with each other. We also expect to correct some "rough edges" that were detected late in the development cycle, such as a rather ponderous procedure for keeping a displayed directory synchronized with changes to the directory made by other terminals. We are convinced, however, that the new interfaces represent a substantial improvement over the existing command language and provide a good base for future development.

References

1. Tom Murphy and Dick Verburg, "Extendable High-Level AIX User Interface," *IBM RT Personal Computer Technology,* p. 110.

Configuration Methods for a Personal Computer System

Shirley Lerom, Lee Terrell, and Hira Advani

Introduction
RT PC AIX configuration is designed to exploit the features of the AIX operating system (see Loucks [1]) and the Virtual Resource Manager (see Lang, et al.[2]) as well as overcome some traditional problems one encounters while building a typical UNIX system.

One of the design objectives had to be ease-of-use. Configuration of a typical UNIX system can be very complex and error prone for a novice user. AIX configuration is easy to use, leads the user to supply all needed information, and handles complex tasks without bothering the user with the details. The user is also warned of error situations.

Other design objectives covered in this article are those of flexibility, availability and extendability. AIX configuration is flexible in that it supports the open architecture of the RT PC system. It is dynamic in that most changes take effect immediately, as opposed to generating and then starting a new system. It is available, since generally the system remains usable by logged in users during the configuration steps. Most important, AIX configuration is extendable to future peripherals and program products.

One of the critical design problems was to provide a structure that could be used dynamically to link kernel device drivers to VRM device drivers by issuing system service calls (SVCs). The structure had to be defined in a generic sense so that it could handle current and future device drivers, protocol procedures, and device managers. (See Figure 1.)

A second design problem was to surface all configuration information to the user in an accessible and modifiable format. What we wanted to achieve, where possible, were "table-driven" routines that could be extended to cover future device drivers, device managers, and/or protocol procedures in an integrated way, as though they had been part of the original configuration.

Configuring in a Typical Pre-RT PC UNIX Environment — The Problem

The "Privileged-User" Philosophy
A typical UNIX system, with its multi-user, multi-tasking nature, has conceptually built-in certain large system philosophies. One philosophy involves a system administrator or system operator requirement. Merely browsing a UNIX System Manager's reference book reveals chapters of information containing recommended procedures, guidelines, and mandatory steps with which 'super' user should comply. In fact, almost all UNIX systems have a superuser login id ('root' in many cases) for the purpose of system maintenance. Programs often check the user id (uid) or effective user id of the person attempting to execute them to make sure the authority is that of superuser (or root). For example, only root can 'mount' a file system, install new device drivers, or build and install a new kernel. This is acceptable for large, multi-user systems with a full-time system operator or knowledgeable UNIX programmers, but in small operations with no reliance on mainframe systems, it may not be affordable to have a system operator to maintain and upgrade the system. Upgrading a system to support a new peripheral or software package is particularly important, since it is such a common occurrence. In typical UNIX systems, it is also one of the more difficult tasks, since it involves rebuilding the kernel.

Process Steps: Building, Testing, Integrating, New Kernels
Device drivers in the UNIX kernel are the funnels through which all Input/Output operations must pass. When reading a file, the disk (or diskette) device drivers are usually involved. When printing a file, the printer device driver is used. When initiating an I/O operation to device 'x', the device 'x' driver must be used. The device driver 'knows' how to deal with the specific device for which it was written. A device driver for device 'x' most likely will not work with device 'y'. Therefore, if the new printer you wish to add is device 'y', in a classic UNIX system you must follow some or all of the following steps to install device 'y':

1. Code and compile the new device 'y' device driver.

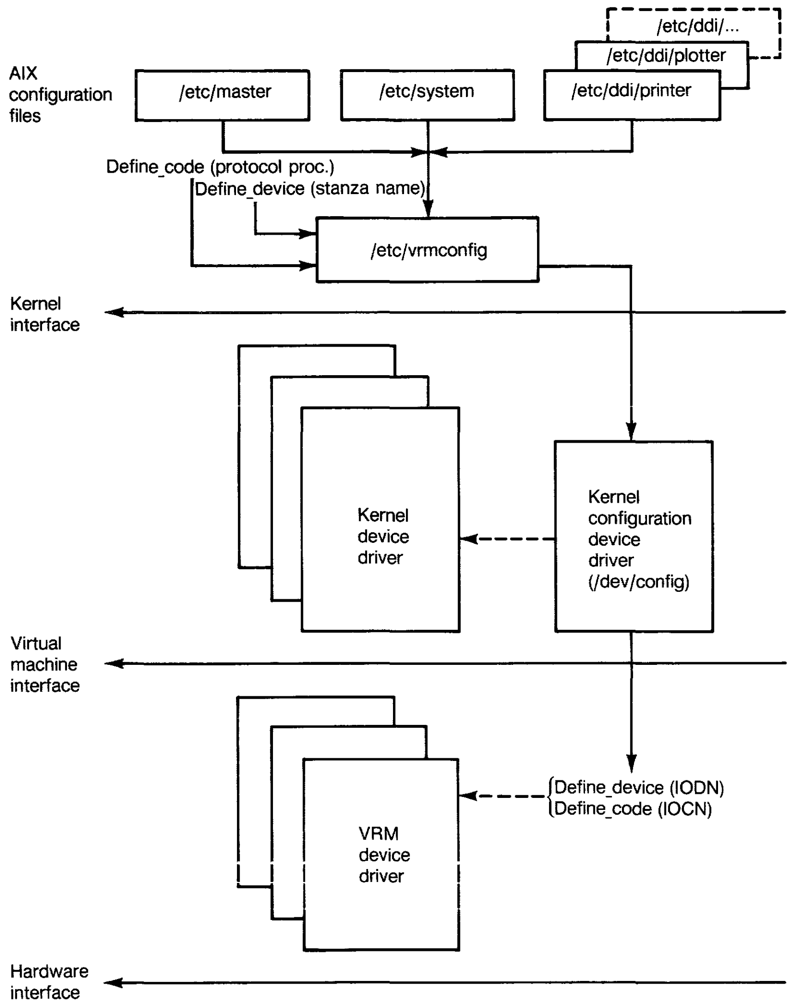


Figure 1 AIX Configuration Interface Structure

2. Archive the driver in the library appropriate to your system.

3. Modify the appropriate configuration files.

4. Build a new kernel (to include the driver).

5. Boot the new (test) kernel and debug the new device 'y' driver.

6. Repeat steps 1 - 5 as necessary till device works satisfactorily.

7. Replace the current kernel with the new debugged version.

The above steps, while not specific to any system, do reflect a general flow and are non-trivial even if one assumes a working knowledge of the C programming language and the kernel routines necessary to write a device driver. This is seemed to us to be a great deal of work for an ordinary user who simply wanted to add a printer. We tried to overcome such limitations in the AIX configuration design.

Overall Configuration Scheme
Our goals for the configuration mechanism were:

Ease of Use A need existed for a simple user interface to configure the peripheral devices. The user is assumed only to know characteristics of the peripheral device, not about AIX. In the most simple form, all the user needs to know is the class and type of peripheral device and the adapter port to which the device is to be connected.

Usability The user interface to the configuration routines needs to guide the user through the information in a 'leading prompt' fashion. It is essential that the user not need to know in advance what to do. Device configuration profiles should be initialized to their nominal state to minimize the need for user intervention.

Flexibility While keeping the design simple and easy to use, it was important to preserve the power and dynamic capabilities of AIX. That is, when the configuration is completed, the device should immediately be available to the user. This dynamic re-configuration is essential in a multi-user system.

Availability	The configuration operations, when possible, should not make the system unusable by the other logged-in users.
Extendability	Equally important, the design must not be obsoleted by new technology, devices or program products. The design is such that all configuration information is maintained in English text files. Critical information is not buried within software, but rather in user-readable files. As new devices become available, these files are updated or new files created to reflect the characteristics of these devices. This technique is used, for example, by program products to add device support.

Adaptation of the Goals to RT PC AIX
These goals would not have been attainable in a UNIX-only environment. The re-configurability of the VRM was critical to adding and deleting devices dynamically.

The architecture of the Virtual Machine Interface (VMI) was also key to reducing the number and complexity of the kernel device drivers. By providing a constant interface at the VMI, a single kernel device driver is capable of handling all printers regardless of the hardware attachment mechanism.

Externalizing the configuration information into files was important in reducing the number and complexity of the VRM device drivers. Unique characteristics of the multitude of hardware adapter cards are kept in configuration files.

Modification of Basic UNIX Concepts

Architecture of the VRM
Because of its architected Virtual Machine Interface (VMI), the VRM maintains a constant calling interface for the kernel device drivers (KDD). Since the kernel device drivers no longer deal directly with the hardware, the drivers are not as susceptible to changes in the hardware. As a result, the kernel device driver for the printer (for example) does not have to be rewritten or even modified when a new hardware adapter card (attachment mechanism) is available. Any differences in new versus old hardware are contained in the VRM device driver. This brings us to a second improvement.

Installable Device Drivers
The VRM allows for device drivers to be installed using calls at the VMI in a run-time (dynamic) environment. Specifically the Define_Code Service Call (SVC) and the Define_Device SVC provide a means for device drivers to be installed in the VRM. When installed, the driver is assigned an I/O Device Number (IODN) which is a 'tag' for interacting with the driver.

VRM device drivers can be installed in two ways. First, there is a set of device drivers known as 'core' or 'nucleus' devices that are installed at VRM Initial Program Load (IPL) time. These devices are installed by hardware diagnostic programs that run during each IPL. If the diagnostic program determines that the 'core' device is actually present in the RT PC system, it installs the VRM device driver for that device. This is true for fixed disks, diskettes, displays, and the tape devices. A user has simply to plug in a second fixed disk, for example, and it is ready to use. This is because the diagnostic program will sense the presence of the new fixed disk and install a VRM device driver to handle it. Second, the portion of the AIX operating system running at the VMI installs VRM device drivers for all 'non-core' devices (e.g., printers, plotters).

Generic VRM Device Drivers
The VRM device drivers in RT PC are written so as to be table driven. Information in a Define_Device Structure (DDS) is created at system startup and contains characteristics for the driver to use. A single VRM device driver is therefore capable of interfacing to multiple hardware adapters.

Kernel Driver Initialization Routines
A further improvement is the addition of an initialization routine to each of the kernel device drivers. Many kernel device drivers have most of the following routines:

- Open routine
- Close routine
- Read routine
- Write routine
- Ioctl routine.

An additional routine was added to AIX kernel drivers to set up (configure) key parameters. Included in the parameters passed to the KDD are:

- IODN, a 'tag' for interfacing with the corresponding VRM driver
- Virtual Interrupt Level
- KDD unique information (optional).

By way of this initialization routine, the KDD is informed which VRM device driver to use.

The printer KDD, for example, will attach and communicate with one of two VDD's depending on the type of printer in use – parallel or serial.

File Driven Configuration
Another key improvement in the AIX design is that all key characteristic information about the peripherals and hardware adapters is contained in configuration files of English text. More specifically, all information contained in the Define_Device Structure (DDS) for the VRM device driver as well as associated information for the kernel device driver (KDD) resides in files in the '/etc' directory on the root file system.

Open-endedness of Adding Peripherals
The RT PC product supports a wide latitude of capability in adding peripherals. The cases for consideration are listed below in an ascending order of difficulty.

1. Adding an IBM Base Operating System (BOS) Device
2. Adding IBM Licensed Program Product (LPP) Devices
3. Adding OEM Printers and Plotters
4. Adding Similar (to IBM) Devices
5. Adding New Device Drivers

Configuration Routines New to the Product
There are several new AIX functions introduced with the RT PC system.

vrmconfig Vrmconfig is the software routine responsible for reading information in the configuration files to define and initialize the VRM and kernel device drivers in the system. It issues the Define_Code and Define_Device VMI SVC's and initializes the kernel device drivers.

devices The 'devices' command is the software program that provides the user interface to the configuration files. These files contain all information about peripheral devices and the hardware adapter cards to which these devices attach. Additional information in these configuration files controls the flow and logic of the devices command itself. This control information is used to present the user with the minimum number of information requests necessary to describe the device being added.

minidisks The 'minidisks' program is the user interface for partitioning the fixed disk into AIX file systems.

installp The 'installp' function provides ease of installation of new program products. If the program product contains support for a new device(s), the configuration files are also updated. The AIX kernel is conditionally re-built (if a new kernel device driver is part of the program product). In many cases the installp program completely handles the installation of new software as well as readying new devices for operation.

Configuration API A significant enhancement of the RT PC system is the inclusion of library routines that provide interfaces to manipulate the configuration files. These application programs are used as a common mechanism for updating/modifying the configuraion files by the 'devices', 'minidisks', and 'installp' programs. Likewise, user-written routines could be linked to these programs (in librts.a) to deal with the configuration files.

Summary
RT PC system configuration offers the user an interface that manages the complex, file-driven elements and functions of the system and ensures that file interaction is complete and error free. In most cases, this eliminates the need for a system administrator with a special set of technical skills to do configuration.

RT PC configuration is a dynamic program that can be used during run time to alter the system to user needs at a specific moment in time. Much of the underlying complexity is transparent to the user. Flexibility is derived from the generic VRM device drivers that handle a multitude of hardware adapter cards.

Configuration is also extendable. Licensed Program Products (LPPs) offered by IBM will automatically use the configuration function to add software and hardware to the system. These programs and procedures are not limited to IBM. Third-party programmers may likewise use the AIX open system architecture to develop and install programs on the RT PC product. Any future peripheral or program product can likewise use the configuration programs and APIs to present new software or hardware to the system. If software and hardware have been added using AIX configuration programs, they will appear as an integrated part of the AIX system.

Acknowledgments
Our thanks go to the entire configuration department for their dedicated team effort — for the technical leadership provided by Nancy Springen during the architecture phase and by Grover Neuman in all subsequent phases and redirections; to Liz Hughes for assuming many key configuration routines and the complex VRM configuration component, to Terry Bouknecht for her work on DEVICES, to Lynne McCue for her work in MINIDISKS, to Carolyn Brady for later enhancing and maintaining configuration components, to Emily Havel for her work controlling the files, and finally to Sylvia Staves, who tested to ensure that our quality goals were met.

References
1. Larry Loucks, "Advanced Interactive Executive (AIX) Operating System Structure," *IBM RT Personal Computer Technology,* p. 83.
2. Thomas G. Lang, Mark S. Greenberg, and Charles H. Sauer, "The Virtual Resource Manager," *IBM RT Personal Computer Technology,* p. 119.

IBM RT PC AIX Kernel — Modifications and Extensions

Larry Loucks

Introduction

At the heart of the Advanced Interactive Executive (AIX) operating system is the AIX kernel. The kernel provides the operating environment for application programs and commands. The structure of the AIX kernel reflects our response to several key objectives of the RT PC program:

- A primary use of the RT PC was expected to be as a personal workstation, rather than as a host for a multi-user configuration.
- We had to ensure that the performance potential of the ROMP/MMU combination was not lost in performance bottlenecks.
- The system had to be tuned to operate effectively in a virtual memory environment.
- The kernel had to be made functionally and structurally robust enough to be the center of a production operating system, rather than an experimental vehicle.

The following sections describe the various changes and additions that were made to meet our objectives.

Appropriate Interfaces for a Personal Workstation Environment

Gary Miller

Auto Logon

The Auto Logon facility of AIX permits a user to be automatically logged on at the system console when AIX is IPLed, without having to enter a login name or a password. This facility is intended for those users who are using AIX as a single-user system or for those users who are the only ones to logon at the system console. The user to be logged on automatically is identified to the system when it is being installed.

When Auto Logon is in effect, powering the system on is all that is necessary to cause the user to be logged on at the system console. Auto Logon is performed when the file /etc/autolog contains the name of a valid login name as its first or only entry. This causes the system to process as if the user has entered a login name and password in response to the login and password prompts.

The name of the user to be auto logged can be changed by editing the /etc/autolog file and changing the login name in the file. Auto Logon can be negated by deleting the contents of the /etc/autolog file or simply by deleting the file itself.

Multiple Concurrent Groups

The Multiple Concurrent Groups facility allows a user to access files that are owned by any of the groups in which the user has membership. A user ID can be specified as belonging to more than one group. The "primary" group is specified in the /etc/passwd file. Any additional groups are specified in the /etc/group file. When the user logs onto the system, the setgroup system call is used to specify to the kernel all of the groups of which the user is a member. When the user attempts to access a file, the standard permission checks based on the user's ID are made for read, write, and execute/search. If the user would not normally be granted access to the file on that basis, the user's group access permissions are checked. If the user is a member of the group that owns the file, access to the file is granted.

No overt actions have to be taken by the user to enable the Multiple Concurrent Group facility. It is a part of standard AIX operation. The group whose ID appears in the /etc/passwd file is the primary group of the user. This is the group whose ID will be given to all files created by the user. The primary group will appear at the head of the list when the group membership is queried with the groups command. The primary group can be temporarily changed by use of the newgrp command.

The system makes extensive use of this facility in controlling and permitting access to certain privileged system files. This is particularly true for those commands that are considered to be for superuser or "limited" superuser use only. These commands are owned by superuser and are assigned to the system group. Read and execution permissions are given to the owner and members of the system group, with execute permissions denied for all other users. System group membership can be given to those users who are to be allowed to execute these commands.

Reduce Superuser Dependency
The AIX system is configured to allow a user on a single-user user system or multiple users sharing a system to perform many of the superuser functions without having to log on to the system as superuser or to issue the su command to gain superuser authority.

This scheme is based on the use of the AIX file permissions, making extensive use of group permissions, multiple concurrent groups, and set user ID.

Each of the files (commands, data, etc.) is assigned to a particular group, and users are assigned to corresponding groups depending on the authority to be given to the particular user. As users are added to the system, they are placed into the Staff group (group of general users). Users can then be given additional authority by assigning them to additional groups, such as the System group.

Removable Media
The removable mount facility of AIX is intended primarily to be used with diskettes which contain mountable file systems. With this facility mountable diskettes may be inserted in and removed from the diskette drive without doing an explicit mount or umount command.

The system determines that the file system on the removable media should be "mounted" when the directory on which the file system is to be mounted is the current directory and media containing a valid file system is inserted in the drive or when a file is opened on the removable media.

The system determines that the file system on the removable media should be "unmounted" when the directory on which the file system is to be mounted is not the current directory and no file is open on the removable media.

Interactive Workstation

Evelyn Thompson

The Interactive Workstation (IWS) program allows the user to easily connect, via the asynchronous ports, to another computer system, such as The Source, or another AIX system from either the AIX system console or an attached terminal. The connection can be initiated via a command interface or a menu-driven interface. The following functions are provided:

- The necessary transformations to make the system console keyboard appear as either an RT PC or an async terminal to the remote system
- Two protocols to transfer files to or from the remote system
- Facilities to allow the user to capture received data in a system file as well as display the data on the user's screen
- A phone directory function which is maintainable by the user
- A menu by which the user can alter the local terminal characteristics
- A menu from which the user can alter the data transmission characteristics
- Facilities to allow the user to utilize any of the supported asynchronous communication adapters
- Facilities to allow the user to connect to another RT PC and invoke IWS on that system to connect to a third system
- Facilities to allow two users on a given system to concurrently use IWS
- Facilities to allow the user to invoke IWS or XMODEM from another RT PC or terminal by dialing into an AIX logger.

The menu-driven interface to IWS consists of several menus. The main menu is first displayed when IWS is invoked. This menu allows the user to:

- Initiate a connection to another system by an asynchronous communication link
- Request a phone directory menu from which he or she can make the connection
- Request help information
- Request the "modify local terminal variables" menu
- Request the "alter connection values" menu
- Execute an operating system command
- Quit the IWS program.

The Connection menu, is similar to the Main menu and can be invoked by the user any time after the connection to the remote system has been established by executing a control_v key sequence. The only differences between the Main menu and this menu are that the initiation and directory request functions are replaced by:

- Send a file over an established connection

- Receive a file from an established connection

- Send a break sequence

- Terminate the connection.

The Directory menu is invoked by the user from the main menu. This menu displays phone number entries from which the user can initiate an auto-dial sequence.

The Alter menu is invoked from either the Main or Connection menu. This menu allows the user to specify or alter data transmission characteristics, such as the number of bits per character or the line speed. It also allows the user to specify the TTY port, dialing prefix, and file transfer mode.

The Modify menu can be invoked from either the Connection menu or the Main menu. This menu allows the user to change the capture file name. It also allows the user to toggle some IWS features such as async emulation mode.

Efficient Operation in a Virtual Memory Environment

Anthony D. Hooten

The Virtual Machine Environment of AIX
The AIX operating system kernel executes in a virtual machine maintained by the Virtual Resource Manager (VRM). The VRM provides virtual machines with paged virtual memory, with paging support logically hidden from the virtual machine. A virtual machine may treat the memory as if it were physical memory with highly variable access times. The VRM supports a large virtual memory, up to 2^{40} or one terabyte. The effective addresses generated by instructions are 32 bits long, with the high-order 4 bits selecting a segment register and the low-order 28 bits providing a displacement within the segment. The segment registers contain a 12-bit segment ID. The 12-bit ID plus the low order 28 bits of the effective address yield the 40-bit virtual address. A virtual machine may have many segments defined. To access one of these segments, the virtual machine loads a segment identifier into one of the 16 segment registers. Segments are private to a virtual machine unless the virtual machine that creates the segment explicitly gives other virtual machines access to that segment. The 16 segment registers represent part of the context-switching aspect of the multiple-process model in AIX.

In addition to segments, there are two other types of virtual memory objects: pages and bytes. Pages consist of 2048 bytes. A segment can contain from 1 to 131,072 pages. Protection is available at the page level. Protection information is stored for individual pages and then some specifics of the protection mechanism are selected when a segment register is loaded.

Pages are brought into active storage (operating system memory) on a demand basis via page faults. A page fault is a memory exception caused by a program trying to reference data that is not currently in real storage.

Virtual Memory Program Management Extensions
The AIX kernel has been enhanced to use the VRM virtual memory services. This section will discuss three AIX program management extensions which take advantage of the advanced virtual memory support. These are the AIX segment register model, demand paging of both users and the kernel, and a process fork enhancement.

Segment Register Model
The segment register model for AIX is as follows. At any given time, the IDs of up to 16 segments may be loaded into the segment registers. Each of the 16 segments may be up to 256 megabytes. Each page in a segment is individually protected for kernel access and user access. The AIX kernel occupies segment register 0. Most of the kernel segment is page-protected no-access for the user. A few kernel-segment pages used to transfer data from the kernel to a user process are protected read-only for the user. Each user process is allocated three segments. Segment register 1 is used for the user text segment. The text segment is protected read-only for the user and read-write for the kernel (so that the kernel can modify programs being debugged). The user data segment occupies segment register 2, and has read-write access. Segment register 3 is used for the user stack. The top of the stack holds the user "u-block," which is protected no-access for the user and read-write for the kernel. The u-block is used by the kernel for process management. The rest of the stack is protected read-write for both the user and kernel. Segment registers 4 thru 13 are used for shared-memory segments and for mapped data files. Shared-memory segments provide a means for sharing data among multiple processes. Mapped data files are described in the following section. Segment register 14 is used by the VRM to perform DMA operations. Segment register 15 is used to address the I/O bus directly.

Demand Paging of Both Users and AIX Kernel
Both the users and the kernel execute in demand-paged virtual memory. When a user-process reference to a page results in a page

fault, the VRM notifies the kernel, so that another process can be dispatched. This page fault notification results in improved overall system performance, since process switching can occur when a process is waiting for a page fault to be resolved. The kernel is only preempted when a page fault occurs on user data. All other page faults which occur for a kernel process are handled synchronously, with no preemption of the kernel process.

Process Fork Enhancement
The AIX "fork" system call creates a new process. The new process (child process) is an exact copy of the calling (parent) process's address space. The address space consists of text, data, and stack segments. Typically, when executing a new command, the "fork" system call is followed by an "exec" system call to load and execute the new command in the new copy of the address space. This results in replacing the forked address space with the address space of the new command, thus undoing much of the work of the fork.

Copying the current process's address space is expensive and time-consuming — too much so to waste, if it is to be subsequently deleted by the "exec" system call. The VRM "copy segment" SVC creates a new segment, but delays the actual copying of the data until one of the sharing processes actually references the data. Therefore, most of the data will not have to be copied when an "exec" system call follows, thus saving the time and memory required for the copy. The AIX "fork" system call uses this VRM copy-segment facility to create the segments of a new process. This enhancement of "fork" reduces wasted effort.

AIX and Mapped Data

Mapped Page Ranges
Simple paging systems usually suffer from conflicts between file I/O and paging I/O. For example, a file device driver may read disk data into a memory buffer, then the paging system might write that buffered data out to disk. Obviously, some coordination is required between the AIX operating system kernel and the VRM to prevent this.

Potential duplication of effort also exists with program loading. A loader may read a program into memory from the program library part of the disk, then the paging system uses another part of the disk to store the program when it is paged out. Having the VRM page the program directly from the program library saves having to explicitly load programs and also eliminates space wasted by copying the program out to a page area of the disk.

Carried to the extreme, only the paging system would need to be able to do physical I/O. The AIX file system manager could tell the VRM the mapping between data on the disk and virtual memory pages, and the paging system could then perform all the physical disk I/O.

The close interaction between the AIX kernel and the VRM offers several distinct advantages:

- Reduction in secondary paging space since many permanent objects, such as program text libraries, can be paged directly from their permanent virtual disk location

- Improvement of performance since the centralized VRM paging supervisor is in a better position to control disk contention and paging

- Simplification of the data addressing model.

The VRM supports a means by which AIX can map the disk blocks of a file to a virtual memory segment and have physical I/O performed by the memory management component of the VRM. This mechanism is known as "mapped page ranges."

The AIX kernel takes advantage of VRM mapped page range support, and applications in AIX benefit from this mapped page range support both implicitly and explicitly.

Mapped Executables
Implicitly, the AIX kernel implements mapped page range support in the form of mapped executables. When a program is loaded, or "execed" in AIX terminology, the AIX kernel maps the program's disk blocks to distinct virtual memory text and data segments. The AIX kernel performs very little physical I/O to load the program. Only the program file header is "read" by the kernel. All remaining disk I/O is demand-paged as the program is executed. This results in a significant performance increase for large programs, which without mapped page range support would have to have been read entirely into memory, and possibly paged out by the paging supervisor.

Mapped Data Files
Explicit AIX mapped file support consists of a system call interface to the data file map page range facilities. The "shmat" system call, with the SHM_MAP flag specified, is used to map the data file associated with the specified open file descriptor to the address space of the calling process. When the file

has been successfully mapped, the segment start address of the mapped file is returned. The data file to be mapped must be a regular file residing on a fixed-disk device. Optional flags may be supplied with the "shmat" system call to specify how the file is to be mapped. If the flag SHM_RDONLY is specified, the file is mapped read-only. If the flag SHM_COPY is specified, the file is mapped copy-on-write. If neither of the flags are specified, the file is mapped read-write. Before a file can be mapped read-write or copy-on-write, the file must first have been opened for write access. Before a file can be mapped read-only, the file must first have been opened for read and/or write access.

All processes that map the same file read-only or read-write map to the same virtual memory segment. This segment remains mapped until the last process mapping the file closes it.

All processes that map the same file copy-on-write map the same copy-on-write segment. Changes to the copy-on-write segment do not affect the contents of the file resident in the file system until an "fsync" system call is issued for a file descriptor for which copy-on-write mapping was requested. If a process requests copy-on-write mapping for a file, and the copy-on-write segment does not yet exist, then it is created, and that segment is maintained for sharing until the last process attached to it detaches it with a "close" system call, at which time the segment is destroyed. The next request for copy-on-write mapping for the same file causes a new segment to be created for the file.

A file descriptor can be used to map the corresponding file only once. A file may be multiply-mapped by using multiple file descriptors (resulting from multiple "open" system calls). However, a file can not be mapped both read-write and copy-on-write by one or more users at the same time.

When a file is mapped onto a segment, the file may be referenced directly by accessing the segment via Load and Store instructions. The virtual memory paging system automatically takes care of the physical I/O. References beyond the end of the file cause the file to be extended in increments of the page size (2K).

Experience with AIX has demonstrated that significant performance benefits can be derived from the judicious use of mapped file support for data file manipulation. A significant amount of system overhead is eliminated by mapping a data file and accessing it directly via Load and Store operations, rather than conventional access via "read" and "write" system calls. "Read" and "write" system calls are still supported, even when the file being accessed is mapped. AIX mapped file support determines whether or not a file is mapped when a "read" or "write" system call is requested for the file, and accesses the file appropriately if the file is mapped. An additional level of efficiency has been found to result from use of the processor-architecture-specific "memcpy" subroutine in conjunction with mapped file support. This subroutine takes advantage of Load Multiple and Store Multiple instructions to perform fast data movement.

Building a "Production" Operating System
A number of enhancements were needed to make AIX suitable for the wide variety of customer environments and applications that we expected it to support. Generally speaking, we felt that we needed to give applications a consistent interface to work to for virtual terminals and communications sessions, improve the performance characteristics of the system, make additions to its application-development capabilities, and simplify the amount of development required to support new devices.

I/O Management
We restructured the I/O Management area of the kernel to make effective user of the VRM's I/O facilities. Instead of a specialized device driver for each distinct device, we created a family of generic device drivers that are capable of supporting a number of unique devices of a given class. For example, a single "async" device driver handles async, RS-232C, and RS-422 interfaces. Truly device-specific considerations are left to the VRM device drivers, which can be added or replaced dynamically without bringing down the system.

Multiplexing
We added a facility to allow dynamic extensions to a file system. If the multiplex bit in the special file inode is on, the last qualifier of the file name is passed to the character device driver. The driver looks for the file outside of the nominal file system. This facility is used to deal with virtual terminals and communications sessions as files.

File System

Alan Weaver

The AIX file system takes advantage of the virtual device interface provided by the VRM. To improve performance, we increased the block size of the file system and the buffer cache to 2048 bytes. To permit AIX to accommodate an indexed data management feature and a data base manager, we added the ability to synchronize the buffer cache with the fixed disk on a file rather than a file system basis, added locking facilities, and incorporated facilities to recover space in sparse files.

Use of Minidisks
The VRM provides the ability to divide a given fixed disk into a number of minidisks. This permits the separation of file systems for different purposes onto different virtual devices.

AIX will let a user make 1 to "n" file systems on a physical disk, where "n" depends on the size of each file system and of the disk device. Each file system is built in a separate VRM minidisk.

AIX uses 512-byte blocks for diskette file systems and 2048-byte blocks for disk file systems. With the larger block size the number of interrupts to be handled is reduced, resulting in faster effective transfer of data to real memory.

The space on each minidisk that contains a file system is divided into a number of 2K-byte blocks, logically addressed from 0 up to a limit that depends on the size of the minidisk. A corresponding cache of 2K-byte buffers is used to reduce re-reading of blocks.

Buffer Cache Synchronization
Cache buffers are normally only written to permanent storage before the buffer is used again or with the "sync" system call. AIX has, in addition, the "fsync" system call that works on an open-file basis to force the modified data in the cache buffer to permanent storage and does not return until all of the buffers have been successfully written. This gives the user more control over the data on the disk and permits an application such as Data Management Services to force writing of only those buffers that really need to be flushed.

Dynamic Space Management
AIX has two system calls to recover space within once-sparse files. The calls are "fclear" and "ftruncate."

- fclear — zeroes a number of bytes starting at the current file position. The seek pointer is advanced by the number of bytes. This function is different from the write operation in that it returns full blocks of binary zeroes to the file by constructing holes and returning the recovered blocks to the free list of the file system.

- ftruncate — removes the data beyond the byte count in a file. The blocks that are freed are returned to the free list of the file system.

File/Record-Level Locking
AIX file and record level locking extensions allow an individual file to be locked in either an advisory or enforced form. The advisory lock notifies the caller of 'lockf' if the requested region of the file is locked. Enforced lock protects the locked region from access by readers and writers even if they have no knowledge of the locking facility. If the object being locked is a directory or a special file, only advisory locks can be obtained.

Records may be of any length ranging from one to the maximum of the file size. The data for a locked record does not need to exist in order to obtain a lock on the record. Locks may be applied beyond the current end-of-file or over an area that has not been written (sparse file regions).

Process/Program Management

Deb Blakely, Carolyn Jones, Conrad Minshall

Signals Enhancements
In addition to the standard set of System V signals, AIX provides an enhanced signal facility. This facility allows a program to mask and block each type of signal while it is executing. If a signal is received while it is blocked, it is queued up and handled after that signal type is released. However, only one of each type of signal will be queued. Except for the SIGCLD signal, all subsequent signals of the same type will be ignored. All SIGCLD's will be queued and processed. Up to 32 different signals are supported by the enhanced signals package, but only those defined in file /include/sys/signal.h can be used. These are the same signals used by the standard facility. The following are brief descriptions of the system calls that make up the enhanced signal facility:

sigblock	Adds specific signals to the list of signals currently being blocked from delivery.
sigsetmask	Sets the signal mask (the set of signals to be blocked from delivery) to a specified value.
sigpause	Sets the signal mask to a new value, pauses until a signal not blocked by the mask is received, and restores the signal mask to its original value.
sigstack	Allows users to define an alternate stack to be used for signal handling or get the state of the current signal stack.

sigvec	Allows users to specify how a specific signal is to be handled. The user can specify whether the signal should be blocked, ignored, or processed by a handler routine, and whether the signal should be processed on the current stack or a special stack.
execve	Starts a new program in the current process, resets all signals that are being caught by the original program to terminate the new program, resets the signal stack state, and leaves the signal mask untouched.

Buffer Bypass Variations
"Buffer Bypass" is a form of disk I/O which, like raw I/O and mapped files, bypasses the kernel's buffer cache, transferring data directly between the VRM disk device driver and AIX user processes. This offers direct and indirect performance gains when it is unlikely that the data will soon be re-accessed. The direct gain is the lack of a memory-to-memory copy of the data. The (more substantial) indirect gain is the generally improved cache hit ratio which results from not replacing useful cache blocks with data that is unlikely to be reused.

The implementation of buffer bypass is not device-specific. Requests are in terms of a file offset and a count, unlike raw I/O requests, which are in terms of contiguous physical blocks. Within the rdwri loop we detect when we need all of a block and (if buffer bypass has been requested) we call an asynchronous block I/O routine which bypasses the buffer cache. After exiting the loop, we wait on all outstanding asynchronous operations. A separate pool of buffer headers, pointing into user space instead of kernel buffers, maintains state information, allowing process switch. A given call into rdwri will end up allocating a ring of these buffer headers, one of which is a ring header. The parallel asynchronous operations permit the VRM disk driver to schedule for minimum arm movement and helps lower the interblock overhead, so that physically adjacent blocks can be read without waiting an entire rotation.

Buffer bypass has not been made available above the kernel. In a virtual memory environment, we consider mapped files superior. For small transient processes that reference all or nearly all of their pages, buffer bypass may have a performance advantage. But how small is "small" and how transient is "transient?" That is, where is the breakeven point and how is it dependent on current memory load? In the presence of these unresolved questions, it was decided to use mapped files wherever possible — buffer bypass has been limited to use by the exec system call for programs whose segment alignment disallows mapped execution (i.e.. for programs linked without the −K option).

IPC Queue Extensions
System V interprocess (IPC) message services have been extended to give more information when receiving IPC messages. The new function call is "msgxrcv".

The msgxrcv function returns an extended message structure that contains the time the message was sent, the effective user ID and group ID of the sender, the node ID of the sender or zero if the sender was on the local node, and the process ID of the sender.

The IPC design model has been changed from that of queue = file and message = record to that of queue = directory and message = file. This means that the same kind of information found on a file can be found on an IPC message: user ID, group ID, time, etc.

Applications and servers can use the additional information found in the extended IPC message structure to check permissions and send time. Servers can now validate requests based on the information in the extended IPC structure, rather than starting new processes that take on the properties of the program being served. The timestamp can be used to make sure the message is not an old one or to perform other tasks based on time priority.

Terminal Support

Rudy Chukran

AIX terminal support is tailored to work in the VRM environment, where terminals are virtual constructs rather than real devices. It permits applications to use multiple virtual terminals and to access their virtual terminals in either extended ASCII mode or in "monitored" mode.

Console Support
In order to take advantage of the unique functions provided by the Virtual Terminal Manager subsystem of the VRM, a console device driver was created and modeled on the RS232 terminal device driver (tty). This new device driver is referred to as the HFT device driver. It provides support for a console consisting of a keyboard, mouse or tablet, speaker, and up to four displays.

Multiple Virtual Terminals
Some device semantics were established to allow programs to create new virtual terminals

and access existing ones. If a program wishes to create a new virtual terminal, the open system call is issued on the device /dev/hft. That special file is designated as multiplexed by setting the "multiplex" bit in the inode. If an existing virtual terminal is desired, the program opens the device /dev/hft/n, where n is the character representation of a decimal number. This number is referred to as the channel number, which may also be interrogated by issuing an ioctl system call on the file descriptor in question.

If a program needs to know about and control activity on all the virtual terminals associated with the console, it opens the device /dev/hft/mgr. This gives the program access to the screen manager component of the VTM subsystem. The program may now query the state of all the virtual terminals, activate any terminal, or hide any terminal by issuing an ioctl.

Extended ASCII Mode
The default mode for a virtual terminal simulates an enhanced version of the standard ASCII terminal. It permits programs built for that interface to run with minimal change. It also permits new versions of such programs to access the sound and mouse functions.

Monitored Mode
In order for a program to operate a bit-mapped display in bit mode, the program deals directly with the hardware display adapter by storing to the memory-mapped I/O bus. This is done for reasons of speed. Some rules were established which are to be followed for programs which operate a display in bit mode and still allow other programs to operate using other virtual terminals.

Since the hardware protects the I/O bus from access by user programs, a program must request bus access by opening the /dev/bus special file. This open sets the bits in the control register which disable bus access protection. This control register is saved and restored for every process dispatch, thus maintaining security of the bus from unauthorized programs.

Next, the program puts the terminal into monitored mode with a control in the output stream. The control may optionally specify that key data be entered directly into a user buffer, thus bypassing clist processing. Otherwise, key data is read through the standard read system call.

The program next does an ioctl to set the signaling protocol. Since other virtual terminals may be activated at any time via the Next Window key, a program operating in monitored mode must relinquish the display hardware to the operating system upon request. This request mechanism is done with signals. When the program is ready to begin display activity, it issues a Screen Request control in the output stream. When the system determines the display is available, it sends the program a signal denoting display availability. The program can now change the hardware settings without interference from the system. When the Next Window key is pressed, the system sends the program a signal to relinquish the display. The program now has a fixed length of time to output a Screen Release control, which signifies that the program has saved whatever states need to be saved. If the program does not respond with a Screen Release, all processes in the tty group are sent the SIGKILL signal.

The program is now ready to access the display. The hardware registers and refresh memory are stored into by dereferencing a pointer which contains the appropriate address in the I/O space. When the program is permanently finished with the display, it would reverse the steps just described, thus leaving the virtual terminal in the same state as when the program began.

Even though operating a terminal in monitored mode is complex, the speed of direct hardware access is attained, and the protected environment of a multiuser system is preserved.

Printer Support

Jim Chen, Larry Henson

Device Driver
The printer device driver provides the interface to the VRM from the kernel environment. Up to eight concurrent printers (/dev/lp0 through /dev/lp7) are supported. Enhancements have been made to provide better error recovery procedures. Errors, as they are discovered, are returned to the application environment only if the application requests that they be returned. A new set of ioctls has been defined to allow printer control from the application. LPRVRMG and LPRVRMS get and set the VRM define device structure associated with a printer. This configuration information and error status allow the user to control the error processing and printer setup. LPRUGES returns the AIX device driver's view of the error situation. After the error has been determined, LPRUFLS allows currently queued buffers to be flushed. LPRURES will tell the VRM to resume printing the job. LPRGMOD and LPRSMOD get and set the synchronous vs

asynchronous printing modes and the option to be signalled when an error occurs.

Printer performance has often been a problem when high speed printers were used. The device driver now supports both synchronous and asynchronous write system calls. The device driver returns immediately after an asynchronous write is queued. A synchronous call returns after the write is finished. Where feasible, buffers are output without making a copy. Each of these functions is performed for both serial and parallel printers.

Previously, serial printers have been run through the tty device driver. Since tty is optimized for terminals, getting the right function for printers has been difficult. By adding serial printer support to the printer driver, the full performance and error recovery enhancements can be utilized. Ioctls LPRGETA and LPRSETA allow the baud rate, character size, parity, and number of stop bits to be queried and set. The splp(1) command has been extended to do a stty-like setting of these options.

Replaceable/Addable Backends
The print command allows user access to the queuing environment (see Figure 1). Multiple queues per printer allow the same printer to be used for different job types. Multiple printers per queue can keep the output flowing in case one printer is unusable. The qdaemon provides background control of the queues. Started up by the qdaemon, backends do the work of getting the data to the device drivers.

The user should not have to know the details of how each printer works. By providing a more general printer-support structure, we made it easier for the user to install and use

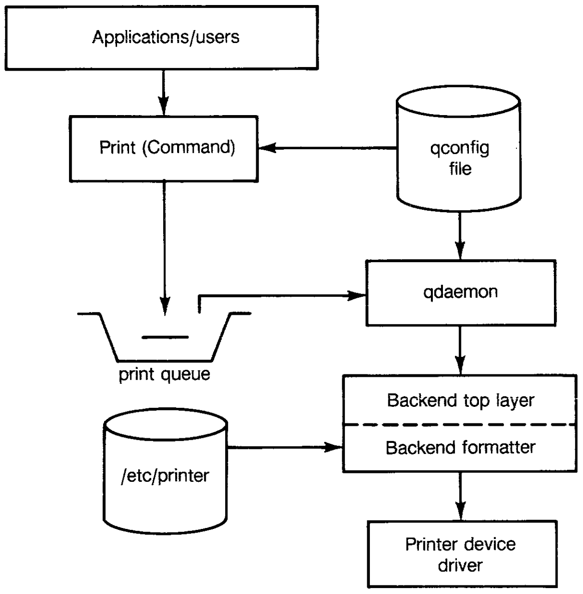


Figure 1 Print Subsystem Structure

one of the new IBM printers without knowing the details of how it works. When the system is set up, the customer describes the printers that are to be used, including such factors as paper size and default print quality. If one-time changes are required, a command line parameter will override the configuration already set up for a single job. These configuration options allow printers to be set up for different types of jobs. Thus, existing applications will work on the new printers without changing the application. The ability to add new printers in a transparent way simplifies change-over requirements.

A multiple queueing environment provides for using several printers concurrently. Replaceable backends for different printers associated with a queue allow the print subsystem to function as needed in different situations. We have provided a backend to support appropriate IBM printers. Error messages are routed back to the user for both fatal errors and intervention conditions. After the intervention condition is corrected, printing resumes automatically. A generic data stream will print on any of the supported printers. Applications are simplified by having to deal with a single printer type. Formatting for the specified margins, justification, and image graphics support help the user to get the output needed.

How can a user attach a printer that is not supported by the IBM backend? If the printer uses the 5152 data stream, that printer can be configured to run through the IBM backend. If the data stream is like 5152 with extensions, the relevant functions can be defined as being on the printer and used as desired. If a dissimilar printer is desired, the user can write his own backend to be used with his printer. This user-written backend can still be used concurrently with the IBM backend.

Extended Character Set
The use of the 7-bit ASCII code definition in 8-bit-byte machines has created some problems. For simplicity, most applications have adhered to the 7-bit standard when writing portable code. While avoiding the problem of how to use the 8th bit, it allowed applications to use that bit for whatever purpose they wished. This "usable" 8th bit solved many a sticky problem for applications that needed "tricks" in their data stream, but it also created a portability problem between applications. Applications that did not use a pure 7-bit data stream could not understand applications that had polluted their data stream with a different 8-bit variation.

With the advent of the IBM PC, many new programs have been written to conform to the PC code page mapping. This mapping uses the 8th bit to map graphics for code points from 128 to 255. This extension has allowed programmers to print and display many scientific and international graphics not previously available to them within the definition of 7-bit ASCII codes. We considered it important to establish a code set definition that could support applications from both worlds. AIX display and printer support for 8-bit codes was implemented to help meld PC applications into the world of AIX. The 8-bit support is compatible with 7-bit ASCII applications and provides an additional degree of commonality with a large number of PC applications and files.

While the 8-bit extension is useful in integrating PC applications, there still exists a large problem in representing all the graphics needed for scientific and non-U.S. applications. Over the years, we have identified and documented most of the character-graphics requirements for scientific and international applications. These character graphics have been organized and standardized across IBM. Each code page is a set of 256 graphics, usually grouped by countries (e.g., UK English, France, Germany, Spain) or major application (e.g., PC or Teletext). These code pages, if handled independently, represent thousands of characters and graphics. However, there are many redundant characters and graphics (mainly alpha-numeric and punctuation characters). This presents a sizable problem for applications to store and process these character mappings to provide extended support for scientific and international symbols.

To aid programmers in dealing with this problem, the AIX system provides a canonical mapping of the most widely used IBM code pages required by scientific and international applications. The display and printer subsystems provide controls for accessing these code points. Data stream controls provide switching to one of three code pages. These code pages are designated: P0, P1, and P2.

The base code page, P0, is based on the IBM PC display font with the exception that the first 32 code positions contain controls instead of graphic characters (which were moved into P1). This base code page allows most applications PC compatibility without any changes. In order to access graphics on code pages P1 or P2, application programs need to imbed switching controls for the printer or display in the output data stream. The application program also needs to use switching controls to return to the original code page. Each entry in a code page can be selected by its 8-bit offset value in the code page. For displays, the 8-bit offset is added to a code page offset value in order to access a particular code point. For printers, once a code page has been selected by an ASCII escape sequence, 8-bit code point offsets select graphics in the active code page.

The extended graphic characters defined in P0, P1, and P2 fulfill the major support requirements for the US, Europe, Teletext, and scientific symbols.

Floating Point Support

Richard Eveland

The RT PC system provides enhanced services for floating point arithmetic. These services are disigned to support the Institute of Electrical and Electronic Engineers' (IEEE) new standard for Binary Floating Point Arithmetic (754). The floating point package is utilized by the C, FORTRAN, and Pascal compilers for all floating point operations. Floating point operations can be further enhanced with the addition of the hardware Floating Point Accelerator feature.[1]

The compilers perform floating point operations by making subroutine calls to the set of floating point routines located in the kernel's segment 0. The interface to these routines is via a vector of entry points at a fixed location in memory. Although these routines are part of the kernel, they are executed in user mode to avoid the overhead of a system call. This area of the kernel is read-only protected by the page protect mechanism to prevent modification by user programs.

The floating point routines provide an environment of six floating point registers, with a status register that controls exception and rounding modes. The floating point registers may contain either a single-precision or a double-precision floating point number. Basic operations are in a two-address (source-destination) form, allowing either register-register or immediate-register operations. A no-result flag allows a routine to return to the caller without bringing the result out of the destination register and returning it to the caller. This allows the ROMP processor to continue executing instructions in parallel with the Floating Point Accelerator. The Accelerator can be given a second operation to perform before the first is complete, e.g., an Add followed by a Multiply, further increasing throughput.

The Floating Point Accelerator has 32 sets of floating point registers available for user processes. These are allocated to a process by the kernel only if the process actually performs a floating point operation. This way the user program does not have to specifically ask for a hardware register set. When there is no Accelerator, floating point subroutines emulate the floating point registers in a reserved area on the user's stack.

The kernel provides two sets of floating point routines: one set that implements the functions entirely in software and another set that utilizes the Floating Point Accelerator hardware for most of the floating point operations. The kernel installs pointers to the appropriate set of routines into the vector of entry points at machine initialization time. Thus programs using this interface will automatically use the Floating Point Accelerator card when it is present, but will use the software emulation subroutines when there is no card. Programs compiled to use this "compatible" mode have the ability to run on any machine, regardless of whether or not the Accelerator option is present.

Although the presence of the Floating Point Accelerator will significantly speed up the floating point operations in user programs compiled in "compatible" mode, the maximum benefit is achieved by compiling the program to drive the Floating Point Accelerator directly with in-line code. For C and FORTRAN programs this may be done by compiling the program with the "direct" option (− f). This results in maximum speed for floating point operations by avoiding the subroutine interface for most floating point operations. However, the Floating Point Accelerator card must be present for these programs to run. Direct mode versions of the C and math libraries are provided to be linked with user programs compiled this way.

The Floating Point Accelerator's hardware floating point unit, the NS32081, does not implement all of the functions required by the IEEE floating point standard, e.g., handling denormal operands. When such an event occurs, the hardware causes a Program Check interrupt which is handled by software emulation routines in the VRM. The VRM routines put the correct result back on the card so that the event is transparent to the running program.

Reliability/Availability/Serviceability (RAS)

Ellen Stokes

The IBM RT PC system RAS support is designed to provide a coherent and consistent set of error detection and correction schemes. Wherever possible, functions and components are self-diagnosing and correcting; that is:

- Error messages with clear unambiguous meaning are generated.
- Formatted error logs are automatically generated.
- Dump facilities are provided.
- Error analysis routines support software and hardware problem determination.

A primary objective of the RT PC system RAS support is to provide problem determination and correction for the customer, for a customer service vendor, or for IBM at the request of the customer. As such, the system must be reliable in all respects, but in the event that there is a failure, the system must be easily and quickly diagnosed and recovered. Note that "the system" is defined as that portion of the system which is IBM developed and/or controlled.

One of the major challenges of the AIX RAS design was to provide a consistent set of user interfaces and information across all components — VRM, kernel, and applications.

AIX error messages emphasize problem resolution. The user should be able to diagnose any "non-catastrophic" problem without resorting to offline documentation. "Catastrophic" may be defined as any problem for which there is no visible means of doing problem determination with the online facilities (e.g., system fails to IPL) and which results in the user being unable to continue the work session. Problem determination may be approached in several ways in the IBM RT PC environment, but the essence of problem determination is to give the user the necessary information for problem correction at the highest possible level *within the system*. The user will generally be able to rely on a single message for the information required to manage a function to successful completion. If the level of user sophistication or problem complexity requires the exposition of additional information, a help file can be displayed by the user if one exists. If additional information is required by the user, diagnostic tools are available in IBM-supported LPPs which provide detailed execution time flow and error analysis. Dump process execution may be initiated by the user to view the state of the system at the time a repeatable error occurs.

The following sections describe the various problem determination facilities in the RT PC system.

Trace

The trace function is intended to provide a tool for general system/application debug and system performance analysis. Trace monitors the occurrence of selected events in the system. Important data specific to each of these events is recorded on disk. When the user needs to view this data, a trace report program formats the trace data in an intelligible form. The report program sorts the disk file by date and time, providing a chronology of the system's behavior. The trace function may be started either by the user or by an application.

The user has two commands for controlling the operation of trace: start and stop. When starting trace, the user should specify a profile. This profile is an AIX file that contains a list of all the classes of events the user has selected to trace, listed by event type with a descriptive label. Any number of trace profiles may exist in the file system. There is a default profile in /etc/trcprofile. This default profile is used if no profile is specified when trace is invoked. However, it is advantageous for the user to tailor a profile to his own needs.

The trace function takes additional information about the trace session from the RAS configuration file /etc/rasconf. This file contains configuration data for all RAS functions. The entry for trace includes the file name where trace data is to be stored (default /usr/adm/ras/trcfile), the maximum size of the file name (default 80 blocks), and the trace buffer size (default 2 blocks).

Trace can operate at all levels of the system: below the VMI, in the kernel, and at the application level.

Below the VMI, trace functions are handled by the VRM trace collector and process. The trace process is initiated by a Send_Command SVC (trace on) which sets up the trace environment and starts the process. The VRM trace collector receives trace information from the trace points in the VRM and double buffers them. When a buffer reaches a threshold number of entries, the VRM trace collector notifies the VRM trace process and the VRM trace process sends that buffer to the trace application to be written to the trace file. The trace process is terminated by a Send_Command SVC (trace off).

In AIX, the kernel trace device driver is the central control point for trace functions. This trace device driver /dev/trace has three minor devices which correspond to the three levels of the system's software. The application data is handled by /dev/trace, kernel data by /dev/unixtrace, and VRM data by /dev/vrmtrace. The trace device driver controls the allocation of buffers to collect all trace data and handles the reading and writing of the data in the buffers. It also issues the Send_Command SVC which initiates trace in the VRM. The trace device driver has an interrupt handler which receives the interrupt from the VRM trace process indicating that a VRM trace buffer has reached its threshold and needs to be emptied. AIX kernel entries are written to the AIX trace minor device /dev/unixtrace by the trsave macro.

On the application level, the trace daemon is the primary process for trace activity. The process issues the ioctl command which sets the trace points "on" in the application, kernel, and VRM according to the specification in the selected trace profile. It also forks two child processes (as needed) which gather trace entries from the VRM and kernel trace buffers and write them to the trace log file. The parent trace daemon reads the application-level buffer. For both AIX extensions and applications, trace entries are collected by the AIX trace collector, which is an AIX run-time routine. The AIX trace collector writes these entries directly to the application trace minor device (/dev/appltrace).

The trace stop command (trcstop) terminates tracing by sending software termination signals to the active trace daemons.

Trace data is formatted and displayed in a readable format with the trcrpt command. Because each record is time-stamped, the trace log file is sorted chronologically and then formatted according to the data saved by the trace point which generated it. The trace report generator is driven by an external table of trace format templates which are found in the file /etc/trcfmt. These templates describe the data layout of the trace data from each individual trace point. The template file may be modified by invoking the trcupdate command to include trace points in newly installed programs or in third-party programs. To improve readability and information content of the report, the template file also allows for substitution of meaningful mnemonics for trace points, predefined data values, etc.

Dump

The IBM RT PC system provides a system level dump capability to enhance the user's ability to do problem determination and resolution. In the IBM RT PC, a "DUMP" environment may be characterized in several ways:

- The VRM or virtual machine ceases execution.
- The VRM or virtual machine abends.

These failures may occur in an application, the base operating system, or the VRM.

When a failure occurs, the user may choose to initiate a dump. The user presses a dump key sequence: CTL-ALT-NUMPAD8 for a VRM dump. The target for a virtual machine dump is the dump minidisk (allocated at AIX install time) and the data placed on that dump minidisk is defined by UNIX System V. The target for a VRM dump is a high-capacity diskette.

The VRM dump program is permanently resident in memory. It has its own diskette device driver. It is self-contained and does not depend on any VRM resources. The dump program is intelligent; it does not dump all of real or virtual memory. The first 32K of real memory, NVRAM, tables, control blocks, page 0 of virtual machines, error log entries, etc., are dumped. Each component (other than base VRM) that resides below the VMI can identify to the dump program the location of its dump table — a table containing relevant control structure addresses of data to be placed on the dump diskette. This identification is normally made at device initialization time, but can be updated at any time. The dump program does not pick up any dynamic structures from components other than the base VRM unless the structure is defined in the component dump table.

The VRM dump formatter utility, invoked with the command dumpfmt, presents the dump information by name and hexadecimal representation with ASCII interpretation. The header information for a dump consists of a concise set of data defining the nature of the dump, such as failing module name and failure address. This header information becomes part of the customer information provided to the IBM service personnel for problem resolution. The dump formatter can be run interactively or in batch mode.

Error Log

The error log function is intended to provide a tool for problem determination of hardware and some software errors. Data specific to a problem or potential problem and certain informational data (e.g., IPL/shutdown time) is recorded on disk. When the user needs to view this data, an error report program formats the error log data in an intelligible form. The report program sorts the disk file by date and time, providing a chronology of the system's behavior. The error log function can be started by superuser, but is generally started by /etc/rc.

The user has two commands for controlling the operation of error log: start and stop. Error logging is generally started by /etc/rc by invoking the executable file /usr/lib/errdemon. This error daemon process is the focal point for gathering error records. The error log file specified in /etc/rasconf is implicitly two files with extension .0 and .1. When the .0 file is full, the .1 is written; when the .1 file is full, the .0 file is then overwritten. This allows a quasi-circular file to be kept with minimum data loss. Any data that cannot be written to the log file (e.g., disk adapter failure) is written to NVRAM. Likewise, at error daemon invocation, NVRAM is emptied and the data written to the log file. Data in NVRAM is in an abbreviated form because there are only 16 bytes available for error logging. But those 16 bytes are mapped to provide all types of error entries. Error logging can be stopped with the errstop command. It issues a software termination signal which is caught by the error daemon. Error logging is normally implicitly stopped at shutdown.

The error log function takes additional information about error logging from the RAS configuration file /etc/rasconf. This file contains configuration data for all RAS functions. The entry for error logging includes the file name where error log data is to be stored (default /usr/adm/ras/errfile) and the maximum size of the file name (default 100 blocks).

Error logging "on" means that all errors reported are recorded on a disk file. When error logging is "off," errors are kept in memory buffers but are never recorded on a disk file.

Error logging can operate at all levels of the system: below the VMI, in AIX, and at the application level.

Below the VMI, error log collection is handled by the VRM error log collector and process. The error log process is initiated by a Send_Command SVC (error log on) which sets up the error logging environment and starts the process. The VRM error log collector receives error information from the VRM and its components. The VRM error log collector notifies the VRM error log process and the VRM error log process sends that error entry to AIX to be written to the error log file. The error log process is terminated by a Send_Command SVC (error log off).

In AIX, the kernel error log device driver is the central control point for error log functions. The error log device driver controls the allocation of buffers to collect all error data and handles the reading and writing of the data in the buffers. It also issues the Send_Command SVC which initiates error logging in the VRM. The error log device driver has an interrupt handler which receives the interrupt from the VRM error log process indicating that a VRM error entry has been generated and needs to be written to disk.

AIX kernel error log entries are written to the AIX error log device (/dev/error) by the errsave macro.

For both AIX extensions and applications, error log entries are collected by the AIX error log collector which is an AIX run-time routine. The AIX error log collector writes these entries directly to the error log device (/dev/error).

Error log data is formatted and displayed in a readable format with the errpt command. Because each record is time-stamped, the error log file is sorted chronologically and then formatted according to the data saved by the component which generated it. The error log report generator is driven by an external table of error log format templates which are found in the file /etc/errfmt. These templates describe the data layout of the error log data from each individual entry. The template file can be modified by invoking the errupdate command to include classes of errors in newly installed programs or in third-party programs. To improve readability and information content of the report, the template file also allows for substitution of meaningful mnemonics for classes of errors, predefined data values, etc.

Error log entries are divided into classes (hardware, software, IPL/shutdown, general system, and user-defined). Each class is optionally divided into subclasses, and each subclass is optionally divided into masks. Because the error log file may become very large, the user can qualify what is to be included in his error report. The user can specify a time span, a combination of classes/subclasses/masks, error entries desired from a particular virtual machine, or error entries desired from a particular node. The default report is a summary report that contains a one-line entry for each error formatted. Optionally, the user can request a detailed report which includes the one-line summary plus the data associated with that entry.

For each hardware entry in the error report, an analysis of the error is appended. This specifies the probable cause, the error, what hardware pieces to suspect as bad, a list of activities the user could perform for further isolation, and a service request number. This analysis is based solely on that error entry.

Update

Updates for software products on the RT PC are packaged together on the same diskette. A new "update" command provides a menu interface to applying these updates. When an update diskette is received, the user can "apply," on a trial basis, the updates for one or more of the software products that are already installed on the system. The user can then test the updated programs to ensure that they still function correctly in that environment. If the updates have caused a regression, the user can run the update command to "reject" (back out) the update. Otherwise, the user issues the update command to "commit" the update as the new base level of the program.

Conclusion

We believe that we have successfully made AIX into an operating system that can be used without detailed knowledge of its internal structure. It takes advantage of the functions of the Virtual Resource Manager to exploit the capabilities of the RT PC hardware. It provides us with a general base on which to provide support for additional devices, applications, and communications features without massive re-coding or user inconvenience.

Acknowledgments

The porting, modification, building, testing, and documentation of AIX have involved hundreds of people. The authors of the various sections of this article wish to thank their colleagues inside and outside of IBM for their ideas, support, and just plain hard work.

References

1. Scott M. Smith, "Floating Point Accelerator," *IBM RT Personal Computer Technology,* p. 21.

Extendable High-Level AIX User Interface

Tom Murphy and Dick Verburg

Introduction

Including the UNIX kernel and command language in the AIX operating system presented us with both an advantage and a problem. The command language came complete with a wide range of functions already implemented. However, the large number of functions was a problem for the new user. The names of the functions were frequently less than mnemonic, and there was little uniformity in the invocation syntax for the various commands. Some accepted keyword parameters, some used letter codes. Some took their input from 'standard-input', others accepted an input file name as part of their invocation sequence. In short, the system was designed for a programmer familiar with the variety of commands and functions, rather than for an inexperienced or casual user.

Objectives

The objective of the Usability program was to provide an alternative interface to the operating system. This interface was to be oriented toward the user who was unfamiliar with the details of the operating system. It was to be available to users on all terminals, those attached using the async interface as well as the system console. It was, however, not to require creation of new commands to provide function already provided in the operating system by existing commands. Finally, while a particular subset of the operating system commands was defined for the initial implementation, the system was to be flexible enough to accommodate additions in the future without major rework to the programs in the Usability program.

Usability Definition

The definition of the Usability program was, naturally, heavily influenced by the choice of the user interface to be provided. The interface chosen was to be a full screen interface on any of the terminals supported by the system. The primary operator action was to be a **point-and-pick** interface in which a selection was to be made from those presented on the screen. When additional information was needed (beyond simple selection) the user was to be presented with an overlaying window prompting for the required information. In addition, extensive 'help' information was to be available to the user at most times. (More information on the design and rationale for the user interface itself is contained in the paper by Kilpatrick and Greene[1].)

Two primary applications, as seen by the user, were defined as part of the Usability package. First was an action-oriented **Tools** program. In this application the user is first presented with a choice of actions to be performed (e.g., print, copy, compile). After selecting one of the presented actions, if additional parameters or 'object' specifications are required, the user is presented with an appropriate 'pop-down' where the appropriate objects (usually files) can be specified. Second was an object-oriented **Files** program. In this application the user is first presented with a set of objects (files in the current directory). When one or more objects are selected, the user can then specify an action to be performed on the object(s).

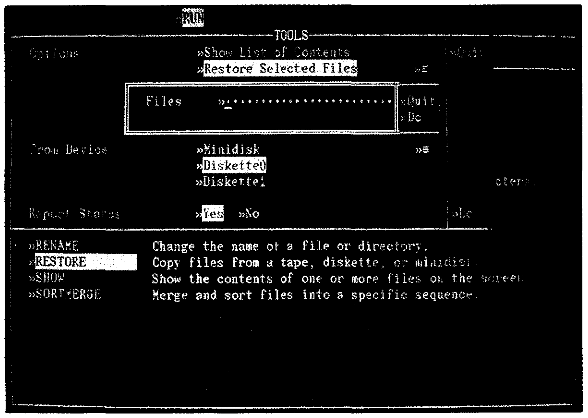
To support these (and other) applications, two additional components were defined as service packages. First, a dialog manager was defined to provide the tools needed by applications to define and display successive user-interface screens. Finally, libcur (an adaptation of an existing UNIX routine package) and terminfo were selected to provide control of the screen appearance and supply an interface that masked specific terminal device requirements.

Implementation

While the components of the Usability programs were defined and developed independently, they share several attributes that can be seen when they are examined closely.

Files Program

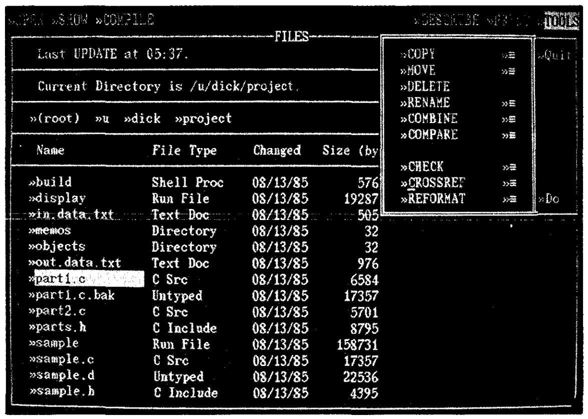
This application presents the user with a list of the files in the current directory. While there are options to limit the set of files presented, to sort the list, or to select other segments of the directory tree for display, the primary operator action is to select the file to be acted on. When a selection has been



```
                    »RUN
---------------------------TOOLS--------------------------
 Options          »Show List of Contents         »Quit
                  »Restore Selected Files   »≡
        Files     »_.......................   »Quit
                                              »Do
 From Device      »Minidisk                 »≡
                  »Diskette0
                  »Diskette1                        cters.
 Report Status    »Yes   »No                     »Do

 »RENAME      Change the name of a file or directory.
 »RESTORE     Copy files from a tape, diskette, or minidisk.
 »SHOW        Show the contents of one or more files on the screen
 »SORTMERGE   Merge and sort files into a specific sequence.
```

Figure 1 Tools application uses action-object



```
»OPEN »SHOW »COMPILE                    »DESCRIBE »PRINT »TOOLS
---------------------------FILES------------
 Last UPDATE at 05:37.                    »COPY       »≡   »Quit
                                          »MOVE       »≡
 Current Directory is /u/dick/project.    »DELETE
                                          »RENAME     »≡
 »(root)  »u  »dick  »project             »COMBINE    »≡
                                          »COMPARE    »≡
 Name          File Type    Changed   Size (by
                                          »CHECK      »≡
 »build        Shell Proc   08/13/85     576  »CROSSREF »≡
 »display      Run File     08/13/85   19287  »REFORMAT »≡   »Do
 »in.data.txt  Text Doc     08/13/85     505
 »memos        Directory    08/13/85      32
 »objects      Directory    08/13/85      32
 »out.data.txt Text Doc     08/13/85     976
 »part1.c      C Src        08/13/85    6584
 »part1.c.bak  Untyped      08/13/85   17357
 »part2.c      C Src        08/13/85    5701
 »parts.h      C Include    08/13/85    8795
 »sample       Run File     08/13/85  158731
 »sample.c     C Src        08/13/85   17357
 »sample.d     Untyped      08/13/85   22536
 »sample.h     C Include    08/13/85    4395
```

Figure 2 Files application uses object-action

made, the operator is presented with a choice of actions that apply to the chosen file. The determination of what actions are valid for a particular file is based on its file type and controlled by a file-type description that resides in a shared data area outside of the Files program.

The determination of the file type is based on the suffix that is part of the file's name. A table of information is maintained that relates the suffix to the set of actions that are valid for files. For each file type there may be a special print program, compiler, editor, interpreter, etc. For any of these entries the specification may be either empty, indicating that the option is not valid for that file type, or may contain the name of the program that provides the support for the function. For example, for most files the editor specified is *ed*, while for object programs no editor is specified.

The description of a file type is carried outside the Files program. This provides a mechanism to modify file types without modification of the Files program itself. New file types can be added in the system simply by adding a description for the new file type (an interactive program makes this addition easy). The main Files program does not require modification unless new classes of actions are added (in addition to print, edit, compile, etc).

Tools Program

This application presents the operator with lists of actions that can be invoked. The lists of actions available are grouped into sets of related actions. The first list presented is the list of available groups. After selection of a group, the commands/actions that are part of that group are presented. Selection of a particular action generally will result in a request for additional information to allow the operator to specify the object to be acted on.

The lists of commands are described in files that are stored on disk, outside the code for the application. The name of the commands or command groups, the descriptive information presented to the user, and the names of other files associated with the commands are stored in these files. With this information stored outside the application, additional commands and command groups can easily be added to the application by simply changing the files, rather than by modifying the Tools program itself.

Dialog Manager

The dialog manager provides application control and services to support the processing of interactive dialogs. Dialogs are named sets of field or record descriptors which can be presented within named screen areas. The dialog manager monitors operator input and performs conditional processing based on that input as specified by the dialog.

Dialog definitions were designed to minimize the discrete number of times an application needed to be directly involved with screen output and operator input. Dialog definitions include information that allows the dialog manager to direct the flow of control from one screen panel to another based on the operator's actions.

Selectable fields, called **buttons**, can be defined within a dialog as can the actions to be performed when a button is selected. Dialog actions include panel-to-panel transition, presentation of a new object, assignment of a value to a named variable, removal of a panel, saving and restoring the dialog state, selection or de-selection of another button, return to the application, and linking to a user exit.

User exits are application routines to be called by the dialog manager as specified in the dialog definition. These routines are usually invoked to perform application-specific tasks, for example, getting a new list of files. The dialog manager can be called recursively from within user exits.

Help text appropriate for the context of the dialog can also be referenced from within the dialog. This help text is accessed using the system help facility to provide for flexibility and translation, again without modification to the programs using the dialogs.

Data entry is supported by character and numeric fields, required entry fields, user exits, and a "blanks not allowed" option. Multi-line input fields are also supported. Default values can be constant text, named variables, or combinations of either. Operator entered data is stored into named variables as specified in the dialog definition.

Applications can create, update and delete named variables. These variables can be local to a single process or shared at the activity, user, or system level. Shared variables can be used independent of dialog applications.

Note again, that the dialog is stored in a file outside both the application and the dialog manager. The dialogs may be changed significantly without requiring any modification, compilation, or reconstruction of the application. Dialog definitions are pre-processed from a readable and editable form to one that is more efficient for run-time processing. Separation of the dialog from the application is still maintained.

Libcur and Terminfo

The libcur package of routines was adapted from a similar set of routines that existed in the UNIX system. This set of routines provides the structures and management routines to control multiple, overlapping areas on the display. Routines were provided to allow the definition of such areas, assist in managing the data that is presented in each area and support presentation on the display with the appearance of overlapping papers. In addition, these routines use the system terminfo routines, which provide access to and processing for terminal description files for each type of terminal being supported.

The content of a terminal description includes the information about a terminal needed to properly control and process that device. This includes information about the data stream that must be sent to the device for required functions (e.g., move the cursor to a selected row and column, delete a character, clear the screen). Also included is feature information (e.g., which attributes are available, what characters should be used to construct boxes). Finally, information about what control strings will be generated by the terminal in response to an operator action is described (e.g., Do key, PF1, Delete character key, etc).

Again, the terminal descriptions are outside the application, and the addition of a new terminal type to support requires only the addition of an appropriate terminal description.

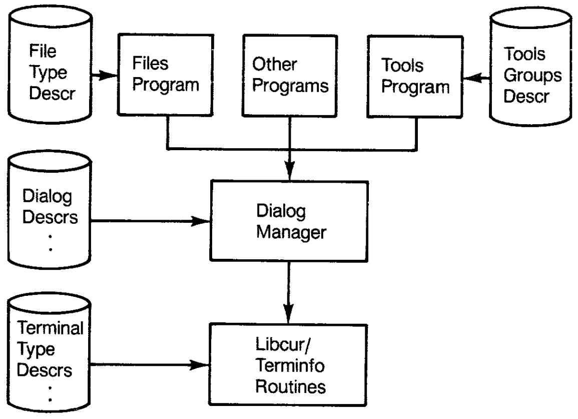


Figure 3 Software Layers

Summary

At each level of the usability package, the programs have achieved a level of flexibility primarily by moving significant amounts of control information out of the program and into external data files. Each time this is done it extracts a penalty in performance since the information must be accessed and must be interpreted. In order to minimize the penalty, each program described above has included logic that minimizes the number of times the data is extracted. Thus the penalty has been limited to a front-end cost when the applications are invoked.

The structure outlined here does, however, provide a reasonable degree of future flexibility and the expense in the form of processing performance has not proven to be excessive. Each layer has provided for future extensions by including a flexible external definition file. The capabilities of the system are thus not bound once and for all at the time the system is shipped, but can be extended easily as future needs dictate.

References

1. P.J. Kilpatrick and Carolyn Greene, "Restructuring the AIX User Interface," *IBM RT Personal Computer Technology,* p. 88.

Extended File Management for AIX

John M. Bissell

Introduction
Most major operating systems provide sophisticated file access methods, with functions that allow the definition and manipulation of records, structures, and fields. Such functions speed and simplify the application programmer's task. Since many applications require random access to records, most comprehensive access methods offer some form of indexed access to data for improved performance.

Base AIX file system calls provide only basic byte and string-oriented file manipulation functions. Applications with a need for more complex data structures and retrieval capabilities such as indexing must architect and implement their own access methods. This lengthens the product development cycle, cost, and risk. These private access methods are typically proprietary and unusable by other applications. This results in applications that cannot communicate or be integrated without some intervening file transform function. In particular, it generally makes file sharing impossible and parallel updating inevitable.

IBM RT PC Data Management Services (DMS) extends the facilities of the base AIX operating system and command interface. It builds on the AIX file system to provide both record and field-level access. The AIX directory is also expanded to provide additional information about files. The product consists of an application programming interface (API) and a set of AIX commands for manipulating both AIX and Data Management files.

Data Access Methods
A key design decision was to provide both a traditional record-level indexed sequential access method, and a higher-level field access method which allows records to be described and manipulated at the field level. Both types of files use the same underlying data and index structures, but are not totally compatible due to the control information that precedes the field data.

Record Access
Record access allows data structures of fixed or varying length to be stored and retrieved. This is the traditional method of file access for languages such as PL/I, FORTRAN, and COBOL. It also lends itself very well to applications development in C, where data structures are easily defined and manipulated.

Records may be retrieved sequentially, in the order in which they were added to the file. They may also be retrieved directly by supplying the relative byte address (RBA) of the record within the file. By storing the RBA of one record in another, complex data structures such as hierarchies and networks can be built. Indexed sequential retrieval is accomplished using the index techniques described later.

Field Access
Field access permits an application to define the contents of a file to the field level. Such a file is a "table," where the rows are records and the columns are fields.

The column is the basic structural entity in a table. The creator of the table describes the characteristics of the columns of the table. The description must include a user name for each column and typing information. Column definitions are stored in a specially named index (SYSCOLUMNS) in the index file.

A row represents a record in a table. A row contains one instance of data for all of the columns in the table. Field access functions are used to retrieve, insert, and update individual fields within rows.

The DMS field access support provides for row selection based on complex selection criteria, including wild cards. These functions provide many of the capabilities of a relational data base on a single table. Field access functions automatically optimize query access using indexes where available.

Indexes
Indexed access for record and field functions is provided using B-tree techniques [1]. Integer, short integer, double precision floating point, and fixed and varying length character data are supported as key types. Up to 16 key parts may be defined for each

index, with an ascending or descending specification on each part. An index may be defined as unique or duplicate. Architecturally, any number of indexes may be defined on a file, although a practical limit may be set by the maximum size of the index file. Each index is given a user-defined name which is then used to refer to the index in subsequent functions. Indexes are automatically updated when insertions, modifications, or deletions are performed.

For record files, each key field is identified by the starting byte displacement of the part within the record, the type of data, and length of the data. For tables, the names of the column(s) that comprise the key are used.

Key compression and prefix B-tree techniques [2] are employed to increase the number of keys that can be contained in each index block. The index block size is always chosen to be a multiple of the file system block size (2K) to provide for efficient access. Index nodes are buffered to reduce I/O. An innovative concept called the "level table" is used to further minimize the tree traversals. Leaf nodes are chained to provide next and previous sequential retrieval.

File Architecture
Each Data Management file or table consists of a data file and an optional index file. Both files are standard AIX files. The decision to use two AIX files instead of storing data and indexes in the same file has several advantages:

- Each file is smaller, which has some performance advantage in AIX.

- Decisions on file strategy (such as whether to map or not) can be made independently for indexes and data.

- I/O errors in the index file can be recovered by rebuilding the indexes from the data file.

Data Management files are specified by using an AIX file system path name. This name properly refers to an AIX path down through all but the last name. Data Management uses the path name through AIX to locate the AIX directory containing the file(s). The files are always manipulated together as a data set.

Data File Architecture
Each data file is defined to contain either fixed or variable length records. Records are located by their RBA within the file. Each record has a 4-byte header containing an ID, consistency counter, and a record length. Following the data is a 1-byte ID and matching consistency counter. When a record is written out, its consistency counters are incremented. Each time a record is retrieved, its header and trailer consistency counters are compared to ensure that all of the record was written and read correctly.

Free space management is done to reclaim deleted space. Free areas are chained by relative size. A request to add a new record always finds a free area within one access, or else the file is extended.

The RBA of a record never changes, even when the record size is changed. (If a record outgrows its slot, it is moved and a pointer is left behind.) This simplifies logic, since indexes do not have to be updated unless the key of a modified record is changed.

Index File Architecture
All of the indexes defined on a particular file are stored in a single index file. The file is managed as a "tree of trees" with the names of the indexes being stored in a B-tree at the beginning of the index file. This allows the file to contain an unlimited number of indexes. Each index block is identified by the RBA of its root node, which is kept constant.

Another design decision was to use the data file algorithms for storing index nodes. Each index node is stored as a record in a data file. The records are chosen to be a multiple of the file system block size, minus the space required by the record header and trailer. Relative byte address pointers link the nodes of each index to form the characteristic B-tree. This scheme allowed use of already available code for free space management and error detection.

Extended Catalogs
The AIX inode structure captures only very basic information about a file, such as creation date, date of last modification, etc. The only descriptive information that is supplied by the user is the file name. No provision is made for storing user-defined attributes, such as file type, profile, or even a comment as to the contents of the file.

Data Management Services provides the ability to create and manipulate an extended catalog for such information. API functions and several utilities are provided. The catalog structure for Data Management files and tables is an extension of the AIX file system directory structure. The user views Data Management files as being handled the same as AIX files.

While extended catalog information is provided implicitly for Data Management files and tables when they are created, normal AIX files and directories do not have any associated extended information kept on them unless a specific request to catalog such objects has been made through the API functions or the "describe" utility. However, API functions that read data from the catalog will supply the standard AIX status information for the non-cataloged files.

A catalog file is given the special name ".SYSCATALOG". This file is created with the same uid, gid, and mode as the AIX directory in which it resides. Thus, if a user has write authority on a directory, he also has write authority to add/update/delete entries in .SYSCATALOG. The file structure in an AIX directory containing Data Management files is shown in Figure 1.

Utility Aspects of System
Integration of Data Management files and ordinary AIX files is accomplished from a user perspective by the utility commands. Since Data Management files and tables can consist of two AIX files, the standard AIX utilities such as cp, mv, and ls do not understand the relationship and special characteristics of these files. For this reason, several AIX utilities have been supplemented with new Data Management utilities. A user need only learn one set of commands for both types of files, and in fact does not even need to be aware of the difference. The integration is further carried out by having the new command names aliased to existing AIX utilities when Data Management is not installed. The new command names are also used by the Usability program [3] to further reinforce their acceptance by users.

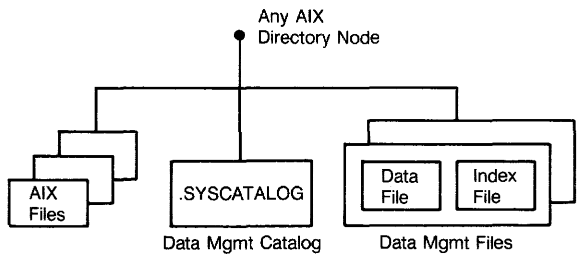


Figure 1 AIX Directory Node with Data Management Services Files

These new utilities understand both ordinary AIX files and Data Management files and tables. They also provide a means to query and update the extended catalog. The utilities give the user a consistent view of files, whether they are AIX, Data Management, or those created by the SQL/RT Data Base LPP. Although the standard AIX commands are still available, their functions are completely subsumed by the corresponding DMS utility commands.

The supplied utilities have the following functions:

- copy — replaces the AIX cp command for copying files. Copy is designed to detect file "holes" (blocks of all zeros) and avoid physically copying them. This prevents AIX from actually allocating any physical space for the block, thus conserving disk. The block materializes as zeros when read.
- move — replaces the AIX mv command for renaming files.
- delete — replaces the AIX rm command for removing files.
- list — is an expanded version of the AIX ls command. List provides information from the extended catalog for cataloged files. If the file is not cataloged, inode information is printed in the manner of ls.
- describe — a utility that not only displays detailed information on a file (including indexes and columns for DMS files and tables), but also allows the user to do the chown and chmod AIX functions. In addition, a brief comment may be entered for the file. The comment is stored in the extended catalog and displayed by the list utility.
- archive/retrieve — used to backup and restore all types of files to removable media. These commands maintain the extended catalog information intact.
- create — allows creation of AIX, record, and field access files.
- recover — used only for Data Management files that have been determined to contain corrupted data, possibly from a system crash.
- condense — used to compress a Data Management file by removing imbedded free space.

Process Model
One of the major design decisions in developing Data Management Services involved the selection of a process model. The two alternatives examined were a subroutine library of Data Management functions, and a two-task interface. The decision was made to design a two-task interface, as shown in Figure 2.

Each application process is supported by its own DMS process. Communication between the application and DMS is via the application shared segment. Control is transferred using semaphores. The reasons for choosing this design over the standard function library approach were:

- AIX provides new facilities for interprocess communication which greatly enhance the ability for cooperating processes to share data.

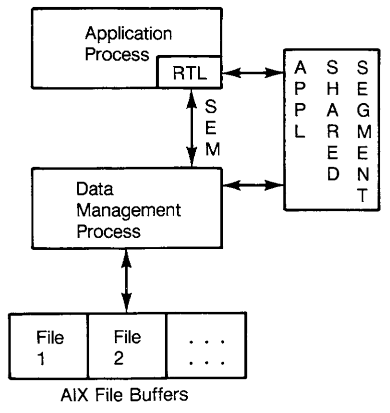


Figure 2 Process Model

- The size of executable applications on disk is reduced, since the amount of linked-in code is small.

- Memory usage is reduced, since AIX shares the text segment of the DMS process among all concurrently executing applications.

- Data integrity is enhanced. An abnormally-terminated application does not take down the Data Management process. Signals, such as software termination or shutdown, can be caught and processed without leaving data in an inconsistent state. Also, since the address spaces of the application and Data Management processes are different, application bugs are much less likely to corrupt DMS data and control structures.

- The model allows for such interesting enhancements as asynchronous I/O, since the application could be allowed to continue processing while waiting for Data Management to store or retrieve the requested data.

An application program containing DMS API functions is compiled and then linked to a library containing the run-time code for the Data Management function calls. Each library function is small, and serves only to communicate between the application and the DMS process. The first time one of these functions is invoked during execution, a Data Management server process is started using the fork/exec mechanism. A private shared segment and a set of two semaphores are obtained. The DMS process waits on its semaphore.

Subsequent interaction between the application and Data Management processes is as follows:

1. The run-time library moves the application-supplied parameters and data buffers from the application address space (stack and data segments) to designated locations in shared memory.

2. The run-time library routine then posts the DMS process by setting one of the semaphores. It then waits on the other semaphore.

3. The DMS process is awakened when its semaphore is set. It looks in shared memory to determine the function to be performed, and calls the appropriate internal routines to process the request. The input data is addressed directly from shared memory, and is not moved.

4. Upon completion of the request, a return code is placed in shared memory, and the application is posted by setting its semaphore. The DMS process then waits for another request.

5. The application is enabled, and the return code and any returned data is moved from shared memory to the application address space. The run-time routine then returns to the application.

The DMS process resets a timer each time the application is posted. If no further request is received within several seconds, the DMS process checks to ensure that its application parent is still running. If it is not, the DMS process commits all updates in process and closes all files. It then removes the shared memory and semaphores and exits.

There is no explicit communication between DMS processes servicing different applications. All communication is done through data contained within the files themselves. Locking using the lockf system call is used to control concurrent requests for data.

Use of Extended AIX Features
Several of the extensions to the base AIX system have been utilized by Data Management Services. These features are discussed further by Loucks[4].

- lockf — this operating system call is used to provide file and record locking.

- fsync — this operating system call causes all updated blocks for a specific file to be forced to disk. Data Management Services uses this function to provide commit processing.

- fclear — this operating system call is used by Data Management when records are deleted, to return blocks of zeros to AIX for reallocation. This feature saves disk space when files have many deleted records.

- ftruncate — this operating system call is used by Data Management to truncate the file size when records at the end of the file are deleted, saving media space.

- mapped files — reads and writes data via the paging hardware and software instead of through AIX file buffers. System calls with the attendant high overhead of the context switch are bypassed by using the mapped file feature to directly access the file as if it were a part of the DMS process's address space.

Conclusions

RT PC Data Management Services enhances the capability and application development environment provided by the AIX file system. The supplied API routines allow the definition and manipulation of data at either the record or field level. Integration with AIX is achieved through a set of utilities that operate on both AIX and Data Management files.

The use of Data Management Services frees the application programmer from the time-consuming task of defining and implementing a proprietary access method. In addition, the application benefits from future function and performance enhancements that may be made to DMS. Extended AIX features are exploited to provide improved performance, media utilization, and data integrity. The resulting Data Management Services files can be shared by multiple applications.

The AIX file system is also available for new or ported applications. An application's choice of storing data will depend on the need for functional services: for simple structures, the AIX file is all that may be needed; for other, more complex structures, the Data Management interface is more appropriate.

Acknowledgments

The author wishes to thank the past and present members of the Data Management development team for their support and perseverance through the many requirements, design, and code changes which inevitably result during an effort of this magnitude. The final product reflects the very best of their talents and abilities.

I am indebted to Linda Elliott and Cheng-Fong Shih for their assistance in architecting and designing the file structures and interfaces. The other members of the development team, comprised of Jeanne Smith, Thuha Nguyen, Karen Depwe, Pierre Fricke, Albert (S.Y.) Chang, and Richard McCue, ensured that the design was implemented efficiently, and offered many suggestions which manifest themselves in the quality of this LPP. The development effort would have amounted to little without the diligent testing of Wayne Wheeler. Many thanks also to Harry Sundberg for the management direction, planning, and insight so necessary for a successful development project. The origins of the Data Management Services LPP trace back to early product definition work done with Jack Olson.

References

1. Douglas Comer, "The Ubiquitous B-Tree," *ACM Computing Surveys,* Vol. 11, No. 2, June 1979.
2. R. Bayer and K. Unterauer, "Prefix B-Trees", *ACM Transactions on Database Systems,* Vol. 2, No. 1, March 1977.
3. Tom Murphy and Dick Verburg, "Extendable High-Level AIX User Interface," *IBM RT Personal Computer Technology,* p. 110.
4. Larry Loucks, "IBM RT PC AIX Kernel — Modifications and Extensions, *IBM RT Personal Computer Technology,* p. 96.

The Virtual Resource Manager

Thomas G. Lang, Mark S. Greenberg, and Charles H. Sauer

Introduction
The Virtual Resource Manager, or VRM, is a software package that provides a high-level operating system environment. The VRM was designed to build upon a hardware base consisting of a Reduced Instruction Set Computer (RISC) and a PC AT compatible I/O channel, although it is not limited to this environment.[1] In fact, the VRM can be easily extended to support different I/O hardware. An example of this is the VRM's support of the IBM 5080 graphics hardware, which is designed to an IBM System/370 channel interface.

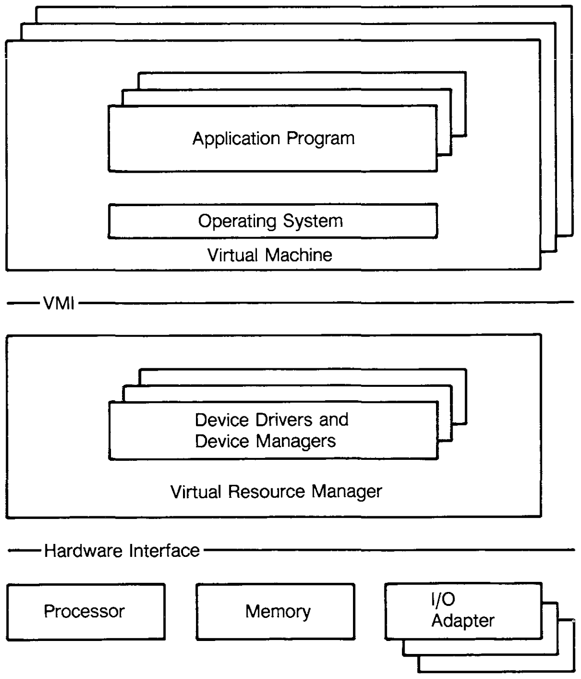


Figure 1 RT PC Software Design

The concept of RISC architecture is the minimization of function in hardware, providing only a limited set of primitives.[2] This allows the processor to be designed with simplified logic and a corresponding increase in the speed of its instruction set. In this environment, the software must provide function that traditionally is provided in hardware, such as integer multiply and divide functions and character string manipulation. The VRM builds on this hardware base to:

- Provide a high-level machine interface, which simplifies the development and implementation of operating systems and their applications.

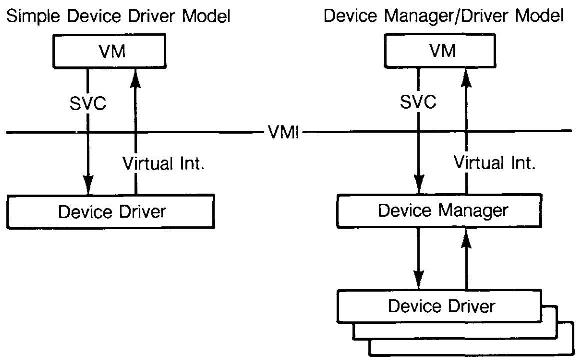


Figure 2 Virtual Device Models

- Maximize performance to support real-time process control type applications.

- Allow users to easily customize the system to meet their needs by providing an extendable, flexible interface.

- Provide compatibility with IBM-PC applications by supporting an Intel 80286 coprocessor.

The approach used to accomplish these goals was to design a Virtual Machine Interface, or VMI, with a set of functions to facilitate the use of a variety of operating systems. The VMI has features that support concurrency of multiple operating systems and applications, while insulating them from most details of the implementation of the hardware, except for the problem state instruction set. Also, the VMI allows operating system programmers to install extensions to the VRM to support additional I/O devices, or even to replace the IBM-supplied I/O subsystems.

Traditionally, virtual machine implementations have suffered in performance due to the overhead of simulating hardware function. The key to maximizing the performance of the VRM is that the vast majority of instructions issued by the operating systems and applications are directly executed by the hardware. The VRM software is invoked mainly to handle I/O operations at a relatively high functional level.

Fundamental to the design of the RT PC is that the VRM is the underlying support layer for an operating system. In particular, the UNIX kernel[6] was chosen as the principal operating environment supported by the RT PC product, and the design of the VRM was influenced by this selection.

The concept of a virtual machine has been implemented on IBM mainframe computers with a software product known as VM/370.[3] The VRM is similar to VM/370 in that it supports the concurrent execution of multiple operating systems. However, there is a significant difference. VM/370 provides a complete functional simulation of the real System/370 hardware, such that an operating system built for the real hardware, like MVS, can run in a virtual machine. The Virtual Machine Interface supported by the VRM provides considerably more function than the RT PC hardware; an operating system implemented to the VMI will not run on the real hardware. The design of the VMI traded off complete hardware compatibility for the benefits of a high-level, high-function machine definition.

Along with the concept of concurrent virtual machines, the VRM supports virtual memory.[4] The hardware memory management capabilities include a 24-bit address space for real memory (i.e., the ability to address up to 16 megabytes of real memory) and a 40-bit address space for virtual memory (1024 gigabytes, or one terabyte).[5] The virtual address space is comprised of 4096 segments of 256 megabytes each. Sixteen segment registers are provided by the hardware, and one of them is permanently dedicated to addressing I/O devices. Thus, up to 15 segments can be accessed simultaneously. The VRM software takes advantage of these features to logically separate the address spaces of the virtual machines from each other and from the VRM address space.

Another VRM feature, related to virtual memory, is "mapped file" support. Mapped files are a relation of logical disk blocks to virtual memory addresses, such that a disk file can be read from or written to simply by reading from or writing to its associated memory addresses. Explicit disk reading and writing is not required.

The AIX operating system contains a complete file system, based on explicit disk reading and writing. When modifying AIX for the VMI, it was desirable to salvage as much software as possible. Also, the concept of mapping files does not work well with removable media, such as tapes or diskettes. So, mapped file support is augmented with a minidisk manager in the VRM, providing more conventional file access support.

The minidisk manager provides access to disks partitioned into separate spaces, or minidisks. In turn, the minidisks are partitioned into logical blocks whose size is determined by the operating system, independent of the characteristics of the physical disk. The minidisk manager also includes functions not normally found in simple hardware access methods, such as error recovery and bad block relocation. Further, the VMI for the minidisk manager allows the potential for "remote" minidisks, accessed across a communication link such as a high speed local area network.

The "virtual resource" concept is also applied by the VRM to I/O devices, such as virtual terminals.[7] The VMI includes a high-level interface to I/O devices that is consistent for all devices. Also, the VMI includes provisions for bypassing the VRM and accessing devices directly. The preferred method of using a device from a virtual machine is to take advantage of the I/O support functions supplied by the VRM. But, there are graphics applications, for example, which can gain enough performance by writing directly to a display device to offset the loss of flexibility suffered when bypassing the VRM services. Another reason for allowing direct access to I/O devices was compatibility with existing applications; for example, a BASIC language program written using the PEEK and POKE functions to access an I/O device.

Extendable Virtual Machine Architecture
Another feature that distinguishes the VRM is the extendability of the architecture. Users of microcomputers have become accustomed to plugging new devices into a machine's I/O channel. However, getting the machine's software to use the device usually requires some ingenuity. One approach is to design the new device such that it "looks like" an existing device, so that the existing software can recognize and use it. Another approach is to run an application program that drives the device directly, independent of the existing operating system. For example, a program could communicate with a device by sending commands to its I/O port, then using a software "spin loop" to poll its status port to determine when the commands complete. The former approach limits the flexibility of the new device, while the latter destroys the effectiveness of a multiprogramming operating system by tying up the processor during I/O operations.

The VRM allows a new approach, whereby software for a new device can be fully integrated into the existing operating system. Further, the reconfiguration of the VRM to add or replace software can be performed in real time without disrupting the normal function of the machine.

A data structure, known as a Define Device Structure, or DDS, is included in the VMI so that a programmer can describe the attributes of a new device and its related software support to the VRM. Information in the DDS includes the I/O port address(es) used by the device, which channel interrupt level it uses, which DMA channel it uses (if any), whether it has any resident RAM or ROM, etc. Also, the DDS indicates which program module should be called to process such functions as:

- Device initialization
- Interrupt handling
- I/O initiation
- Timeout or exception handling
- Device termination

Using information from the DDS, the VRM is able to determine which user-installed program to call to handle an interrupt generated by an installed device. The additional software required to support a new device is added in real time, in contrast to existing systems that require the use of an off-line or stand-alone program to reconfigure the system.

To use devices, the VMI contains a set of functions including:

Define Code	Install software into the VRM, or delete installed software.
Define Device	Install a DDS into the VRM, or delete an installed DDS.
Attach	Reserve a device and allocate any resources its software may require.
Detach	Undo the function of "Attach."
Send Command	Send a command to a device.
Start I/O	A variation on "Send Command," which allows a set of commands, or buffers, to be sent to a device.

To use a device, a logical connection ("path") is established between the user and the device. The Attach function is used to establish a path, and a path identifier token is returned to the user. Subsequently, the path identifier is used to send requests to the device. When the device completes the request, it returns status information or an interrupt to the user, using the path identifier to route the data.

The VMI defines two ways to send requests to a device, the Send Command and Start I/O functions. Parameters for these functions include the path identifier in addition to device specific parameters such as a request code and buffer pointer. The difference between the two functions is that the latter passes its parameters in a data structure, know as a Channel Control Block, or CCB, which allows the specification of a chain of commands or buffer pointers. This can be useful, for example, when using a device that supports "scatter/gather" functions. During a read request data can be input from a device and "scattered" into different memory buffers. Or, during a write request data can be "gathered" from different buffers and output to a device.

Another parameter for the two request functions is an operation option that determines if the request is to be processed by the VRM synchronously or asynchronously. Implicitly, this also determines how completion status is returned to the virtual machine. For synchronous requests, completion status is supplied as a return code from the requested function, while the completion of an asynchronous request is indicated by a "virtual interrupt". The VMI defines nine interrupt levels for a virtual machine, which allows the assignment of relative priorities to interrupting conditions. When not processing an interrupt, the virtual machine is considered to be on level 7. Seven levels can be assigned to interrupting I/O devices. In order of decreasing priority, they are levels 0 through 6. In addition, there are two other levels. The machine communications level is used for messages between the VRM and the virtual machine. The highest priority level is the program check level, which is used by the VRM to report exception or error conditions to the virtual machine. The return code from a synchronous request provides 32 bits of status, while up to 20 bytes of status can be supplied with each virtual interrupt.

Two types of programs can be installed into the VRM: device drivers and device managers. A device driver is a collection of subroutines that support a specific hardware device. The VRM synchronously calls the subroutines to handle device-specific functions, such as handling interrupts and time-out conditions, and processing I/O commands from virtual machines. The VRM device driver support is intended to be sufficient for implementing relatively simple devices, such as printers, diskette drivers, and tape drives.

Device managers provide an additional level of support for more sophisticated devices, such as virtual terminals or communications subsystems (see Figure 2). These types of device subsystems typically have requirements to handle multiple asynchronous events and to manage different types of resources. For example, the Virtual Terminal Manager coordinates the activities of device drivers for the keyboard, display, speaker, and locator to simulate a higher level device known as a "terminal."

Allocation of System Resources

Resources in the VRM are categorized as serially reusable or shared. Serially reusable resources are those that can be used by different applications, but only by one at a time. For example, multiple applications may use the printer but one application must finish before the next takes over. Otherwise, the result would be scrambled printer output. Shared resources, though, may be used "simultaneously." Examples include the disks and memory, which are shared by dividing them into logical pieces (minidisks and segments), and the processor and communication lines, which are shared on a time basis.

The VRM manages several shared devices, most notably the keyboard, locator, speaker, display, and hard files. Virtual machines can have many logical terminals. The user controls which logical terminal is associated with the physical hardware via a set of reserved key sequences. Virtual terminal input is routed by the VRM to the owner of the screen that has been selected for display by the user. Output to virtual terminals is updated in memory if that display is not selected.

Support of the PC AT coprocessor presented some interesting challenges for resource management.[9] The main constraint was that the VRM had to be transparent to the applications using the coprocessor. A considerable amount of hardware support is dedicated to this purpose, in the form of "trap" logic that monitors access by the coprocessor of I/O addresses.[8] For nonshared devices, the VRM reserves the device for exclusive use by the coprocessor. I/O operations using devices of this type proceed with no further intervention required by the VRM. When using shared devices, however, the VRM must intercept each I/O operation requested by the coprocessor and simulate the function as if it were dedicated to the coprocessor. For example, when the coprocessor writes data to what it thinks is the display screen, the VRM saves this data in a memory buffer. And, when the coprocessor's virtual terminal becomes the "active" terminal, the data is moved to the actual display buffer. Also, at this time keystrokes are routed to the coprocessor when it accesses what it thinks is the keyboard adapter's I/O port. Notice that since the coprocessor accesses nonshared devices directly, they perform at precisely the same speed as they do in a PC AT. However, shared devices suffer some performance penalty since functions must be simulated by the VRM software.

Another resource that can be shared with the coprocessor is memory. The VRM can reserve some of its own memory for use by the coprocessor. In this mode, memory translation hardware detects memory references by the coprocessor and routes them to the VRM's memory. Alternatively, a memory card can be plugged into the I/O channel, and coprocessor memory references will be directed to it. This allows a great deal of flexibility to trade off the lower cost of shared memory against the higher performance of dedicated memory. The trade off is not "all or nothing." For example, a 1-megabyte address space can be provided for the coprocessor using a 512-kilobyte memory card and sharing 512 K of system memory.

Virtual memory is utilized by the VRM to eliminate arbitrary restrictions on resource usage. It is not uncommon for operating systems to restrict the number of processes in the system or the number of devices that are supported. The VRM defines internal control block areas in virtual memory that are large enough to support thousands of processes and device drivers. Thus, limitations are a function of the amount of real memory, disk space, and I/O channel slots available on a particular machine.

Designs for Real-Time Performance

The VRM was customized for the real-time processing environment, compensating for the shortcomings of the kernel in this area. Features of this design include:

- Low overhead creation and deletion of new processes and interrupt handlers
- Efficient interprocess communication
- Preemptable processes and interrupt handlers, to minimize interrupt latency time
- Prioritized scheduling of processes and interrupt handlers

- Interval timer support with 1-millisecond granularity.

Multi-programming is implemented in the VRM by dividing work into logical units, or "processes," which are scheduled by priority. In addition, the VRM contains "interrupt handlers," which are invoked in response to interrupt signals from hardware devices. In "Extendable Virtual Machine Architecture" on page 120, programs in the VRM were characterized as device managers or device drivers. Device managers, and virtual machines, are represented as processes in the VRM, while interrupt handlers are among the subroutines that comprise a device driver.

Processes and interrupt handlers can communicate using shared memory, or by using the VRM's interprocess communication functions, which include queues (for message passing) and semaphores (for serialization and synchronization).

Particular emphasis was placed on supporting high-speed devices, with stringent latency time requirements. Hardware interrupt processing is the highest priority work in the system. Interrupts from devices are further divided into four priority classes, such that the servicing of an interrupt can be preempted by a higher priority interrupt.

Also, an "off-level" interrupt handler capability is available that allows a device interrupt handler to process time-critical operations without being preempted, and to defer less critical processing to a lower priority level that can be preempted by other device interrupts.

After all pending interrupts are handled, the VRM selects the next process to execute based on 16 priority levels. The selected process will remain executing until it "waits" for some condition (such as the completion of an I/O operation or the arrival of a new work request), or until it is interrupted. Among processes with the same priority, "time slicing" is implemented; that is, if a process does not relinquish control after a period of time, the VRM will suspend it and pass control to another process. The default time slice interval is 16 milliseconds, and this value may be increased in increments of 16 milliseconds. If a sufficiently large increment is selected, time slicing is effectively disabled.

The design of the data structures for multiprogramming was influenced by performance considerations. The processor has a large number of registers (16 system registers, 16 segment registers, and 16 general-purpose registers), which makes context switching between applications a lengthy job. In a typical operating system, when an interrupt occurs, the state of the interrupted program is saved in a known location, then transferred to a control block associated with the interrupted program if it becomes necessary to switch control of the processor to a different program. In the VRM, this would require moving a large amount of data, so the interrupt handlers are set up such that the state of an interrupted program is saved directly into its control block. This contributes to faster context switching.

Another aspect of the control block design that contributes to fast context switching is that the "dispatcher," which selects which program next gets control of the processor, never has to search through queues of control blocks. The control blocks for programs that are ready to execute are always kept sorted by priority, thus only about 1% of the total time required for a context switch is required to select the next program. The remaining time is spent saving the state of the current program and restoring the state of the next program.

The VRM was designed using top-down structured programming techniques. The program code was written first in a high-level language, using primarily PL.8 (an internal IBM development language, derived from PL/I).[10, 11, 12] After the system was functioning to the point where meaningful applications could be implemented and run, the performance of the system was measured in detail. The performance data was used to determine critical paths in the software, or "bottlenecks." These parts of the system were then tuned to maximize performance. The first step in tuning was to attempt to make the PL.8 code more efficient. In many cases, this tuning turned out to be sufficient to meet performance objectives. However, some critical paths required recoding in assembler language to achieve desired performance.

The process of tuning the system was an iterative one for the measurement and recoding steps. For example, one performance objective was that the disk device driver be able to handle a disk formatted with a 2:1 interleave factor without missing revolutions, with enough of a margin to allow for an interrupt from an Async communications adapter during the critical path. A factor that increased the difficulty in meeting this objective was the disk hardware, which does not support DMA for transferring data between the adapter and memory. The disk hardware, chosen mainly on cost and compatibility considerations, is similar to the PC AT disk hardware. Using that hardware, the PC AT supports a 3:1 interleave.

In pursuit of the 2:1 objective, the VRM interrupt handling logic and disk device driver were measured and recoded numerous times, each time squeezing out a few more microseconds from the path length, until the objective was met. At several stages in the process, software ingenuity was required to surmount hardware timing limits. Some of these software "tricks" included:

- Sorting the queue of disk requests according to sector/track number, influenced by the current position of the disk arm

- Looking ahead in the queue when one request completed, to anticipate the requirements of subsequent requests

- Sending the next command to the disk adapter before processing of the current command is complete

- Using a table look-up algorithm to determine how long a "seek" operation should take, based on current and future arm position. then setting a timer to wake up the disk driver just prior to the operation completing

- Taking full advantage of the overlapped load, store, and branch capabilities of the pipelined processor.

In this extreme example, the large tuning effort paid off when a difficult objective was met. Fortunately, most other tuning problems were easier to solve. Also, there were "spin-off" benefits gained in the disk driver tuning. The path length reductions in the VRM common interrupt handling logic benefitted all device drivers, and some of the techniques used in the disk driver were applied to other device drivers. In particular, the overlapped processing of queued requests increases the throughput of all devices.

Critical to the job of performance tuning was accurate measurement of the system. Three different techniques were used. First, selected operations were executed repetitively, so that elapsed time could be measured. The measurement device was a stop watch, so to eliminate reaction-time errors and to increase accuracy, the repetition factors were chosen to be very large (e.g., thousands or even millions of iterations). Some of these "bench mark" loops were internally developed, while others were selected from bench marks published in trade journals. The latter type of bench mark was especially useful when comparing performance of competing systems.

The second type of measurement was done by inserting "hooks" into critical paths. These hooks consist of I/O instructions that output data to reserved channel addresses. To obtain measurements, a special I/O adapter is plugged into the channel to monitor the output from the hooks. The data collected by this adapter is saved on a tape. Afterwards, the tape is input to a data reduction program that generates a path flow analysis with timings. This technique allows very sophisticated path analysis, but suffers the drawback that the hooks themselves take a small amount of time to execute. Although the hook execution time is relatively small, the cumulative times can, in some cases, add up to a significant amount. Also, as the interval between hooks decreases, the hook's execution time becomes proportionately more significant.

The third technique involved a logic analyzer to monitor the output of signals from the processor chip. Using the analyzer, it is possible to measure precisely the time it takes to execute individual instructions or sequences of instructions. This is impractical for measuring large programs, but is well suited for analyzing small sections of program code that are executed very frequently. For example, program context switching and interrupt handling functions execute hundreds of times per second. In these critical paths, a few microseconds can be significant.

A great deal of performance tuning effort was spent maximizing the "pipeline" effects of the ROMP processor. The pipeline effects result from the processor's ability to overlap various stages of instruction execution. Two different situations illustrate these effects. First, if the next instruction(s) after a memory load instruction do not use the value being loaded from memory, they may be executed in parallel with the memory access. By properly interleaving instructions, this effect can be exploited to reduce the total execution time of a sequence of instructions. Second, when a branch instruction is executed, the processor must reload its instruction pre-fetch buffer with the new instruction stream. By using the processor's Branch-with-Execute instructions, it is possible to overlap the execution of one instruction with the pre-fetch buffer reload time.

The high level language compilers for the RT PC, in particular the PL.8 compiler, are designed to take advantage of the pipeline effects of the processor. For assembler programmers, the pipeline effects can be utilized, although usually at the cost of cleanly structured programming. For the tightly optimized critical paths in the VRM this has been done, but the programming effort required, contrasted with the high efficiency of the compilers, has resulted in the majority of the VRM being implemented in high-level language.

Conclusions

The VRM builds upon the low level RT PC hardware interface to provide a high-function system environment. It brings to a desk-top microcomputer many features that formerly were found only on much larger, more expensive systems, such as virtual memory and virtual I/O subsystems. It also includes features, such as dynamic reconfiguration and an extendable architecture, which are unique; and it allows for the migration of existing UNIX and IBM PC based applications to a new architecture.

During the past several years of development, the RT PC hardware underwent several major changes, but the Virtual Machine Interface has remained relatively stable throughout this time, thus minimizing the impact of the hardware changes to the implementation of AIX and its applications.

The VRM's functions complement the hardware instruction set, providing features such as virtual memory, virtual devices, minidisks, and multi-programming. This creates an environment for implementing operating system extensions and hardware device support that has the flexibility to evolve as the hardware technology evolves without forcing radical changes to existing software.

Acknowledgments

The authors would like to acknowledge the efforts of the people who contributed to the development of the VRM. The virtual terminal software was developed by Lynn Rowell's department. The RAS and Install were developed by Hira Advani's department. The VRM device drivers were developed by Mark Wieland's department. Special thanks go to Joe Corso and each member of his department thoughout the VRM development for the VRM design, the testing methodology, and the technical leadership for integration of the product.

References

1. George Radin, "The 801 Minicomputer," *IBM Journal of Research and Development,* 27, pp. 237-246, May 1983.
2. D.A. Patterson and C.H. Sequin, "RISC: A Reduced Instruction Set Computer," *Proc. 8th Annual Symposium on Computer Architecture,* May 1981.
3. R.A. Meyer and L.H. Seawright, "A Virtual Machine Time-sharing System," *IBM Journal of Research and Development,* Volume 9 Number 3, 1970.
4. J.C. O'Quin, J.T. O'Quin, Mark D. Rogers, T.A. Smith, "Design of the IBM RT PC Virtual Memory Manager," *IBM RT Personal Computer Technology,* p. 126.
5. P.D. Hester, Richard O. Simpson, Albert Chang "IBM RT PC ROMP and Memory Management Unit Architecture," *IBM RT Personal Computer Technology,* p. 48.
6. Larry Loucks, "IBM RT PC AIX Kernel — Modifications and Extensions *IBM RT Personal Computer Technology,* p. 96.
7. D.C. Baker, G.A. Flurry, and K.D. Nguyen, "Implementation of a Virtual Terminal Subsystem," *IBM RT Personal Computer Technology,* p. 134.
8. John W. Irwin, "Use of a Coprocessor for Emulating the PC AT," *IBM RT Personal Computer Technology,* p. 137.
9. Rajan Krishnamurty and Terry Mothersole, "Coprocessor Software Support," *IBM RT Personal Computer Technology,* p. 142.
10. M. Auslander, et al., "An Overview of the PL.8 Compiler," ACM, 0-89791-074-5/82/006/0022.
11. Alan MacKay and Ahmed Chibib, "Software Development Tools for ROMP," *IBM RT Personal Computer Technology,* p. 72.
12. M.E. Hopkins, "Compiling for the RT PC ROMP," *IBM RT Personal Computer Technology,* p. 76.

Design of the IBM RT PC Virtual Memory Manager

J.C. O'Quin, J.T. O'Quin, Mark D. Rogers, T.A. Smith

Design of the RT PC Virtual Memory Manager

Support of a large virtual address space was the main objective that guided the design of the Virtual Memory Manager (VMM). The VMM should take advantage of the large address space provided by the Memory Management Unit (MMU) and the abilities inherent in the MMU's inverted page table. Furthermore, the VMM needed to be designed to minimize the overhead associated with performing disk I/O. At the same time, the VMM should not adversely impact system performance, especially in situations where sufficient real memory exists to perform the desired function.

Each instruction executed by the ROMP has direct addressability to a 32-bit address space. This is referred to as the effective address space. The effective address space is divided up into 16 equal size segments by the MMU. The MMU converts an effective address into a virtual address by concatenating the 12-bit segment identifier associated with the selected segment onto the 28-bit segment offset in the effective address. This results in a 40-bit virtual address space. Each segment in the virtual address space is further divided up into 2048-byte pages. This division of the virtual address space into segments and then into pages is known as paged segmentation.

One of the two main functions of the VMM is to provide its users, primarily virtual machines, the ability to define both the virtual address space and the effective address space. The VMM provides services such as create segment and destroy segment that allow a virtual machine the ability to define the valid subset of the virtual address space. It also provides services such as load segment register and clear segment register that control the relationship between the effective address space and the virtual address space.

The MMU translates a virtual address into a real address within a 24-bit real address space. The real address space may not be big enough to contain all of the pages defined in the virtual address space. Therefore, both real memory and a secondary storage device are used to contain all of the virtual pages.

A page fault interrupt results when an instruction references a memory location that is not defined in real memory. The second main function provided by the VMM is the page fault handler and it is the part of the system that is responsible for resolving page fault interrupts. The job of the page fault handler is to assign real memory to the referenced virtual page and to perform the necessary I/O to transfer its data into that real memory. This is known as demand paging.

All of real memory may become filled with code and data. When this happens, the page fault handler must select which page of data in real memory is to be transferred to the secondary storage device. The clock page replacement algorithm is used by the VMM to select this page. Here the real page frames are examined in a circular or clock-like order. A frame is selected for replacement if the data in it has not been referenced in the last cycle through the page frames. The "referenced" indicator is reset if the data in the frame has been referenced, and the next frame is then examined.

The secondary storage device used by the VMM is the disk. One or more disks may be used for paging. Also, paging I/O may be concurrently active on each of the paging disks.

Virtual Memory Data Structures

There are three primary data structures associated with the VMM. They are the segment table, the external page table, and the inverted page table (see Figure 1).

Virtual memory is divided up into segments. A segment represents an object such as a program, a mapped file, or computational data. A segment identifier can be thought of as the short form of an object name. The segment table defines the objects that can be referenced at any moment. It contains information such as the segment's size and the start of the segment's external page table. The segment table contains one entry for each segment and it is not pageable.

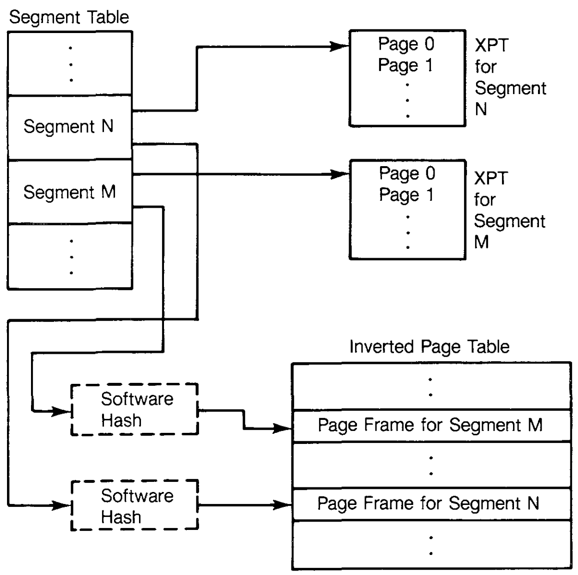


Figure 1 Virtual Memory Data Structures

A segment is divided up into 2048-byte virtual pages. A virtual page can be located in real memory or on the disk. Each segment has an external page table (XPT) with one 4-byte entry for each of its virtual pages. The XPT entries for a given segment are in contiguous virtual memory and are therefore directly addressable. An XPT entry describes the characteristics of its corresponding virtual page, such as its protection characteristics and its location on disk. The XPT is pageable.

There is a pool of external page table entries defined in the VRM [1] segment. The size of this pool limits the size of the virtual address space. The XPT for each defined segment is contained within this pool. The XPT for the VRM segment defines each page in the VRM segment, including the pool of XPT entries. The subset of the VRM segment's XPT that defines the pool of XPT entries is referred to as the XPT of the XPT. It is not pageable.

Real memory is divided up into 2048-byte page frames. A page frame can be thought of as a container for a virtual page. The Inverted Page Table (IPT) defines the virtual page that is currently associated with each page frame. The MMU uses the information in the IPT when translating a virtual address into a real address and when determining if a protection violation has occurred[2]. The MMU will respond with a page fault for any virtual memory reference that cannot be translated using the information in the IPT. The IPT contains one 32-byte entry for each page frame and is not pageable.

Support a Large Virtual Address Space
Virtual memory extends the power of computer memory by expanding the number of memory addresses that can be represented in a system while relaxing the limitation that all addressable memory must be present in real memory. The address translation hardware requires page tables fixed in real memory to perform its function. The size of a conventional page table is proportional to the size of the virtual address space, placing a practical limit on the address space size.

Paged segmentation is a means of reducing this overhead. It takes advantage of the grouping of related data in virtual memory by representing page table data separately for each segment. This allows space savings for short or unused segments of the address space.

An inverted page table further expands the range of addressability by reducing the real memory overhead required to support a very large virtual address space. Since an inverted page table contains an entry for each page of real memory, its overhead is proportional to real rather than virtual memory size. This makes it feasible to map a system's entire data base using a single set of virtual addresses (the "one-level" store). With a one-level store each segment can be large enough to represent an entire file or collection of data.

This is possible because the address translation hardware only needs the location of pages that are present in real memory. If a page is not present, the hardware must detect this fact, but it does not require the secondary storage address. The VMM does need this information, however. Hence, the VMM must keep this information in some data structure that is associated with the page. In the VRM this data structure is the external page table. Unless this external page table is pageable, the advantage of the inverted page table is lost, because the pinned real memory requirements become proportional to virtual memory size.

Large and Sparse Segment Support
The VMM supports segments of up to 256 megabytes. The VMM defines any segment that is one megabyte or larger to be a "large" segment. A large segment can be totally filled with data, assuming sufficient disk space. A large segment may also be lightly filled with data that is scattered throughout the segment. This is known as a sparse segment.

The external page table for a large segment can itself be fairly large. An XPT entry defines 2048 bytes of virtual memory. A page of XPT entries contains 512 of the 4-byte entries and defines 1 megabyte of virtual memory. Therefore, 256 pages of XPT entries are required to define a 256-megabyte segment.

Since the XPTs are pageable and reside in virtual memory, a subset of them describe the XPT area itself. These are the XPT of the

XPT. One entry in the XPT of the XPT defines a page of XPT entries or one megabyte of virtual memory. The XPT of the XPT for a 256-megabyte segment need only contain 256 entries, which require only half a page of memory. Therefore, the size of the XPT of the XPT required to define a large segment is significantly smaller than the XPT for that segment.

The VMM takes advantage of the XPT of the XPT in its support for large and particularly for large, sparse segments. For example, only the XPT of the XPT is initialized for a new segment. This decreases the number of pages that are initialized for a 256-megabyte segment from 256 to 1, thus decreasing the overhead when creating large segments.

Serialization
It is useful to describe the VMM in terms of the states of the objects it manages. The most important such objects are virtual pages and page frames. Each of these objects always has a well-defined state. Updates to the data structures that record these states must be logically serialized with respect to system events that require that the data structures be accessed or changed. Such events fall into three categories:

- Page fault interrupts
- Paging I/O completion interrupts
- Calls to VMM services

Without this serialization, the VMM would be unable to use these data structures safely. It must ensure that all accesses to these data are made within a "critical section." On entry to a critical section all objects must be in well-defined states. Code running in the critical section can perform state transitions, as long as it leaves all these objects in valid states on exit.

VMM critical sections are serialized by executing them as the lowest priority interrupt handler. This places them lower in priority than all I/O interrupts and higher in priority than all processes. Due to the characteristics of VRM interrupt handlers, this ensures that paging I/O completion interrupts and VMM services are serialized with respect to each other.

In order to support a large virtual address space it is desirable that some of the VMM data structures are pageable. Therefore, the VMM interrupt handler was extended to support the concept of backtracking with careful update. A page fault within a VMM critical section causes the critical section to be exited and makes the process that initiated the critical section wait for the page fault to be resolved. The process is allowed to retry executing the VMM critical section upon resolution of the page fault. This is backtracking. Careful update implies that all VMM critical sections must be coded such that all VMM data structures that are changed are left in a consistent state whenever the critical section can be exited, either by completing the service or by page faulting. Backtracking with careful update serializes the page fault handler with the VMM services.

I/O Management
The VMM attempts to manage disk I/O in such a way as to compensate for the imbalance between the high speed of the processor and the relatively low speed of the disk. This entails adding complexity to the page fault handler and to the disk I/O routines in order to decrease the average I/O time required to perform a disk I/O or to eliminate disk I/Os altogether.

Disk Affinity
One of the more important concerns in the VMM, in the area of performance, is to efficiently schedule disk requests so as to minimize seek time. This is done by attempting to write pages to nearby disk blocks, and, when reading them back in, to read more than one page at a time. The act of writing pages out to nearby locations on disk may be called "pageout affinity", while the act of reading more than one nearby page at once is called "prepaging".

Pageout I/O Affinity
Page faults tend to happen in bursts; that is, when a process is first invoked, it will "fault in" its "working set". It is desirable to have enough free page frames available to satisfy a "typical" burst of page faults. Not having an available free page frame can cause a process to wait on both a pageout to free a page frame and a pagein to bring in the desired page. The page replacement algorithm will select more than one page at a time to pageout, in order to maintain this threshold of available free page frames.

Careful management of paging space can take advantage of the above characteristics of the page replacement algorithm to reduce the I/O time associated with writing out pages to disk. This is achieved by always allocating paging space at pageout time. If a disk address is already assigned to a page, that address is freed and a new one allocated. Furthermore, a circular allocation algorithm is used. This means that two pageouts in a row will most likely be to adjacent locations in paging space. This concept of "late allocation" also makes it possible for the VMM to better know where the disk arm is located at that moment, and to find the paging space with the arm closest to it. Taken together these things tend to reduce the seek and rotational delays associated with VMM disk I/O operations.

Prepaging
When a segment is written out via the purge page range service, it is probable that the data in this segment is related; that is, it is data for a program, or it is a program itself. When this program runs, it will start page faulting on the segment that was written out. Some, or all of these page faults may require disk reads. Since the pages were placed near one another on the disk at pageout time, it is reasonable to assume that in a lightly loaded system the disk arm would not have to move much in order to read back all the pages. This assertion breaks down if some other disk I/O request is processed between the faults on the segment, thereby moving the disk arm away from the area where the segment resides. In order to counteract this problem, a "prepaging" policy is used in the page fault handler. What this means is that when the VMM processes a page fault that requires a disk read to resolve, it will attempt to read in pages that are nearby on the disk, and in the same segment as the one faulted on. Prepaging will also work for segments that are mapped to a file system, when the file system disk blocks are either contiguous, or close together on disk as is often the case.

Disk Cache
The VMM maintains a write-through disk cache that is under control of the page replacement mechanism. This disk cache can be thought of as a dynamic RAM disk that is managed by the VMM. Unmodified file system pages, when released, are placed into the disk cache. When a page fault occurs that might require a disk read to resolve, the page fault handler first looks in the disk cache to see if the contents of the desired disk address reside in the cache. If so, that page is "reclaimed" from the disk cache, and no I/O is required. Since cache entries may be stolen by the page replacement mechanism when there are no free page frames available, the size of the disk cache is directly proportional to the size of real memory and the level of system activity.

The VRM allows the disk to be accessed via minidisk manager[1] I/O operations and via the VMM. The VRM enforces the following rules to ensure that the disk cache is synchronized with minidisk operations. First, a write to a minidisk may cause an entry to be purged from the disk cache. Secondly, closing a minidisk may cause all of the entries in the disk cache associated with the minidisk to be purged.

Special Topics
The design of several of the functions provided by the VMM were greatly influenced by the virtual machine concept and by specific attributes of the AIX operating system.

Asynchronous Page Fault Processing
Typically in a paging system a process is forced to wait until all I/O required to resolve a page fault is complete. Other processes are allowed to execute, but the faulting process must wait until it can successfully execute the faulting instruction. This is known as synchronous page fault processing.

Synchronous page fault processing is not desirable when a multi-tasking virtual machine appears as a single process to the VMM. The multi-tasking virtual machine may have other tasks that can execute while the faulting task waits for the page fault to be resolved. The VRM solves this problem by informing the virtual machine about page faults via machine communications interrupts. The VMM generates a "page fault occurred" virtual interrupt for each page fault and a corresponding "page fault cleared" virtual interrupt when I/O is complete for the page fault. This allows the virtual machine the ability to dispatch another task while the faulting task waits for the "page fault cleared" virtual interrupt. This is known as asynchronous page fault processing since the entire virtual machine is not forced to wait.

Page fault notification virtual interrupts are under the control of the virtual machine. Therefore a virtual machine that executes only a single task can disable page fault notification interrupts and leave all page fault processing up to the VMM. Here the virtual machine's page faults would be processed synchronously. Similarly a multi-tasking virtual machine can disable page fault notification interrupts in selected critical sections when preemption is undesirable.

Mapping of Files
Usually, the data in a segment does not persist beyond the execution of a program. The VMM allows the data contained within a segment to be associated with files in the virtual machine's file system, thus allowing that data to exist after the execution of a program. This association of file data with virtual pages is achieved through mapped files.

The map page range service is provided to allow a virtual machine the ability to create a "one-level store" environment or a subset, such as mapping an individual file. This service is necessary because neither the operating system executing in the virtual machine nor the VRM have the capability by themselves to map a file. The virtual machine does not have access to the VMM tables and the VRM is designed to be independent of the virtual machine's file system structure. The map page range service provides the virtual machine the ability to tell the VMM the relationship between a logical entity, such as a file, and its location on the disk.

Scheduler Paging Control
Thrashing is a term commonly used to describe a system that is spending most of the time paging and little time performing useful work. The VMM maintains information in the virtual machine's page 0 that can be used by the virtual machine's scheduler to detect thrashing. The scheduler can periodically examine this information to determine how many concurrent tasks it will allow to be active.

The scheduler may determine that the system is thrashing and decide to reduce the number of active concurrent tasks by quiescing one or more of them. The VMM purge page range service provides the scheduler an efficient method to swap out a task's current working set. This can result in the pageout I/O affinity benefits discussed earlier. Furthermore, the task may also benefit from prepaging when it is reactivated.

Delayed Copy
Typically, the AIX operating system executes a program by using the fork system call to copy the process's address space and the exec system call to load and execute the program in the new copy of the address space. The copying of the address space can result in unnecessary processing and I/O delays when followed by an exec system call, since the exec system call will overwrite the just copied address space with the new program's code and data. Therefore, the VMM's copy segment service attempts to delay copying data until the data is actually referenced, and with the exception of the speed of the operation, the two address spaces are equivalent to the virtual machine. The data in the two address spaces need only be equivalent during the time between the fork and the exec. Much of the data may be unreferenced when an exec follows a fork, and such data is never actually copied.

Conclusion
The VMM simplifies the design of advanced applications and operating systems, reduces I/O costs, and complements the hardware's ability to support a large virtual address space. Advanced applications and operating systems are supported by the following VMM services:

- Demand Paging in a Virtual Machine environment
- Asynchronous page fault processing
- Mapped files
- Scheduler control information

The cost of VMM disk I/O is reduced by:

- Pageout I/O affinity
- Prepaging
- Disk caching

Finally, the VMM meets its major design objective of supporting a large virtual address space. The VMM takes advantage of the inverted page table by defining an external page table that has very small entries and that is directly addressable. Backtracking with careful update allows efficient support of a pageable external page table. Together, they greatly reduce the VMM data structure overhead associated with each page of virtual memory.

References
1. Thomas G. Lang, Mark S. Greenberg, and Charles H. Sauer, "The Virtual Resource Manager," *IBM RT Personal Computer Technology,* p. 119.
2. P.D. Hester, Richard O. Simpson, Albert Chang, "IBM RT PC ROMP and Memory Management Unit Architecture," *IBM RT Personal Computer Technology,* p. 48.

The IBM RT PC Subroutine Linkage Convention

J.C. O'Quin

Introduction

A subroutine linkage convention is a set of rules concerning the machine state at subroutine entry and exit. Because these rules are generally understood by compiler writers and assembly language programmers, it is possible to call separately compiled functions and get meaningful results. Since knowledge of these conventions pervades the system, it is important that they be carefully evaluated before introducing a machine with a new instruction set. Mistakes in this area usually cannot be remedied in future releases.

This paper discusses some important design decisions reflected in the linkage convention used by the IBM RT PC system. Details of this interface are described in section 6 of the *IBM RT PC: Assembler Language Reference Manual* [1]. The initial release provides compilers for C, FORTRAN, Pascal and BASIC, which all support this common linkage convention.

Performance Considerations

The subroutine call mechanism materially affects system performance. Studies have shown some UNIX programs spending as much as 20% of their CPU time in call/return linkage code [2]. Other factors, such as the number of registers available for allocation and optimization can have dramatic second-order effects on the quality of compiled code.

Good programming style suggests that programs be decomposed into small routines. System design should encourage this practice by providing an efficient call mechanism. This is particularly important for "reduced instruction set" machines, which emulate some higher-level instructions in software. The IBM ROMP microprocessor uses "extended opcode" subroutines to perform such functions as storage move, integer divide, and floating point operations.

A properly designed computer's performance is primarily limited by its memory bandwidth requirements. One effective method of eliminating storage references is to keep intermediate data in fast CPU registers. The IBM ROMP microprocessor has 16 general-purpose registers for this purpose, and instructions with only register operands are much faster than those involving storage references. This is true of all existing processors to varying degrees. Although mainframes with very large data caches provide relatively inexpensive storage references, their register operations are still faster. A primary design goal of the RT PC linkage interface is to minimize the number of storage references when calling a subroutine.

Function Prologue and Epilogue

When the framesize is less than 32,768 bytes, the function entry and exit code is very straightforward:

```
stm  r6, -save(r1)            # save modified
                                non-volatile
                                regs
cal  r1, -framesize(r1)       # adjust stack
                                pointer
.
.
.
lm   r6, framesize-save(r1)   # restore caller's
                                regs
brx  r15                      # branch to
                                return point
                                after
cal  r1, framesize(r1)        # restoring the
                                stack pointer
```

The "Load Multiple and Store Multiple" instructions (LM and STM) provide an efficient mechanism for saving some of the caller's registers. By convention, registers r6 through r14 must be preserved across the call. The stack pointer register (r1) must also be restored. This avoids forcing the caller to reload register variables and temporary data after every call. Research with the PL.8 compiler has demonstrated the benefits of preserving some, but not all, registers across calls.

The called routine will save only the registers it modifies. If it can perform its entire job using only registers r0 through r5, no registers need be saved. If registers r13, r14, and r15 are modified, but r6 through r12 are unused, then r13 would be specified in the STM and LM instructions in place of r6.

Register r1 was chosen for the stack pointer because the Load Multiple and Store Multiple instructions address registers from the first operand through register r15. This means that these instructions can be used within a routine without disturbing the stack pointer. These instructions are particularly valuable for performing large block moves.

The fact that r1 is not saved and restored across the call by the STM and LM in the function prologue and epilogue is no disadvantage, because the "Compute Address Lower" instruction (CAL) does the job more efficiently. In fact, the "Branch Register with Execute" (BRX) completely overlaps restoring the stack pointer with the instruction fetch for the return branch. If a subroutine does not require a stack frame, it need not update r1 at all. This means that small routines can be coded with no prologue or epilogue; the minimum call overhead is just a branch out and a branch back. The system call interface routines and the "extended opcodes" (such as move and divide) fall into this category. It is also possible for optimizing compilers to avoid prologues and epilogues for some small routines.

Note that there is no in-line check for stack overflow. The IBM RT PC AIX operating system uses page fault interrupts to detect overflow. A 256M-byte segment of the address space is reserved for each process's stack. This segment is "inverted", meaning that it grows towards lower-numbered addresses. The stack segment is automatically extended on demand, until a 1M-byte limit is reached, at which time the process receives a memory fault signal.

Parameter Passing

C Language Compatibility Requirements
Languages that permit a linkage editor to ensure that actual parameters and formal parameters agree in number and type improve not only program reliability, but also performance, since the parameter passing mechanism can be tailored to specific argument types. For example, it would be possible to pass floating point arguments in the floating point registers.

Unfortunately, the C language permits actual parameters to differ from formal parameters in both number and type. Some C programs address their arguments as an array in storage. This is an easy way to handle a variable number of arguments. The standard C library function "printf" is the archetypal example of this, but there are others; and UNIX System V provides a "varargs.h" include file in the standard distribution, which tends to encourage people to do this. Even though this practice can be considered non-portable, there seem to be enough programs using it to make supporting it worthwhile.

Passing Parameters in Registers
Within a single source program, register allocation is the compiler's responsibility, although C programmers may provide guidance via "register" declarations. When a subroutine call occurs, the linkage convention interacts strongly with register allocation. Expressions have been evaluated as arguments for the subroutine. We would like to pass their values in registers to avoid storage references in both the caller and the subroutine.

There is a conflict between our desire to pass values in the registers and the need to support programs like "printf". This is resolved by allowing C programs to view their arguments as a storage array with each argument aligned to a full-word boundary. Space for this array is allocated in the stack by the caller, but it does not store the first four words in the stack. Instead, these values are passed in registers r2 through r5. If the subroutine takes the address of some parameter, it must store these registers in the stack space provided. Then normal C pointer arithmetic will give the expected results.

Some compilers may elect to store all arguments except those explicitly declared "register". This is easy to implement, and surprisingly effective. Most of the functions in the standard C library and in the AIX operating system kernel benefit from this simple strategy.

The registers selected for passing arguments are not preserved across function calls because argument expression values are often not needed after the call.

Function Values
Function values are returned in a register, making them available for immediate use in the enclosing expression. A subtle advantage is gained by using r2 for this purpose. This register no longer contains any of the caller's data, and it is not unusual to pass a function value as an argument to another subroutine.

Functions return their values according to type:

- **int**, **long**, **short**, **pointer**, and **char** values are returned (right justified) in r2.

- **double** values are returned in r2 and r3.

- structures are returned in a buffer allocated by the caller, the address of which was passed in r2 as a hidden first parameter. This means that the first actual parameter word will be in r3 and that all subsequent parameters are moved "down" one word.

This structure return scheme has the advantage of being reentrant. A function returning a structure can be called recursively from a signal handler at any time without mishap. It also permits optimization of assignment statements such as "s = foo(x);" to return the result in the desired location. Its disadvantage is that a caller who does not want the returned value cannot omit the function declaration, since **int** would be assumed in that case. The advantages seem to outweigh this disadvantage. Incompatible declarations are likely to fail nearly every time, whereas small timing windows in which certain types of signal must not be received are much more difficult to discover and debug. In practice, these potential incompatibilities do not seem to cause trouble.

Summary

It is possible to pass parameters in the CPU registers and still maintain compatibility with C programs that use techniques like those provided in "varargs.h". The result is a speed-up in CPU time for system code of about 15% compared to an earlier implementation that passed all parameters in the stack.

Linkage interface experience on the RT PC project points out a major benefit of reduced instruction set machine architectures. Even after years of research compiling to the ROMP microprocessor and similar instruction sets, as late as December 1984, we were able to gain substantial performance improvements by further tuning the call interface. This would have been impossible had microcode changes to a high-level call instruction been required.

The fact that the deceptively simple problem of calling a subroutine turns out to have many hidden complexities may come as a surprise to some. The conflicting semantics of various programming languages make it difficult to arrive at a final solution to this problem that can be etched forever in silicon.

Acknowledgments
These ideas grew out of years of research and experiment on several different compilers. Major contributions were made by Marc Auslander, Jim Christensen, Martin Hopkins, Mark Laff, and Peter Oden.

References

1. *IBM RT PC: Assembler Language Reference Manual.* IBM Corp., document number SC23-0815-0.
2. S.C. Johnson and D.M. Ritchie, "Portability of C Programs", *The Bell System Technical Journal*, Vol. 57, No. 6, Jul-Aug 1978, p. 2044.

Implementation of a Virtual Terminal Subsystem

D.C. Baker, G.A. Flurry, K.D. Nguyen

Introduction

This article describes the IBM RT PC Virtual Terminal Manager subsystem, which is part of the Virtual Resource Manager. The subsystem provides terminal support for the RT PC virtual machine environment.

The terminal model for the RT PC must support the terminal requirements for the AIX operating system in a virtual machine environment. The IBM RT PC AIX operating system requires a "glass teletype" emulation such as the Digital Equipment Corporation VT-100 or the IBM 3161, which is an enhancement of the original keyboard send/receive (KSR) teletype. This KSR terminal model provides support for the AIX "termio" model used in the shell and the majority of the current teletype based applications. The model also must support other users of the virtual machine environment such as diagnostics, installation and various internal test packages. Although these additional users did not control the choice of the terminal model, the KSR does accomodate their requirements.

The KSR terminal model is an ASCII terminal emulation in the spirit of the ANSI 3.64 standard utilizing a PCASCII code set rather than the ANSI 3.4/3.41 code sets. The ANSI 3.64 data stream is extended, as specified by the standard, to support enhanced sound-generation capability, to handle the flow of locator events, and to provide various controls to switch physical displays, fonts, and terminal characteristics.

The classic teletype model has several limitations in the AIX environment. Consider multiple processes writing to the same terminal; with the classic teletype model, there is no guarantee of atomicity in the output streams. This has caused many unnatural contraints, most notably, blind background processing, which is difficult for the user to monitor. One can envision that the problem is aggravated by the multiple virtual machine environment provided by the RT PC, where there is even less possibility for synchronized use of the terminal.

One way to solve the multi-thread problem is to add physical devices to the workstation. However, there are obvious reasons that many tasks and/or virtual machines cannot each have a unique physical terminal. The cost of multiple displays, keyboards, locators and other interactive resources prohibits a profusion of such devices. A workstation generally restricts the number of devices it can support due to adapter slot or power limitations. The facilities (office or desk space, electrical outlets, etc.) also place obvious constraints on the number of devices that an individual can effectively use. Not so obvious is the inconvenience of the physical movement and refocusing of concentration required to use a multiplicity of interactive devices. Thus, one must conclude that these devices have to be shared in some fashion to minimize system cost and maximize operator convenience.

The existence of multiple thread environments as described above and the required sharing of physical interactive resources lead us to the concept of "virtual terminals" to provide terminal I/O in support of multi-tasking for an operating system running in a single virtual machine and/or for multiple virtual machines. By virtual terminals we imply the appearance, to a virtual machine or several virtual machines, of more terminals than physically exist on the workstation.

The implementation of multiple virtual terminals requires sharing of the available physical resources needed for interaction with the user. Obviously, these resources must at least be time-shared among the virtual terminals. We considered the implementation of space sharing of both the input (keyboard and locator) and output (displays) devices, resulting in a so-called "messy desk" model. We rejected the latter for displays because the significant processing required to support the "messy desk" is better spent in end-user application execution. Thus, virtual terminals simply time-share the physical displays, resulting in a full-screen virtual terminal management system. Our implementation of time sharing in no way restricts application packages from space sharing the screen of a single virtual terminal among multiple processes. As will be described later, however, we did choose to space-share the input devices.

The Concept of Virtual Terminals

The concept of terminal virtualization has advantages classically associated with virtual resources such as virtual storage or virtual disks. For discussion, let us consider the analogy between virtual storage and virtual terminals:

- A classical virtual storage system gives the programmer and or user the impression of having more storage than is physically present. The concept of virtual terminals allows the virtual machines the impression that there are considerably more display devices than are physically present, that there are more input devices than are physically present and that these devices have different characteristics than the physical devices.

- Virtual storage relieves the programmer of clumsy mechanisms, such as overlays, for dealing with limited physical resource. Virtual terminals relieve the programmer of developing his or her own mechanisms for dealing with the limitations of the actual resources.

- In a virtual storage system, programs can be written to be independent of the specifics of the physical resource, such as the size of primary storage, types and formats of secondary storage devices, etc. With virtual terminals, a program can be written to be independent of the specifics of physical terminal devices, for example, display buffer organizations, presence of optional input devices, etc.

Virtual Terminal Manager Subsystem Structure

Figure 1 shows a simplified conceptual model of the Virtual Terminal Manager. It consists of a supervisor, a Keyboard Device Driver (KDD), a Locator Device Driver (LDD), a Sound Device Driver (SDD), a Display Device Driver (DDD), and multiple virtual terminals.

Each virtual terminal embodies the characteristics of a single keyboard send/receive terminal. That is, it recognizes and processes the data stream received from the virtual machine causing the requested actions to occur, for example, move the cursor or draw characters onto the virtual display, insert or delete lines, clear the screen, or change the attributes with which characters are rendered. In addition to these actions, the outbound data stream can cause the generation of sequences of continuous tone sounds or cause the virtual display to be rendered on any of the available physical displays.

A virtual terminal receives input from a virtual keyboard and/or a virtual locator; it outputs to a virtual display. Thus, the virtual terminal can always expect to get input from its virtual input devices and can always output to its virtual display. These virtual devices may or may not have physical devices allocated to them, however, so the virtual terminal may not actually get input or write to a physical display. As each virtual terminal recognizes and processes the data stream inbound from the keyboard, it can, if requested, automatically echo various characters and simple functions back to its virtual display.

The virtual terminal has a special mode of operation in which it handles virtualized input in an essentially normal way but relinquishes

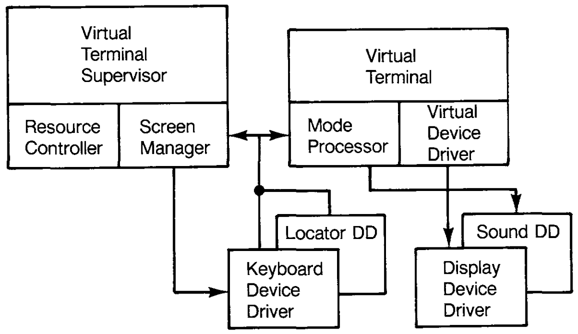


Figure 1 Virtual Terminal Manager structure

all physical display control to the application environment. In this mode, application programs can enjoy a very short path to the physical display and can implement, if they choose, arbitrarily complex graphics rendering algorithms, character algorithms, etc. The purpose of this mode is twofold. First, it allows the functional content of the base virtual terminal subsystem to be enhanced. Second, it reduces the performance overhead associated with device-independent display virtualization.

The supervisor comprises two components: the resource controller and the screen manager. The resource controller initializes and terminates the subsystem, allows a virtual machine to query and modify the configuration and characteristics of the interactive devices (the real terminal) available to the user, and allocates and deallocates the system resources required for the operation of a virtual terminal as it is opened and closed.

The screen manager is the analog of the paging supervisor in a virtual storage system. It performs the allocation of physical devices to the virtual devices used by the virtual terminals. The screen manager, in conjunction with the keyboard and locator device drivers, implements the time and space sharing required to virtualize these input devices. For example, we partition the physical keyboard into two subsets, termed logical keyboards. The first of the logical keyboards (the alphanumeric keys, the function keys, and their shifted states) is allocated at all times to one and only one of the virtual keyboards used by the virtual terminals; the other logical keyboard (the shifted states of the Action key) is reserved for the screen manager. In a similar manner, the screen manager, in cooperation with the display device driver, implements the space sharing required to virtualize a display. At any time, the display is allocated to one and only one of the virtual displays used by the virtual terminals.

The screen manager allocates all the physical devices en masse to the virtual devices of the ''active'' virtual terminal; that is, the virtual terminal with which the user may interact. The active virtual terminal can actually get input and produce output on a display. The screen manager also provides for reallocation of the physical resources. The impetus for reallocation results from either user requests (via the logical keyboard, or a similar logical mouse, allocated to the screen manager) or application requests. It involves deallocating the resources from the currently active virtual terminal and the allocation to the newly active virtual terminal. This allocation requires the cooperation of both virtual terminals. As mentioned above, the participation of the device drivers ensures synchronization of events such as keystrokes and work request acknowledgments.

It is important to note that while a single virtual terminal is restricted to a single physical display at any one instant, the collection of virtual terminals is not. The virtual terminal subsystem supports up to four physical displays and any virtual terminal may use any one of the four at any instant. This restriction is due to the number of expansion slots in the hardware rather than an architectural restriction of the Virtual Terminal Manager subsystem.

Resource Management
The virtual terminal supervisor provides for changing the physical environment. Global changes affect all virtual terminals. For instance, a virtual machine may add or delete physical displays, add a locator, a sound device or fonts, and change the physical characteristics of the keyboard and locator.

Local changes affect only a single virtual terminal. Applications effect these changes through the data stream sent to their virtual terminals. Applications may change the current font, the current physical display and various mappings and default operating characteristics. Additionally, we provide an escape mechanism to allow the application to release a display on a local terminal basis for direct access to the display adapter.

Conclusion
The virtual terminal system just described is deliberately restricted in order to reserve processing resources for application tasks. These restrictions take the following form:

- No support for a multi-window, space-sharing approach to physical display resource allocation
- No built-in graphics or paint program support
- Primitive resource allocation via the virtual terminal supervisor.

The above restrictions enabled us to implement a virtual terminal system with the following desirable characteristics:

- Multiple interactive threads with near single-terminal performance
- Physical-device-independent terminal model
- Application escape for direct physical display access.

The result is a virtual terminal environment compatible with existing applications and capable of supporting new, more sophisticated applications.

Use of a Coprocessor for Emulating the PC AT

John W. Irwin

Introduction

The IBM RT PC workstation, based on the ROMP processor, is a significant departure from previous small computer architectures and cannot run object code assembled or compiled for older processors.[1] Users who are moving up to the RT PC from the IBM Personal Computer (PC) may wish to continue using their present software library for reasons of economy and familiarity. Some existing software for the PC may never be rewritten for the new processor, particularly software that was originally written in assembler. The PC AT coprocessor was developed to provide continued use of such PC software products, including the popular editor and spreadsheet packages.

Even after most PC software is ported to the RT PC, the vast support network of programming expertise, publications, and programs that exists today for the PC AT may never be equalled for the RT PC. The PC AT coprocessor makes this base of support immediately available to the RT PC user by emulating the IBM PC AT within the RT PC architecture.

Critical to the use of Personal Computer software in the RT PC was the development of coprocessor hardware for the purpose of protecting the system against improperly written PC code and for sharing system resources between the ROMP and the coprocessor. When properly supported by a ROMP software driver, the result is a safe and accurate emulation of the PC AT over a wide variety of RT PC configurations.[2]

The System Environment of the Coprocessor

The PC AT coprocessor card was designed as a feature card for the RT PC Input/Output (I/O) channel.[3] The Intel 80286 runs simultaneously with the host ROMP processor, appears to the user as a PC AT, and can use the RT PC diskette drive or assigned minidisks in the RT PC disk space. The keyboard and display(s) are time-shared between the two processors under control of the ROMP.

The I/O channel in the RT PC is separate from the processor channel and is attached to the ROMP processor by an I/O Channel Converter (IOCC) as shown in Figure 1. The I/O channel is physically identical to the PC AT channel, has compatible signal timing, and will accept most 16-bit PC AT adapter cards and many 8-bit PC AT adapter cards. The I/O channel does not support ROMP memory access and therefore has unused bandwidth to support the coprocessor.

Like the other members of the PC AT family, the coprocessor will attempt to load and run any diskette the user places in the diskette drive. Neither the coprocessor nor the ROMP has any control over what code will be executed. Such "unfriendly" code may not follow approved access procedures to the I/O controllers, preserve critical memory locations, or share I/O devices such as the keyboard and display. A combined hardware/software protection system safely isolates such unfriendly code from the balance of the system.

Many adapter cards that may be installed on the RT PC I/O channel may not have existed when the PC AT code was written. An example is an all-points-addressable (APA) display adapter. The same coprocessor hardware that protects the system against improperly written PC code and supports the sharing of system resources between the coprocessor and the ROMP is used by ROMP host software to emulate current PC AT adapters while using the new adapters.

Hardware Design Features

The coprocessor is packaged on a PC AT form factor adapter card. The heart of the PC AT system board is duplicated in the coprocessor architecture in that 65,536 distinct I/O addresses are possible using a 16-bit subset of the 24 I/O channel address lines. Within this address space, 80286 access must be limited to those I/O devices assigned to the coprocessor while access is blocked to those devices assigned to the ROMP. This is accomplished by an 8192 by 1-bit Random Access Memory (RAM) on the coprocessor card, which can be written to only by the ROMP. During I/O operations by the coprocessor, the trap RAM is addressed by the 80286 I/O address lines A15-A3, so that the corresponding access bit is read from the RAM. In the trap RAM, each bit controls

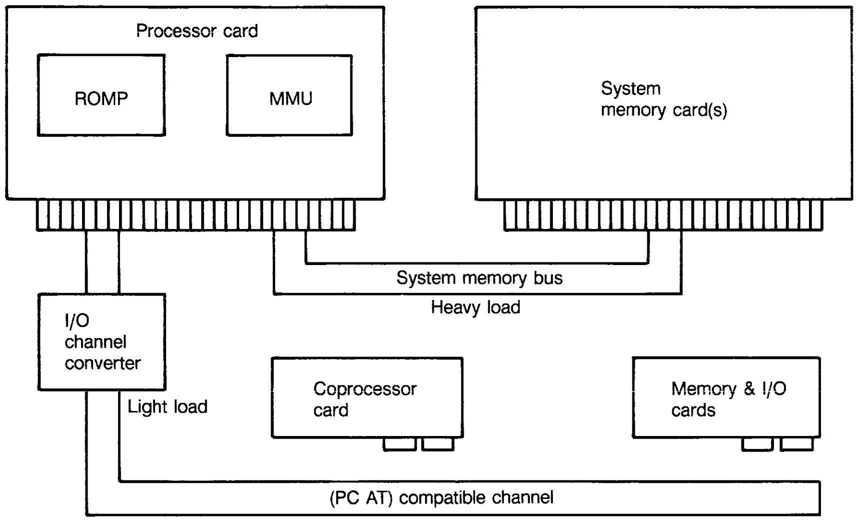


Figure 1 System Block Diagram

access to a group of eight I/O addresses. As shown on Figure 2A, if the bit is on, the coprocessor can access the device normally, otherwise the coprocessor access is trapped and one of several conditions may exist:

1. The device does not exist on the system.

2. The device is currently assigned to the ROMP. For instance, the printer might currently be in use by the ROMP and unavailable to the coprocessor.

3. The device may exist in a different form than expected by the coprocessor code and the ROMP will emulate the device expected by the coprocessor. The keyboard is an example.

4. The device is assigned to the coprocessor, but the ROMP must track all changes made to the device by the coprocessor. A time-shared display adapter is an example.

5. The device address is being used as a parameter passing port to synchronize operations between the coprocessor and the ROMP. The diagnostics use this mechanism.

When an I/O trap occurs, the coprocessor is stopped by assertion of the NOT-READY line. The coprocessor card I/O channel drivers are turned off so that the ROMP can use the channel to access the coprocessor trap and filter registers and other I/O devices needed to emulate the addressed I/O device. The ROMP can set up the coprocessor to issue an interrupt whenever an I/O trap occurs.

The ROMP responds to an I/O trap by reading the flag register of the coprocessor where individual bits show I/O read trap, I/O write trap, and the I/O access width (8 or 16 bits). If the trapped operation is an I/O write, the ROMP reads the coprocessor trap address and trap data registers and emulates the hardware device. The ROMP read of the coprocessor trap data register releases the coprocessor for further operation. If the trapped operation is an I/O read, the ROMP must read the trap address register, calculate the emulated response to the read, then write that response to the coprocessor data register. The ROMP write to the data register releases the coprocessor for further operation.

Interrupt Filtering
The I/O channel interrupt lines are shared by both processors and certain conditions must be met for the system to operate:

1. The coprocessor must not respond to interrupts from I/O devices assigned to the ROMP. The code running in the coprocessor cannot be controlled; therefore, such interrupts must be blocked by hardware.

2. The coprocessor must have free access to interrupts from I/O devices assigned to it.

3. The ROMP must force interrupts to the coprocessor for certain shared or emulated devices.

The coprocessor card contains two 16-bit registers for interrupt control as shown in Figure 2B. One register is used to mask I/O channel interrupts to the coprocessor, the other register is used to force interrupts to the coprocessor. By the manipulation of these two registers, the ROMP completely controls the interrupt environment of the coprocessor.

The interrupt mask register contains a bit for each I/O channel interrupt line. If the bit is set by the ROMP, that I/O channel interrupt is gated to the interrupt controller, otherwise the I/O channel interrupt is blocked and cannot affect the operation of the coprocessor.

The interrupt force register contains a bit for each channel interrupt. If a bit is set by the ROMP, the corresponding interrupt is asserted. Additional register bits are provided to force the keyboard and non-maskable interrupts (NMI) to the 80286 since these interrupts do not appear on the channel.

Sharing Video Resources
The problem of sharing or emulating video adapters is made difficult by the variety of video configurations that are supported by the PC AT code and the different set of display options available in the RT PC. Video sharing or emulation is accomplished by a combination of ROMP code and video filter logic on the coprocessor card.

If the user prefers to have a display dedicated to the coprocessor, any PC AT compatible display may be plugged into the channel for this purpose, provided that there are no address conflicts with displays assigned to the ROMP. The I/O filters and traps are set by the ROMP to allow the coprocessor access to the coprocessor display and to prevent access to the ROMP display(s).

A lower cost configuration is a display time-shared between the ROMP and the coprocessor, under operator control through the keyboard. To properly restore the display when the coprocessor regains control, the ROMP must keep a record of coprocessor video actions occurring during the time the ROMP is using the display. Control actions to the display adapter are trapped and recorded using the I/O traps. The memory mapped video buffer is recorded by video buffer relocation hardware as shown in Figure 2C.

In the PC AT memory map there are 128K bytes assigned to video buffers. At address 0A0000h, there are 64K bytes assigned to the

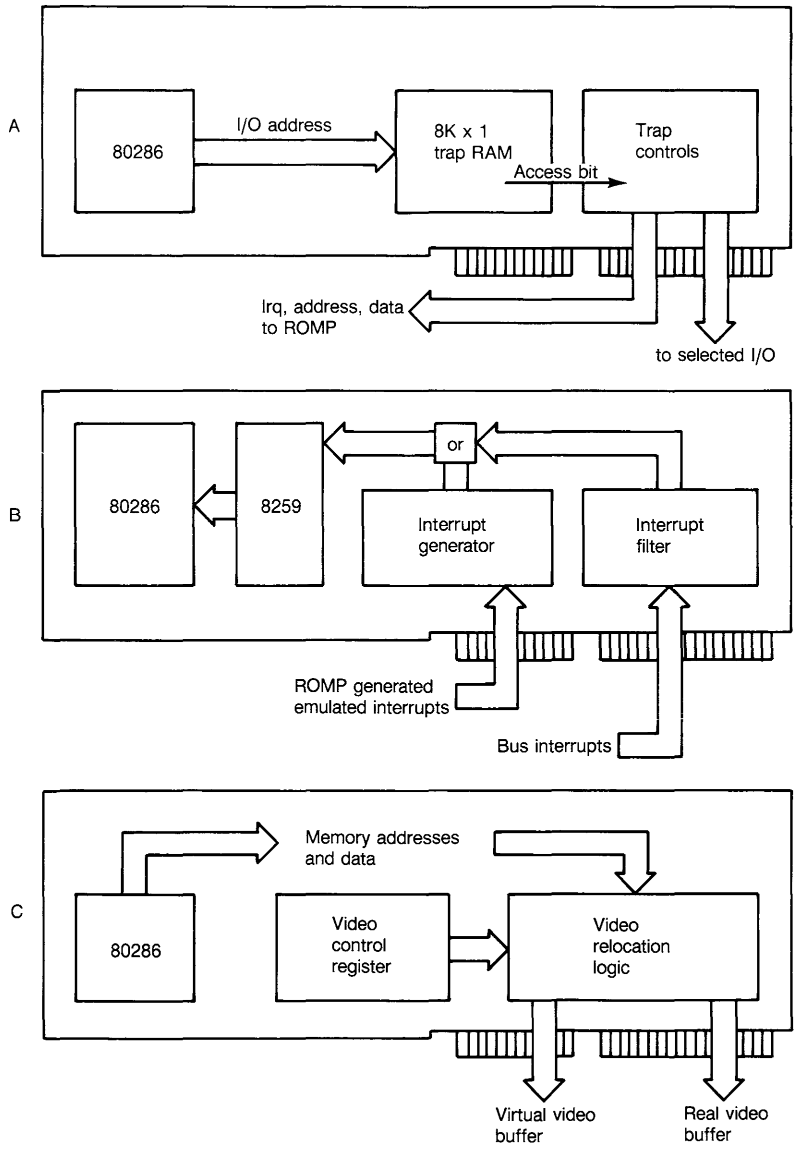


Figure 2 I/O Traps and Filters

Enhanced Graphics Adapter (EGA). At 0B0000h, there are 32K bytes assigned to the monochrome adapter and also used by the EGA. At 0B8000h, there are 32K bytes assigned to the Color Graphics Adapter (CGA) and also used by the EGA. The coprocessor hardware can distinguish these ranges and process accesses to each range separately.

The ROMP writes to a control register on the coprocessor card to set each buffer address range to one of four modes:

1. Assigned to the coprocessor
2. Made invisible
3. Relocated into a memory buffer
4. Relocated into a memory buffer and the addresses of the buffered data queued.

When a display is assigned to the coprocessor, the traps and relocation hardware for the corresponding address range are turned off.

When a display range is made invisible, I/O reads and writes are trapped, reads and writes to the video buffer are suppressed, and the coprocessor is thus prevented from interfering with displays assigned to the ROMP.

When the installed display adapter is one of the common adapters used by the PC family, display control is not difficult. Control becomes much more difficult when the installed display adapter is not supported by the code running in the coprocessor (an APA display) or when the processor display output is mapped into a window on a shared display. In these instances, the ROMP must translate the virtual video buffer to the display.

When the ROMP translates the virtual buffer to a display (Figure 3A), the ROMP code must know when the video buffer has been changed by the coprocessor. This may be accomplished by using interrupts or by polling. When the coprocessor display activity is low, the ROMP allows a coprocessor interrupt to occur whenever a buffer write occurs. This interrupt is issued without stopping the coprocessor. Since display updates tend to occur in bursts, the interrupt is masked off during periods of high display activity and the ROMP then polls periodically for buffer changes.

The display may have a different pel density or pel aspect ratio than the display supported by the coprocessor code, so considerable processing is required to map the virtual buffer to the screen. If the ROMP had to refresh the entire screen each time a change was made by the coprocessor, emulation performance would be entirely unacceptable. For instance, the coprocessor update of the time of day on the screen might frequently write a single character to the screen. If the ROMP could not easily determine which characters were changed, this action could result in an unacceptable ROMP processing load to examine the entire buffer for changes or to recalculate the entire screen. Either method would require processing several thousand bytes of buffer data to make the single-byte change.

The video address queuing performance aid is used to inform the ROMP exactly which buffer bytes have been changed so that minimum buffer data has to be processed. This performance aid yields as much as a 100,000 to 1 performance improvement.

The queue is a memory area assigned by the ROMP (Figure 3B) that is organized as a circular buffer. The low-order address of the circular queue is provided by a counter on the coprocessor card and the queue size can be set to 1024, 2048, or 4096 entries. When relocation with queuing is selected, the coprocessor card logic inserts a hardware-generated memory write cycle immediately after each write to the virtual video buffer.

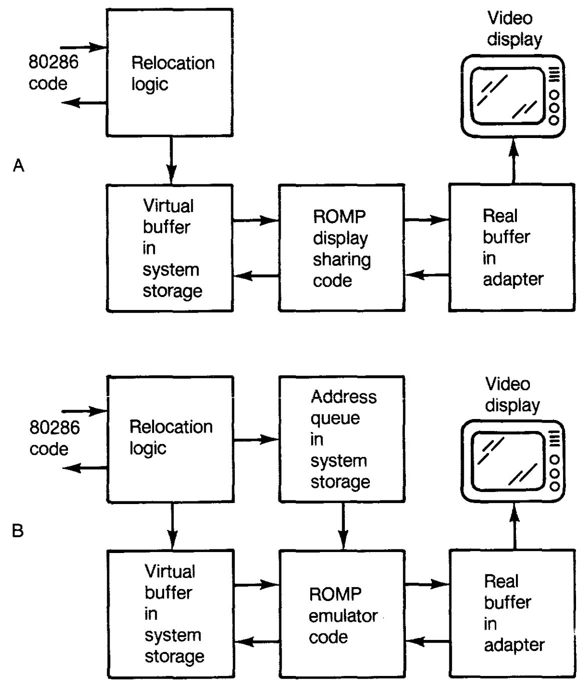


Figure 3 Video Relocation and Queuing

The low-order 16 bits of the video address are written into the queue area in memory and the hardware queue input counter is advanced. The input counter (queue tail) is read by the ROMP, the data translated, and the output register (queue head) is advanced by the ROMP. An interrupt informs the ROMP when the queue overflows during scrolling or clearing the screen. When overflow occurs, the ROMP resets the queue pointers and regenerates the entire screen.

Bus Arbitration
The separate memory and I/O channels of the RT PC system relieve the I/O channel of the ROMP instruction and data fetch load so that the channel is lightly loaded when the coprocessor is not running, averaging about 10% usage. The coprocessor, when running,

can use about 90% of the available bus cycles or all the available bandwidth.

An additional Direct Memory Access (DMA) channel, using a special arbitration method, was added to the system board logic to service the coprocessor. In the channel socket assigned to the coprocessor, the pins normally connected to DMA channel 7 are, instead, connected to the special arbitration logic, DMA channel 8. Where the acknowledgment to normal DMA channel requests grant the channel to a secondary master for an indefinite length of time, the special coprocessor arbitration grants the channel to the coprocessor only until another user (or memory refresh) requires service. At that time, the acknowledge signal for the coprocessor is dropped by the arbitration logic to signal the coprocessor to vacate the channel at the next possible point in the cycle. The coprocessor drops the request and master lines to release the channel and then immediately requests service again to recover the channel when it is free. The coprocessor DMA channel is the lowest priority, below ROMP Programmed I/O (PIO), refresh, and all other DMA channels, so that other channel activity is affected very little by the coprocessor.

System Memory Versus Channel Memory
If channel memory is not installed, the coprocessor uses system memory that is pinned by the VRM and will not be paged out. This is the most economical coprocessor configuration, but not the best performing one. Because of the long access path from the coprocessor to system memory, the coprocessor performs only slightly better than the PC XT and affects ROMP performance by memory and I/O channel interference. If better coprocessor performance is required, a 512K, 16-bit, PC AT channel memory card can be installed at address 0 and additional memory cards may be installed above one megabyte.

At the time of coprocessor initiation, the ROMP code tests for the presence of channel memory at channel address 0 and again at one megabyte. If memory is found at either location, all contiguous channel memory is used by the coprocessor. With most of the coprocessor memory access cycles taken from the one wait-state channel memory, the coprocessor performance approaches PC AT performance. There is less system memory contention with the ROMP, so that ROMP performance is increased as well.

Conclusions

The RT PC's PC AT coprocessor card with its ROMP programming support represents an ambitious attempt to emulate a specific machine environment within a machine of radically different architecture. This aim has succeeded to the degree that, except for processing speed, the user cannot ordinarily distinguish the coprocessor from an actual PC AT. The most important new features of the coprocessor are the ability to provide true concurrent processing with system protection against unfriendly coprocessor code and the combination of coprocessor hardware and ROMP software that provides flexibility in I/O adapter and I/O channel or system memory allocation. Future expansion is provided by the ability to emulate PC AT adapters using new RT PC adapters or future I/O device adapters not known to the original writers of the PC code. In addition, the coprocessor features special arbitration for increased I/O channel performance.

References

1. P.D. Hester, Richard O. Simpson, Albert Chang, "IBM RT PC ROMP and Memory Management Unit Architecture," *IBM RT Personal Computer Technology,* p. 48.
2. Rajan Krishnamurty and Terry Mothersole, "Coprocessor Software Support," *IBM RT Personal Computer Technology,* p. 142.
3. Sheldon L. Phelps and John D. Upton, "System Board and I/O Channel for the IBM RT PC System," *IBM RT Personal Computer Technology,* p. 26.

Coprocessor Software Support

Rajan Krishnamurty and Terry Mothersole

Introduction

Compatibility with existing applications that run on the IBM Personal Computer AT was deemed to be a significant enhancement to the RT PC ROMP-based product. The IBM RT PC Personal Computer AT Coprocessor Services LPP provides the user with a means to run PC applications on the IBM RT PC Personal Computer AT Coprocessor card in a way that is essentially the same as their execution on a PC AT. The coprocessor card consists of an Intel 80286 processor and hardware support to allow PC AT applications to be executed concurrently with ROMP applications[1]. Coprocessor software support includes ROMP code executing in the Virtual Resource Manager, which provides several levels of device support and virtual terminal ring support with ROMP virtual terminals. Also, the Coprocessor LPP includes AIX Shell code to allow the user to configure the PC AT environment and start/end a coprocessor session.

System Environment

The AT Coprocessor Services LPP, when installed on the RT PC, allows the PC AT coprocessor to run concurrently with the native ROMP processor. The coprocessor card is attached to the I/O channel. Figure 1 shows the PC AT coprocessor system environment and Figure 2 shows the coprocessor software support. VRM software running on ROMP, with the aid of trap logic on the coprocessor card, protects the RT PC base operating system from applications

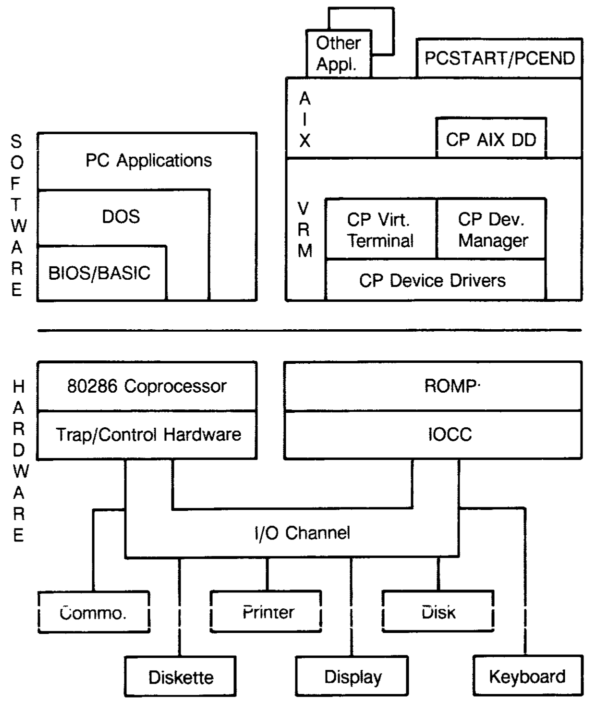


Figure 1 Coprocessor environment

running on the coprocessor. Most of the challenge of incorporating the coprocessor was the allocation of memory, sharing of devices, supporting displays that are not supported by the PC AT, and monitoring I/O accesses in the creation of a PC AT environment.

An AIX command, PCSTART, is provided to the user to tailor the configuration of the PC AT environment. PCSTART allows the user to select a combination of system resources that are to be attached to the coprocessor. The user may optionally save this configuration in a default profile to be used in future initiations of the coprocessor. PCSTART also provides a prompting mode for the casual user, in which the user is first shown the current default value for a parameter and has the option to alter its value.

There is no memory on the coprocessor card, so memory is provided as a mixture of system memory and I/O channel-attached memory. Memory is first allocated from the cards plugged into the I/O channel. If there is not enough channel-attached memory (there might not be any), the rest is provided from ROMP system memory and pinned to prevent page faults. This is accomplished by setting up the IOCC to convert the I/O channel addresses to the appropriate virtual addresses. The virtual addresses use the MMU segment register that is dedicated for I/O device access to system memory. Channel-attached memory must begin at address 0 and be continuous if it is to be recognized and used by the coprocessor. The amount of system memory that is available to the coprocessor depends upon how much is left after VRM and the operating system have met their requirements.

Devices connected to the I/O channel are supported in different ways, depending on the specific attributes of the device. The following

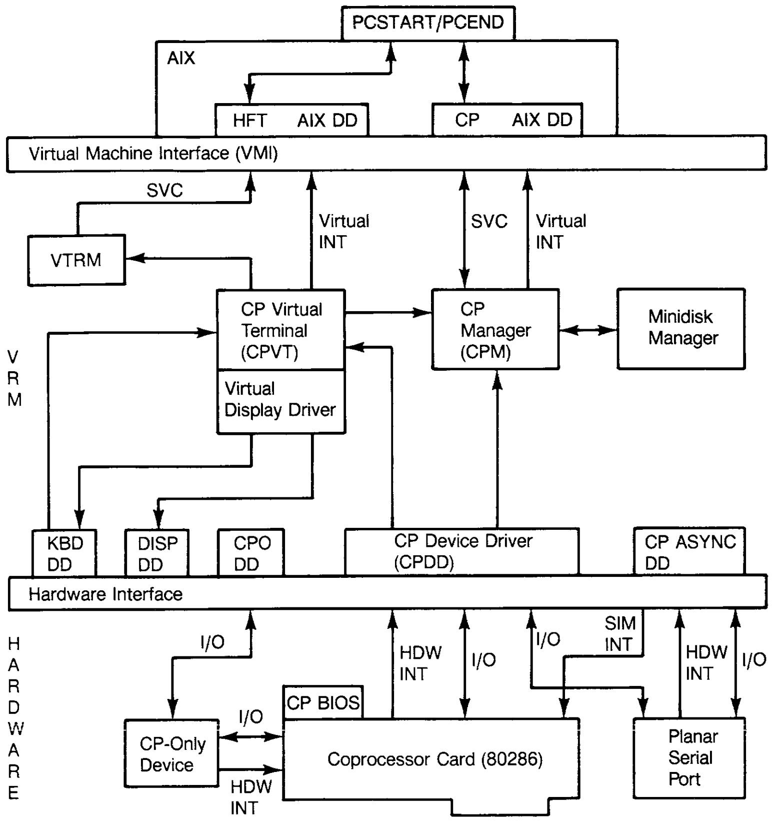


Figure 2 Co-Processor Software Structure

Device	ROMP Only	Shared	Dedicated	Coprocessor Only
MMU	X			
IOCC	X			
System Mouse	X			
DMA		X		
Disk		X		
Keyboard		X		
Real Time Clock		X		
Configuration CMOS RAM			X	
Display		X	X	
Communications Ports			X	
Planar Serial Ports			X	
Printer			X	
Diskette			X	
Speaker			X	
New Devices			X	
System Memory			X	
I/O Channel Memory			X	
80286 Int. Ctlr				X
80287 Math Coprocessor				X
Interval Timer				X
Light Pen				X

Figure 3 Device Support Modes

definitions describe the range of device support:

- ROMP only — the device is available only to the ROMP.

- Shared — a device is shared when it can either be accessed by both processors concurrently (e.g., disk, DMA) or can be dynamically switched between them (e.g., the display). Device sharing is accomplished by allowing only the ROMP to issue the I/O commands to the device.

- Dedicated — the device is allocated to the coprocessor for the duration of the coprocessor session. If the device is PC AT-compatible (e.g., planar serial ports), coprocessor accesses to it go directly to that device with no ROMP intervention. If the device is not PC AT-compatible, coprocessor accesses to it are trapped and emulated by the ROMP.

- Coprocessor only — available only to the coprocessor. Either the ROMP is unable to get to it or the device is never allocated to the ROMP.

Figure 3 shows the ways in which the various supported devices are treated.

Disk support is provided using RT PC minidisks. File system integrity is maintained by trapping all coprocessor accesses to the disk. Coprocessor support code performs the data transfers using the minidisk manager and then sends acknowledgments back to the coprocessor.

Terminal Support: Displays
The RT PC display support plan includes PC AT supported display adapters and new RT PC adapters that are unknown to the PC family product line. Given that the coprocessor option could reside on any RT PC display configuration, code and hardware were required to supply the coprocessor user with various display support possibilities in order to run PC applications on a non-PC supported display adapter. Thus, the coprocessor user has two display configuration options available: dedicated

mode, which allows the coprocessor to access the display adapter directly; and monitored mode, which prevents the coprocessor from accessing the adapter directly, while allowing the coprocessor virtual terminal to be in the ring of virtual terminals supported by the Virtual Terminal Manager[2].

Allowing the coprocessor to run a PC-supported system display adapter in dedicated mode (with channel-attached memory) gives the user up to 80% of the performance of the PC AT since there is no intermediate device driver trapping/emulating the I/O address range or relocating the video memory. The only Release 1 displays that this mode can be used with are the IBM Personal Computer Monochrome Display Adapter and the Enhanced Color Graphics Display Adapter. The APA8 can only be run dedicated if a future PC application is written that supports it. In any case, the APA8 in dedicated mode is not supported as the primary coprocessor display.

Depending on the display utilization by ROMP virtual terminals, dedicated mode is formally defined as "direct" or "allocated:"

- If ROMP had a virtual terminal opened on the requested system display at the time the coprocessor was started, then the display is configured in "direct" access mode. In this mode, the virtual terminal does not provide a device-driver interface that monitors the state of the adapter at all times (I/O and video buffer). The only way for the user to access the next ROMP virtual terminal in the ring is to terminate the coprocessor virtual terminal session by use of a special key sequence (CNTL-ALT-ACTION). This prevents the user from terminating the coprocessor session inadvertently with the ALT-ACTION hot-key sequence.
- If ROMP did not have an open virtual terminal on the requested system display, then the display is "allocated" to the coprocessor for the duration of its session. In this mode, the user can traverse the virtual terminal ring without ending the coprocessor session.

The user is unaware that the display is configured as dedicated/direct or dedicated/allocated, and the performance is the same. Note: If the system display can be allocated to the coprocessor (as opposed to direct), it will be, so as to enable the user to hot key to other virtual terminals.

If a PC AT-supported system display cannot be allocated to the coprocessor, or if the user wishes to emulate the PC Monochrome adapter or PC Color Graphics display adapter on the APA8, the system display can be configured to be shared with ROMP in a display access mode referred to as "monitored." All I/O commands from the coprocessor are trapped and saved, to allow the display adapter state to be restored when control is returned to the coprocessor. When the display adapter is the PC Monochrome or the Enhanced Color Graphics Adapter, the video buffer is accessed directly by the coprocessor. When the display adapter is the APA8, video buffer accesses are relocated to system memory and the APA8 virtual display driver is used to update the screen. This mode also frees the display adapter so that ROMP can open new virtual terminals on it. Control of the requested system display adapter is switched to the coprocessor when its virtual terminal becomes active via the ALT-ACTION hot-key sequence.

Coprocessor support of the APA8 was the major driving force behind providing a display access mode that would allow sharing display devices between ROMP and the coprocessor. The coprocessor virtual terminal mode processor utilizes the same VRM virtual display driver (VDD) interface that other virtual terminals do, to emulate the PC Monochrome Adapter and the PC Color Graphics Adapter. Emulation of the graphics modes involved a great amount of transformations on the video pel buffer. Emulation details are outlined below:

- Color text emulation is supported on the APA8. However, the color select register is ignored since this adapter is monochrome. Text characters are built up into text lines, while the color attributes are ignored. This allows an application that switches between text and graphics to operate on this one adapter, without requiring the user to configure in two displays to support the two different modes.
- The pel resolution and the pel aspect ratio of the PC Color Graphics Display Adapter differs from the APA8. Converting a graphics image built for a 640 x 200 resolution display with a pel ratio of 2:1, to a 720 x 512 high-res display with a pel ratio of 1:1 was accomplished with a simple and fast algorithm that duplicated scan rows to create a 640 x 400 image. Basically, the algorithm squared off the rectangular pel ratio of the PC color graphics image, in order to proportion the image onto the square pel ratio of the APA8. The resulting image was then centered onto the APA8 screen with a border around it.
- To support an application operating in 320 x 200 medium resolution color graphics mode, four different shades of gray are

provided, by a "half-toning" method, to simulate the four selected colors. On a Color Graphics Display Adapter (CGA), two bits describe a logical pel unit, which consists of two physical screen pels. With the scan rows duplicated as in high-resolution (640 x 200) mode, this creates a 640 x 400 image with a square logical pel unit of four physical pels. The four pels comprising the logical pel unit are turned on or off to produce one of four shades (i.e., black, dark gray, light gray, white). As an example, with all pels turned on, the color produced is white. With two diagonal pels turned on, the color produced is a light gray.

The coprocessor may have use of more than one display, but it can use only one ROMP system display on which a virtual terminal exists. Other displays can be ROMP displays that can be allocated to the coprocessor or coprocessor-only display devices such as the PC Color Graphics Adapter. The PCSTART command monitors the system usage of the available system displays and only allows a valid coprocessor environment to be set up.

Terminal Support: Keyboard
The keyboard on RT PC is a shared device between the ROMP processor and the coprocessor. Residing in the VRM coprocessor terminal support code are a device driver and a mode processor that provide a PC AT 8042 keyboard controller interface to the coprocessor. The mode processor takes in RT PC key positions from the VRM keyboard device driver, translates them to PC or PC AT scan codes, places them in a simulated keyboard buffer, and then generates an interrupt to the coprocessor. The scan code sets for PC and for PC AT are stored in a structure indexed by the type of the emulated keyboard. The simulated keyboard buffer is a 16-byte FIFO queue with a 17th byte for overrun.

The keyboard layout of the RT PC keys is a superset of a PC AT keyboard. It contains all of the engravings resident on a PC AT keyboard, while some of them have been moved or duplicated to other key positions. For example, the new set of cursor motion keys and edit keys (INS, DEL, PAGE UP, PAGE DOWN, HOME, and END), will be translated as engraved, without numbers. The coprocessor virtual terminal will maintain state flags for the NUMLOCK and the SHIFT keys, as well as for the CAPSLOCK, SCROLL LOCK, CTRL, and ALT. Depending on the combined state of the NUMLOCK and the SHIFT, SHIFT make/breaks may be sent around the scan codes for the new native cursor motion and edit keys, in order to force the engraved key translation.

From a system perspective, the coprocessor mode processor works with the VRM keyboard device driver in raw mode, receiving all makes and breaks of keys. During virtual terminal transitions, the break of keys may be sent to the next active terminal. Since the mode processor keeps track of the control/shift keys, it can send break scan codes appropriately. The mode processor also traps the situation where the user wishes to terminate the coprocessor session with the CNTL-ALT-ACTION key sequence. This simulates the user entering the PCEND command in the AIX operating system.

Disk Emulation
The fixed disk devices on the RT PC are divided into logical minidisks that are managed by the minidisk manager. PC AT fixed disks are emulated through the use of minidisks. Up to two minidisks can be allocated to the coprocessor during a given session. The only interface to the fixed disk from the coprocessor is through BIOS. Any attempts to issue I/O instructions to the physical disk addresses are trapped by the coprocessor card and are interpreted as unallocated device accesses.

Async Redirection
There exists a coprocessor async device driver that will emulate serial port functions of the PC AT serial/parallel card on the native RT PC planar serial ports (RT PC model 6150 only) which are Zilog Z8530 based. I/O between the coprocessor and the PC AT Serial/Parallel card (National 8250) is redirected to one of the two planar async ports. Control information being sent will be translated from the PC AT Serial/Parallel card format to the planar serial port format and redirected to the async ports on the planar. Data bytes containing control information being received from the async ports on the planar will be translated from the planar serial port format to the PC AT serial port format and redirected to the coprocessor.

All PC AT serial port commands are emulated on the planar ports except stuck parity and diagnostic loop mode.

Summary
The RT PC AT Coprocessor Services LPP provides PC compatibility to the user, complete with device-monitoring features to emulate an IBM PC AT environment.

There are known incompatibilities that exist due to hardware restrictions, or to limited software emulation capabilities:

1. (Mini)disk access supported through BIOS interface only. All BIOS calls are supported except formatting a disk,

initializing drive pair characteristics, read long, and write long.

2. Code dependent on instruction execution timing may cause unpredictable results.

3. Some 6845 display controller commands and some PC color graphics modes are not supported on the APA8.

4. Enhanced Color Graphics Adapter high-resolution modes are supported only on the Enhanced Color Display in dedicated mode.

5. Partial DMA support.

Even with the known restrictions, the AT Coprocessor LPP provides the RT PC user with the ability to run PC applications concurrently with ROMP virtual terminals. The simulated PC AT environment is made virtually transparent to the user after configuration time and allows system devices to be shared by both processors.

Acknowledgments
The Coprocessor Licensed Program Product is the result of the outstanding contributions of many people. The dedication of the development team which included Harish Agarawal, Wayne Blackard, Dean Boschult, David Chiang and Chuck Nunn and the management leadership of Johnny Barnes, Joe Corso, Stan Douglas, Pat Motola and Gary Snyder during the various stages of the project were critical to the success of the product. The cooperation and efforts of the Systems Test, System Assurance and VRM departments were key in ensuring a high quality product. The encouragement and support provided by Clay Cipione, Glenn Henry, Larry Loucks and Frank King were invaluable in helping us through many difficult times.

References

1. John W. Irwin, "Use of a Coprocessor for Emulating the PC AT," *IBM RT Personal Computer Technology,* p. 137.
2. D.C. Baker, G.A. Flurry, and K.D. Nguyen, "Implementation of a Virtual Terminal Subsystem," *IBM RT Personal Computer Technology,* p. 134.

PC DOS Emulation in the AIX Environment

Leonard F. Brissette, Roy A. Clauson, Jack E. Olson

Introduction

This article describes the major features of the PC DOS emulation functions that are a part of the IBM RT PC AIX operating system. The widespread acceptance of the IBM PC and the DOS operating system mandated that a user interface with a similar set of DOS functions be part of the RT PC AIX. These functions allow the existing PC DOS user to easily move up to the higher performance and capacity of the RT PC with minimal re-training. The nature of the AIX operating system allows a single RT PC to concurrently support multiple PC DOS emulation users, hence an apparent "multi-tasking" DOS. The PC DOS emulation functions are provided by two major pieces; the RT PC AIX DOS Shell and the RT PC AIX DOS File Access Method.

The RT PC AIX DOS Shell Interface

The RT PC AIX DOS Shell (hereafter referred to as the "DOS Shell") provides a more friendly alternative user interface to the traditional UNIX Shells. It allows a user to manipulate "UNIX" files on AIX file systems and "DOS" files on diskettes or RT PC minidisks. It is an emulation of most of the PC DOS 3.0 functions, using identical command syntax. Those functions not emulated are PC DOS or hardware specific.

The DOS Shell allows for PC DOS batch file execution and also emulates the line editing functions provided by PC DOS. An escape mechanism is provided (via a prefix of '!') to allow a user to execute an AIX Shell command without having to exit the DOS Shell and then restart it.

Figure 1 shows one instance of a DOS Shell and its relationship to the rest of the system. At "login", a user's AIX .profile file can specify "dos" and the DOS Shell will automatically be invoked for the user. Using this procedure makes the system look entirely "DOS" to the user, except for the login. Environment variables are used to define the relationship between DOS Shell "logical devices" like A:, LPT1:, etc., and the system's resources. These environment variables may be individually overridden by placing the override information in the user's .profile file. Thus, each DOS user on the system can customize his or her own DOS environment.

A user may also place an "autoexec.bat" file in the A: drive, and at "login" time this batch file will be executed just as in DOS.

The DOS Shell uses the DOS File Access Method set of routines in the DOS library to access user files. These routines allow transparent access to either an AIX file system or a DOS file system and will be discussed in the next section. The DOS file system may have been created for use by the PC AT coprocessor feature. Thus, data objects created or processed on the coprocessor may also be processed by native running RT PC applications, though not concurrently.

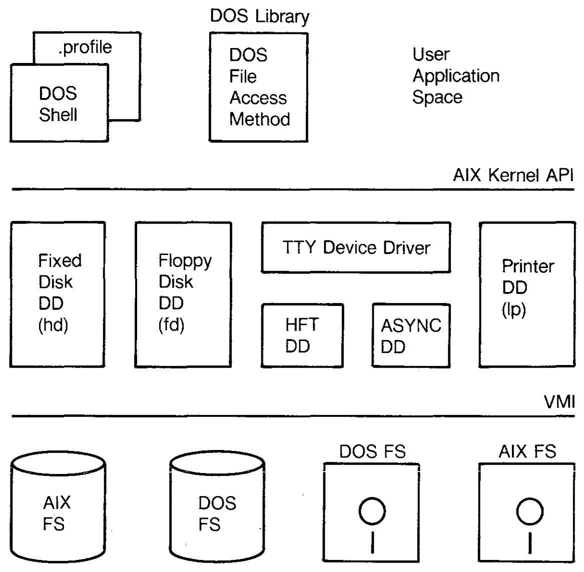


Figure 1 PC DOS Emulation Component Relationships

Application programs written for the RT PC system may be executed from the DOS Shell. These programs may be stored in either AIX or DOS file systems. The system searches for these programs starting in directory /usr/dos, then in the user's current directory, and then in each directory specified by the PATH list.

The DOS Shell was implemented as a separate shell, rather than using the "link" command, because of the difference in semantics between the DOS Shell special characters (? and *) and the AIX shell metacharacters. The DOS Shell parses the command line input following PC DOS rules and passes the parsed/expanded parameters to RT PC AIX programs in the conventional AIX format of argc, argv values.

The RT PC DOS File Access Method
Many existing PC DOS applications create data files in the form of data bases, spreadsheet models, and various forms of text documents. In an attempt to build upon the existing customer base, the need was recognized for easy migration of that data to the RT PC, as well as easy use of PC application data in a mixed PC and RT PC environment. Although file conversion utilities are provided for some files, it was felt that many applications may want to use the same format of data on both PC's and on the RT PC.

The DOS File Access Method (DFAM) consists of a set of library routines that allow applications to access both DOS and AIX file systems transparently. The application program interface contains those functions available in the UNIX file access method. However, these functions have been generalized to allow access to either DOS or AIX files. The syntax of these routines is identical to corresponding AIX system calls. Hence, existing applications can be easily converted to use DOS file systems.

DFAM uses the concepts of path names and a current directory to determine the location of a specified file. If a specified file resides in the AIX file system, calls are converted to AIX file system calls. If the files reside in the DOS file system (either on a DOS diskette, or on a DOS minidisk created by the coprocessor), DFAM interprets the DOS directory structure and retrieves data from the DOS file system.

DFAM resides above the kernel. No changes were made to the AIX file system access method in order to implement DFAM. This was done to assure portability of existing applications that use the current UNIX file systems.

As was mentioned previously, a number of environment variables have been established in the AIX environment for use by DOS emulation functions. DFAM uses these environment variables to determine the binding of DOS device names to AIX directories or system devices, as well as to aid in the interpretation of file and path names received by the various library routines.

Automatic conversion of ASCII files is provided through the use of a DOS environment variable DOSFORMAT. If a file is opened explicitly as an ASCII file, each read or write to that file results in the conversion of the data to either DOS ASCII or AIX ASCII, depending upon the value of the DOSFORMAT variable. This conversion deals primarily with line ending characters (NL, CR, LF).

DOS file attributes are closely emulated by the DFAM library routines. The "hidden" and "read-only" file attributes are emulated if the file resides in the AIX file system. This was done using analagous attributes in the AIX file system. The "system", "volume" and "archive" attributes do not directly map to AIX attributes, and are therefore, not directly supported.

Some Limitations
Some of the DOS functions have not been implemented. The current design allows for the possibility of adding those functions, if necessary.

DOS file sharing became available in DOS Release 3.1. This was too late in the RT PC development cycle to be included in Release 1. File sharing allows concurrent access to the same file by multiple processes. At file "open time", an application specifies whether or not other processes may have write, read, or read/write access concurrently.

Coprocessor applications run independent of applications executing on the native RT PC processor. No provision has been made for access concurrency by coprocessor applications and RT PC applications.

Summary
A high degree of PC DOS compatibility has been achieved at the user interface level. This should allow PC DOS users to easily migrate to the RT PC. The system behaves like a multi-tasking DOS system from an end user's perspective. PC data files move easily into and out of the RT PC, allowing users flexibility in their data processing environment.

Authors

Hira G. Advani

Engineering Systems Products, Austin, Texas

After receiving an MS in Computer Science from Georgia Institute of Technology, Mr. Advani joined the IBM Office Products Division at Austin, TX in 1978. He worked on several software projects related to OS/6 and Displaywriter. He is currently a Development Programmer and is Operating System Extensions manager for the IBM RT PC.

David C. Baker

Engineering Systems Products, Austin, Texas

Mr. Baker is an advisory programmer. He received the Bachelor of Science in Electrical Engineering from New Mexico State University in 1972 and the Master of Science in Engineering from the University of Texas at Austin in 1979. He joined IBM in 1976 and was involved in microprocessor and systems architecture and design. Mr. Baker worked on the system design of the Displaywriter. After a temporary assignment to the IBM T. J. Watson Research Center at Yorktown Heights, Mr. Baker returned to Austin where he worked on the advanced technology effort leading to the startup of the RT PC development project. Mr. Baker currently serves as chief programmer for local terminal support.

Ronald J. Barnett

Engineering Systems Products, Austin, Texas

After joining IBM in 1965, Mr. Barnett served in a variety of staff and management positions in the areas of quality and industrial engineering. He worked in quality and reliability positions on NASA Space and Navy Sonar programs before moving to Austin in 1978. In Austin, he has managed quality and industrial engineering areas for IBM's 5520 Administrative System, Displaywriter, and RT PC products. Mr. Barnett holds degrees in Industrial Management and Industrial Engineering from Auburn University.

J. R. (Bob) Barr

Engineering Systems Products, Austin, Texas

Mr. Barr joined the Federal Systems Division of IBM in Owego, NY in 1960, where he had design and test responsibilities on several major government contracts. He received a BS degree in Electrical Engineering from the University of Missouri in 1960 and an MS degree in Electrical Engineering from the University of Arizona in 1964. He is currently an advisory engineer in the Microprocessor Design group.

Charles W. Bartlett

Engineering Systems Products, Austin, Texas

Charles Bartlett is a project engineer and manager of Manufacturing Test for the IBM RT PC. He joined IBM in 1981 after receiving a BSEE from the University of Texas at Austin. He received an Outstanding Technical Achievement Award in 1983 for the design of printer test equipment.

John M. Bissell

Engineering Systems Products, Austin, Texas

Mr. Bissell was the team leader for Data Management Services, and vendor technical interface for the SQL/RT LPP. Mr. Bissell joined IBM in 1972 after receiving a BSEE degree from MIT. Prior to coming to Austin, he worked on software for the Space Shuttle Launch Processing System at Cape Canaveral, Fl. He currently manages the Data Management Services department.

Leonard F. Brissette

Engineering Systems Products, Austin, Texas

Leonard Brissette is an advisory engineer with the Engineering Systems Architecture department. He came to IBM from Eastman Kodak Company in 1978 as a senior associate engineer in the Office Products Division at Austin, TX. Before joining the RT PC development group, Mr. Brissette worked on various Austin word processing products in the areas of diskette and data stream design and applications. His work with the Advanced Applications Development group involved data base and LAN applications. Mr. Brissette holds a BSEE from the University of Texas at El Paso.

Bertram E. Buller

Engineering Systems Products, Austin, Texas

Mr. Buller is an advisory programmer in the Hardware Architecture group for the RT PC. He joined IBM in 1956 in Kingston, NY as a diagnostic programmer for the SAGE system. He has participated in the development of the 1410 operating system, OS/360, System/23, and Displaywriter. He has held management assignments in the FAA Air Traffic Control, 3705 Emulator, and 6670 SNA communications projects. From 1972 to 1976, Mr. Buller provided technical guidance on SNA for a number of major application projects, including Credit Lyonnaise, Mid-Atlantic Mastercard, and the State of California. He received a BA degree from Gettysburg College in 1950 and has done graduate work at Syracuse University.

A. V. (Tony) Burghart

Entry Systems Division, Austin, Texas

Mr. Burghart is an advisory engineer responsible for the design and implementation of a new manufacturing process. He joined IBM in 1965 in Endicott, NY. He transferred to Austin in 1973. His career experience at IBM has been in the development and support of manufacturing processes, both component manufacturing and product manufacturing. He has held positions in both engineering and management. He received an IBM Outstanding Contribution Award for his work in developing a chemical recovery process. He holds a patent for a circuit line repair tool. Mr. Burghart received his BS in Electrical Engineering from Wichita State University in 1965.

Nancy A. Burns

Engineering Systems Products, Austin, Texas

Mrs. Burns is a senior associate programmer for IBM. She is currently a technical member of the team developing an expert system to perform diagnostics on a machine with limited memory. Mrs. Burns is a PhD candidate at Southern Methodist University. She received a BS in Statistics and Quantitative Methods at Louisiana State University and an MA in Mathematical Sciences at the University of North Florida. She is a member of the Association for Computing Machinery and its Special Interest Group on Artificial Intelligence, the American Association for Artificial Intelligence, and the Association for Computational Linguistics.

Albert Chang

IBM Research Division, Yorktown Heights, New York

Dr. Chang is manager of systems development in the Advanced Minicomputer Project of the IBM Research Division which he joined in 1965 after receiving a PhD in Electrical Engineering from the University of California at Berkeley. His current interests are in systems programming and computer architecture.

Ahmed Chibib

Engineering Systems Products, Austin, Texas

Mr. Chibib joined the IBM Research Division in Yorktown in 1965. There he worked on computer assisted instruction and symbolic execution of programs. In 1977, he joined the OPD Architecture group in Austin and participated in the early PL.8 compiler work for ROMP. He is currently a member of the PL.8 Tools group for IS&CG.

Roy A. Clauson

Engineering Systems Products, Austin, Texas

Mr. Clauson is currently the manager of the Engineering Systems Architecture department for the IBM RT PC. During the development of the RT PC he managed one of the operating system departments responsible for porting UNIX to the RT PC. Mr. Clauson joined IBM in the Boston, MA, branch office in 1970. He worked on testing 3270 Display products, various communications networks, and the System/370-135 while in Kingston, NY. In 1975 he joined the 5256 printer development group in Rochester, MN and subsequently worked on the 5280 Distributed Data System I/O Subsystem. In 1980 Mr. Clauson transferred to Austin, TX, where he has held management assignments for 5280 follow-ons, Office Systems Interconnect Architecture, and IBM RT PC software development. Mr. Clauson received a BSEE from Northwestern University in 1970 and an MSEE from Syracuse University in 1979.

Raymond A. DuPont

Engineering Systems Products, Austin, Texas

Mr. DuPont joined IBM in April 1964 at East Fishkill, NY after receiving a diploma from the Penn Technical Institute. He held various assignments in test equipment maintenance, bipolar circuit design and manufacturing engineering. He received a BS degree in Physics from Marist College in June, 1974. He transferred to Austin in 1976 and has worked in the circuit technology area. He received a MSEE degree from the University of Vermont in 1982. He is currently a senior engineer and manager of the AESD Advanced Circuit Technology department.

Charles K. Erdelyi

General Technology Division, Essex Junction, Vermont

Mr. Erdelyi was born in Hungary in 1938 and came to the U.S. in 1957. He joined IBM as a Customer Engineer in 1958 and, with the exception of military and educational leaves, has been with the company since. He holds a BSEE and MSEE from MIT and an MSEE from the University of Vermont. He has spent most of his engineering career in circuit design activities. He is currently a senior engineer responsible for CMOS macro development.

Gregory C. Flurry

Engineering Systems Products, Austin, Texas

Mr. Flurry joined the Office Products Division of IBM in Lexington, KY in 1973. He worked in Advanced Ink Jet Technology and on an electronic typewriter using that technology. He spent two years at the IBM Research Laboratory in Yorktown Heights, NY working on various projects related to office workstations. He transferred to Austin in 1980 to work on the RT PC virtual terminal subsystem. Mr. Flurry received a BS in electrical engineering from Vanderbilt University and an MS in electrical engineering from the University of Kentucky. He is currently an advisory programmer in RT PC system architecture.

C. P. (Chuck) Freeman

Engineering Systems Products, Austin, Texas

Mr. Freeman is a staff engineer assigned to AESD Memory Management Design. He joined IBM in 1976 and the ROMP project in 1980. He designed a portion of the ROMP nodal model and the Memory Control portion of the MMU memory management chip. Mr. Freeman has had principal responsibility for MMU since 1983. He received a BS degree in Electrical Engineering from Tennessee Technological University in 1975 and an MS degree from the University of Texas at Austin in 1981.

Willie T. Glover, Jr.

Engineering Systems Products, Austin, Texas

Mr. Glover is a senior associate engineer in Advanced Microprocessor Design. He joined IBM in 1980 after receiving a BS in Electrical Engineering from the University of Tennessee, Knoxville. Mr. Glover is a member of Tau Beta Pi, Eta Kappa Nu, and the Institute of Electrical and Electronics Engineers.

Mark S. Greenberg

Engineering Systems Products, Austin, Texas

Mr. Greenberg is a senior programmer who has been working on the design and implementation of the VRM since 1982. He joined IBM FSD at the Cape Canaveral Facility in 1965 after receiving a BS in mathematics from MIT.

Carolyn Greene

Engineering Systems Products, Austin, Texas

Ms. Greene is a staff programmer and is currently technical assistant to the manager of the AESD Systems Extensions group. She was the architect responsible for the externals of the Usability Services package. Ms. Greene received a BS in Information and Computer Science from Georgia Institute of Technology in 1980. After two years of graduate study on user interface design, she joined IBM in 1982. Ms. Greene has worked on several software projects for the Displaywriter and the IBM RT PC, doing design, modelling, and evaluation of user interfaces.

Randall D. Groves

Engineering Systems Products, Austin, Texas

Mr. Groves is a staff engineer in the Advanced Microprocessor Development group in Austin. He joined IBM in 1979 in Manassas, VA and transferred to the MMU chip design team in Austin in 1982. Mr. Groves received BS degrees in Electrical Engineering and in Business Administration from Kansas State University in 1978 and 1979. He is a member of Tau Beta Pi, Eta Kappa Nu, Blue Key, and Phi Kappa Phi.

G. Glenn Henry

Engineering Systems Products, Austin, Texas

Mr. Henry is an IBM Fellow and is the manager of Hardware and Software System Development for the IBM RT PC. He joined IBM in 1967 in San Jose, CA, and has been involved in the design and management of the IBM 1800, IBM System/3, the IBM System/32, and the IBM System/38. He has received several formal awards, including an IBM corporate award in 1982 for his work on the System/38. Mr. Henry received a BS and an MS in mathematics in 1966 and 1967 from the California State University at Hayward, and is a member of the ACM and IEEE.

Phillip D. Hester

Engineering Systems Products, Austin, Texas

Phil Hester is a senior engineer and manager of Hardware Architecture for the IBM RT PC.

His area's responsibilities include hardware architecture, performance, compatibility, and follow-on requirements. He joined IBM in 1976 after receiving a bachelor's degree in Electrical Engineering from the University of Texas at Austin. His previous experience includes design of various portions of ROMP, lead engineer for MMU, and management of the Microprocessor and Memory Management department. While at IBM, he received an MS degree in Engineering from the University of Texas in 1981.

Harrell Hoffman

Engineering Systems Products, Austin, Texas

Mr. Hoffman joined IBM in 1976 at Houston where he was involved with programming the space shuttle onboard computers. He worked as a programmer on the 5520 Administrative System in Austin before moving to his current position of staff engineer with the advanced microprocessor group. Mr. Hoffman has a MS in Computer Science and BS degrees in Electrical Engineering and in Mathematics.

John T. Hollaway

Engineering Systems Products, Austin, Texas

Mr. Hollaway is the manager of the AESP Workstation Development group. He has development and product engineering responsibility for RT PC. Mr. Hollaway joined IBM in 1964 as a junior engineer at Lexington, KY. He worked on such projects as the Tape Transmission Unit, communication controls for the MC/ST, the Small Business Terminal, and Communicating Mag Card I. Mr. Hollaway entered management in 1972. His management assignments include Dictation Systems, OS/6, communicating Mag Card, and Displaywriter I/O and RAS. Mr. Hollaway received a BSEE from the University of Missouri at Rolla in 1964, and an MSEE from the University of Kentucky in 1966.

M. E. Hopkins

IBM Research Division, Yorktown Heights, New York

Martin Hopkins is manager of compilers in the Advanced Minicomputer Department of the IBM Research Division. He spent ten years with Computer Usage Co., a software house, mainly working on compiler development. In 1969 he joined the Research Division. Since then he has worked on computer architecture as well as compiler development and language design. In 1985 he received a Corporate Award for his work on the PL.8 language and compiler. Mr. Hopkins has a BA in Philosophy from Amherst College.

John W. Irwin

Engineering Systems Products, Austin, Texas

John W. Irwin is a senior engineer in Advanced Engineering Systems Development. He joined IBM as a Customer Engineer in 1956 after serving as a USAF jet fighter pilot, then transferred to the Poughkeepsie Laboratory in 1958. He participated in the design of Hypertape I and II, the 2415 Tape Unit, and the 2803 Mod I and II Tape Controllers. He received an IBM Corporate Award in 1974 for development of the GCR recording method and an IBM Eighth Level Invention Award in 1985 in recognition of 24 patent applications and 28 patent publications.

Jerry Kilpatrick

Engineering Systems Products, Austin, Texas

Jerry Kilpatrick joined IBM in 1968 after receiving a BS in Mathematics from Centenary College. He initially worked as a Systems Engineer in the Shreveport, LA Branch Office. In 1969 he took an educational leave of absence to do graduate work in Computer Science at the University of North Carolina at Chapel Hill. After receiving his PhD in 1976, Dr. Kilpatrick returned to IBM at Austin. He was the User Interface Design manager for the RT PC.

Rajan Krishnamurty

Engineering Systems Products, Austin, Texas

Rajan Krishnamurty, a staff programmer in Austin, TX received a BSEE degree from the University of Houston, 1976, and a MSEE degree from the University of Texas, 1983. After joining IBM in 1976 at Austin, TX he worked on the media attachment hardware on the Displaywriter from 1979 to 1982. After working on a number media proposals for follow-on products, he joined the RT PC program in 1983 where he was responsible for development of the PC AT coprocessor services licensed program product. Mr. Krishnamurty holds two U.S. patents and has had nine articles published in the IBM Technical Disclosure Bulletin.

Thomas G. Lang

Engineering Systems Products, Austin, Texas

Mr. Lang is a staff programmer for the Advanced Engineering Systems Development group, where he works on design and implementation of the VRM. He joined IBM in 1978 in Rochester, MN, after receiving a BS in Computer Science and Electrical Engineering from Michigan State University. In Rochester, he worked on SNA communications programming for the S/32 and S/34. He transferred to Austin in 1980, where he was involved with operating systems development on the 5280 Data Entry System before joining AESD.

S. A. Lerom

Engineering Systems Products, Austin, Texas

During the development of the RT PC system, Ms. Lerom managed the AIX Configuration department. She currently is Technical Assistant to G. G. Henry. Ms. Lerom received a BA in Mathematics from the University of Minnesota. She joined IBM in 1976 in Rochester, MN, and has worked in communications software development on S/3 CCP, S/34, and 5280. She transferred to Austin in 1980 and entered management in 1981 as manager of a SNA Communications department.

Larry Loucks

Engineering Systems Products, Austin, Texas

Larry Loucks is a member of the IBM Senior Technical Staff and is the lead architect of the RT PC system. He received a BA in Mathematics from Minot State University in Minot, ND. Larry joined IBM in 1967 in the Fargo, ND branch office. In 1970 he transferred to Raleigh, NC, where he worked on QTAM, TCAM, and SNA. In 1977 he transferred to Austin, where he has worked on the 5520 and the RT PC.

Alan MacKay

Engineering Systems Products, Austin, Texas

After receiving a BS degree in Computer Science from Brigham Young University, Mr. MacKay joined the IBM Office Products Division at Lexington, KY in 1974. There he worked on Software Development Tools. He received his MS in Computer Science from the University of Kentucky in 1977. In 1977, he joined the OPD Architecture group in Austin and participated in the early PL.8 compiler work for ROMP. He is currently a member of the PL.8 tools group for IS&CG.

F. T. May

Engineering Systems Products, Austin, Texas

Mr. May joined IBM as an associate engineer in Lexington, KY in 1961, with BS and MS degrees in Electrical Engineering from the University of Kentucky and the University of Tennessee. In 1968, he moved to Austin to head a new development laboratory in conjunction with the release to production of the Mag Card I. In 1973, he was promoted to director of the Office Product Division engineering organization in Lexington, KY. He was vice president of OPD development from 1973 to 1976 and then returned to Austin as vice president of office systems with responsibility for Austin manufacturing and development. He was director of the Austin laboratory from 1977 to 1978. Mr. May was the IBM RT PC Workstation Development Manager and is currently responsible for future workstation development.

Peter E. McCormick

General Technology Division, Essex Junction, Vermont

Mr. McCormick received a BSEE degree from Worcester Polytechnic Institute in 1965 and a MSEE degree from Michigan State University in 1967. He joined IBM in 1967 at East Fishkill, NY, transferred to Manassas, VA in 1970, and to Essex Junction, VT in 1978. His work experience has been in bipolar and FET circuit and chip design spanning SSI, LSI, and VLSI digital integrated circuits. Development projects have included masterslice, master image, and custom chip designs. He is currently an advisory engineer in the Microcomponent Design area of the IBM General Technology Division.

T. L. Mothersole

Engineering Systems Products, Austin, Texas

Ms. Mothersole received her BA in Computer Science from the University of Texas at Austin in 1978. She spent 3 years at Motorola Semiconductors in the Computer Aided Design group, working on engineering tools for circuit analysis and logic simulations. She joined IBM in 1982 in the Displaywriter Display Support group. After working on several screen management research projects, she joined the design and implementation team building coprocessor terminal support in the VRM.

Tom Murphy

Engineering Systems Products, Austin, Texas

Mr. Murphy is a senior programmer. He received a bachelor's degree in Education and a master's degree in Computer Science from the University of Wisconsin. Mr. Murphy joined IBM in 1967 and has been involved with language implementation on System 3, 5100 Series, and 5280. Currently he serves as lead programmer for the usability package.

Khoa D. Nguyen

Engineering Systems Products, Austin, Texas

Mr. Nguyen is an advisory engineer in the system design department of the AESD group, where he worked on the design of the virtual terminal subsystem. He joined the IBM Office Products Division at Austin, TX in 1975 and worked on hardware and software projects related to OS/6, ROMP, and Displaywriter. He received a BS degree in Electrical Engineering and Computer Sciences from the University of California at Berkeley in 1974 and a MS degree in Electrical Engineering from the University of Texas at Austin in 1980.

Jack E. Olson

Engineering Systems Products, Austin, Texas

Jack Olson is a senior programmer with the Advanced Strategic Products Development Group in ESD. He joined IBM as an associate programmer in 1969 at the Application Development Center in Des Plaines, IL. Before coming to Austin he was in the DB/DC Development Group in the Santa Teresa Laboratory where he worked on the CICS and IMS projects. In Austin he was a member of the 5520 Administrative System development team before joining the RT PC development group. He holds a BS in Mathematics from the Illinois Institute of Technology and an MBA in Operations Research from Northwestern University.

John C. O'Quin

Engineering Systems Products, Austin, Texas

Jack O'Quin joined IBM in 1977, after receiving a BA in Computer Science from the University of Texas at Austin. He began doing operating system work while employed by the University Computation Center. Prior to joining the RT PC project, he worked on the IBM 5520 operating system. After assisting in the initial bringup of the RT PC prototype hardware, he worked at improving the code produced by the C compiler. This led to redefining the subroutine linkage interface. He also was active in developing the virtual memory paging supervisor in the VRM.

John T. O'Quin

Engineering Systems Products, Austin, Texas

Mr. O'Quin is an advisory programmer who has been working on the design and implementation of the VRM since 1982. He joined IBM in 1977 in Austin, TX after receiving a MSEE degree from Georgia Tech.

Mukesh P. Patel

Engineering Systems Products, Austin, Texas

Mr. Patel received his MSEE degree from Utah State University in June 1966 and joined the General Electric Company in Lynchburg, VA, in the Communication Division. In June 1968 he joined IBM in East Fishkill, NY where he worked on bipolar device characterization and modeling. He transferred to Manassas, VA in 1971, where he worked on a variety of bipolar and FET circuit designs and chip design systems. He continued to work in this area until his transfer to Austin, TX in 1978. He has been a key member of the design team on the ROMP and MMU chip designs and has defined the design system and methodology and directed their implementation. He is a senior engineer in the Advanced Microprocessor Development Function.

P. T. Patel

Engineering Systems Products, Austin, Texas

Mr. Patel joined IBM in Manassas, VA in June 1973 upon receiving a MSEE degree from the University of Connecticut. He worked in various bipolar circuit design activities in Manassas. He transferred to Burlington in 1978 where he worked in the I$^{(2)}$L circuit technology. He transferred to Austin in 1980 and has worked in the area of VLSI design. He has been the lead designer on the memory management chip and made numerous contributions to the design methodology. He is currently an advisory engineer in the Advanced Microprocessor Development Function.

Sheldon L. Phelps

Engineering Systems Products, Austin, Texas

Mr. Phelps is a staff engineer in Advanced Engineering Systems Development with RT PC System Architecture. His primary work on RT PC is on the I/O Channel Architecture. He joined IBM in 1969 at San Jose working on the System/3 file system. In 1972 he moved to Rochester, MN Development Laboratory to work on processor development for the 3657 Ticket Unit. Mr. Phelps received his BSME from Los Angeles State College in 1961 and his MSEE from the University of California at Santa Barbara in 1969. Prior to working for IBM, he worked for the Naval Civil Engineering Laboratory at Port Hueneme, CA.

Mark D. Rogers

Engineering Systems Products, Austin, Texas

Mark Rogers joined IBM in 1982 after receiving a BA in Computer Science from the University of Texas at Austin. He is a senior associate programmer in the Advanced Engineering Systems Development group. On the IBM RT PC he has been involved in the design and implementation of the VRM Virtual Memory Manager.

Ron Rowland

Engineering Systems Products, Austin, Texas

Mr. Rowland, a staff engineer, received a BSEE degree from the University of Cincinnati (1978) and attended graduate classes at the Electrical Engineering department of the University of Texas at Austin. Prior to his present position in memory systems development for the RT PC, he had worked on communication controllers for the IBM 5520 Administrative System, memory systems for the IBM System/36, and memory systems for the IBM Displaywriter. He has authored papers on design for testability, gate array design methodologies, and on various aspects of memory systems design.

Charles H. Sauer

Engineering Systems Products, Austin, Texas

Dr. Sauer received his BA in mathematics and PhD in computer sciences from the University of Texas at Austin in 1970 and 1975, respectively. He joined IBM at the Thomas J. Watson Research Center in 1975. From 1977 to 1979 he was an Assistant Professor of Computer Sciences at the University of Texas at Austin. In 1979 he returned to the Watson Research Center and in 1982 transferred to the IBM Communications Products Division laboratory in Austin, TX. Currently he is Manager of System Architecture for the IBM RT PC. Dr. Sauer has published three textbooks, *Computer System Performance Modeling*, co-authored by K.M. Chandy, *Simulation of Computer Communication Systems*, co-authored by E.A. MacNair, and *Elements of Practical Performance Modeling*, co-authored by E.A. MacNair. He has received an IBM Outstanding Innovation Award for creation and basic design of the Research Queueing Package (RESQ). Dr. Sauer is a member of the Association for Computing Machinery.

Martin S. Schmookler

Engineering Systems Products, Austin, Texas

Dr. Schmookler, who is a member of the Microprocessor Development group in Austin, joined IBM in 1956 at the Poughkeepsie, NY laboratory. There he worked on the designs of many large systems, including Stretch, 7074, 7094, System/360 Models 91 and 195, 3033, and the 3081. He received a BSEE from Pennsylvania State University in 1956, an MSEE from Syracuse University in 1964, and a PhD from Princeton University in 1969 through the IBM Resident Study Fellowship program. In 1976, he was a Visiting Associate Professor in the Computer Sciences department at the University of Texas at Austin, where he is currently an adjunct associate professor. Dr. Schmookler has received two IBM Invention Achievement awards, and is a member of Tau Beta Pi, Pi Mu Epsilon, Eta Kappa Nu, and the Institute of Electrical and Electronics Engineers.

Ed Seewann

Engineering Systems Products, Austin, Texas

Mr. Seewann joined IBM in Austin in 1969 after receiving an MEE degree from Rice University. He has had numerous circuit design responsibilities and his development work includes circuit designs for the Mag Card I, Mag Card II, and System/6. He is currently an advisory engineer in the Advanced Microprocessor Development Function.

Richard O. Simpson

Engineering Systems Products, Austin, Texas

Richard Simpson is a senior programmer in the Advanced Microprocessor Development department of IBM's Engineering Systems Products group, working on ROMP architecture. He joined IBM in 1969 and has worked in several IBM divisions on various projects, including JES2 and the 5520 Administrative System. He has been involved with ROMP architecture since 1981. He holds BA and MEE degrees from Rice University and is pursuing studies for a PhD in Computer Science at the University of Texas at Austin.

Scott M. Smith

Engineering Systems Products, Austin, Texas

Mr. Smith joined IBM in Austin in 1978 working in test tool development for the Systems Assurance function and became manager of test tool development in 1980. In 1983 he transferred to Advanced Engineering Systems Products Development, where he has been involved with floating point accelerators. He received a BSEE from the

University of Texas at Austin in 1969 and an MSEE from the University of Maryland-College Park in 1972. Prior work experience includes Texas Instruments Incorporated and the University of Texas Applied Research Laboratories. Mr. Smith is a member of Tau Beta Pi, Eta Kappa Nu, and the IEEE Computer Society.

T.A. Smith

Engineering Systems Products, Austin, Texas

Todd Smith is a staff programmer. He joined IBM at Austin in 1984. He received a BS in Mathematics and a BS in Electrical Engineering from MIT and an MS in Computer Science from SMU. He is a member of the RT PC Architecture department and worked on the design of the Virtual Memory Manager and other VRM components.

Joe C. St. Clair

Engineering Systems Products, Austin, Texas

Mr. St. Clair is an advisory engineer in Display Subsystem Development. He has been working in the area of display adapter design since 1980. He joined IBM in 1976 after receiving a MS degree from the University of Illinois at Urbana-Champaign. In 1971 he received a BSEE degree from the University of Texas at Austin. He is a member of IEEE, ACM, Eta Kappa Nu, and Tau Beta Pi.

Lee Terrell

Engineering Systems Products, Austin, Texas

Lee Terrell is an advisory engineer in the Advanced Engineering Systems Development group. He joined IBM in 1974 after receiving a BS degree in Electrical Engineering from the University of Texas at Austin. He was technical lead programmer in the AIX Configuration group for the RT PC.

Abraham Torres

Engineering Systems Products, Austin, Texas

Mr. Torres is an advisory engineer working in the Advanced Microprocessor Design group. He joined IBM in 1973 after receiving a BS degree in Electrical Engineering from the University of Texas at El Paso and has done graduate work at the University of Texas at Austin. He worked in MOSFET circuit design and in the Design Automation group supporting Austin's development lab. In 1978, he joined the ROMP group and worked on the logic design of ROMP. He also had responsibility for the ROMP design methodology and testing.

John D. Upton

Engineering Systems Products, Austin, Texas

Mr. Upton is an advisory engineer in the Full Function CPU development area of the Advanced Engineering Systems Development group. He received his BS in Physics in 1971 and a BS in Electrical Engineering in 1977 from Lamar University. He joined IBM in 1977 at Boulder, CO and moved to Austin in 1980. He has been involved in the areas of logic design, simulation, and testing on several microprocessor-based IBM products. He joined the RT PC development group as design team leader for the 6151 system board. He later assumed design responsibility for the 6150 system board as well.

Dick Verburg

Engineering Systems Products, Austin, Texas

Mr. Verburg is a staff programmer. He received bachelor's degrees in Marketing and Computer Science from Michigan Technological University. Mr. Verburg joined IBM in 1978 and has been involved with language implementation on the 5280. Currently he serves as lead programmer for the dialog manager.

Donald E. Waldecker

Engineering Systems Products, Austin, Texas

Mr. Waldecker is a senior engineer and manager of Microprocessor Development. He joined IBM in 1961 in the Federal Systems Division, Owego, NY and transferred to Austin in 1978. Since then he has been involved in the management of ROMP/MMU chip development and system related application of these chips. While in Owego, he worked on development and management of numerous militarized computers, data links, and computer system integration activities. Mr. Waldecker has a BSEE from the University of Missouri — Rolla, and a MSEE from Syracuse University. He is a member of Tau Beta Pi, Eta Kappa Nu, and Phi Kappa Phi.

Frank C.H. Waters

Engineering Systems Products, Austin, Texas

Mr. Waters joined IBM in 1962 after graduating from Oklahoma State University with a BS in Physics. He has worked as a technical writer, programmer, and programming manager on projects such as the 7040/7044, OS/360 Release 1, TERMTEXT, OS/VS1 and VS2, VSAM, 3850

Mass Storage System, 8100 Data Base and Transaction Management System, and AIX user interface design. Mr. Waters is currently an advisory programmer on educational leave of absence from IBM to pursue graduate studies in the cognitive and social psychological aspects of the human use of computers.

T. G. (Tom) Whiteside

Engineering Systems Products, Austin, Texas

Mr. Whiteside is currently manager of AESD Memory Management Design, with responsibility for both the ROMP and MMU chips. He joined IBM in 1982 and worked on the ROMP project for about a year. He then participated in the RT PC product definition as a member of AESD Hardware Architecture until his return to the ROMP area in 1984. Mr. Whiteside received a BS degree in Electrical Engineering from the University of Texas at Austin in 1975. Prior to joining IBM, Mr. Whiteside worked at the Motorola Government Electronics Division from 1975 to 1981 and the Motorola MOS Division Microprocessor Design Group from 1981 to 1982.

Kenneth G. Wilcox

Engineering Systems Products, Austin, Texas

Mr. Wilcox joined IBM in 1957 in Owego, NY where he worked on test and design of Federal Systems Division computers and computer systems for projects such as TITAN, SATURN missiles, and F15 and A7 aircraft. For the past two years he has been the development engineer for the RT PC processor card. He graduated from DeVry Technical Institute in 1954 and has taken undergraduate courses at the University of New York and the University of Texas.

C. Edward Williams

Engineering Systems Products, Austin, Texas

Mr. Williams is a staff engineer for IBM. He is currently a technical member of the team developing expert system diagnostics for a personal workstation. Mr. Williams attended Atlantic Christian College in Wilson, North Carolina. He was on the 5258 Ink Jet Printer Development team and the Displaywriter Diagnostics Development team. He has won three Informal awards and one IBM Means Service award.

Peter Y. Woon

Engineering Systems Products, Austin, Texas

Dr. Woon received his BS in Physics from the University of Toronto, MS in Mathematics from the University of Waterloo, and PhD in Computer Science from New York University. He joined IBM in 1962 as a programmer in the New York Programming Center, and has since worked in research and development divisions in areas such as languages, compilers, operating systems, computer architecture, and advanced software technology. He was manager of OPD Austin Architecture, and then manager of Advanced Microprocessor Software Architecture and Tools. At present, he is manager of Software Technology at the IBM Japan Science Institute in Tokyo. Dr. Woon is a member of Tau Beta Pi and Eta Kappa Nu.

C. G. (Chuck) Wright

Engineering Systems Products, Austin, Texas

Mr. Wright, an advisory engineer in the Advanced Microprocessor Development group in Austin, received a BS degree from Trinity University in 1974, and an MS in Electrical Engineering from Texas A & M University in 1975. For the past several years he has worked in the microprocessor design area, mostly in the design of the bus interface portions of various LSI designs.

George M. Yanker

Engineering Systems Products, Austin, Texas

Mr. Yanker worked as an advisory engineer on the RT PC, first in the Development Engineering group and then as manufacturing engineering's design-for-robotics coordinator between development and manufacturing engineering. He joined IBM in 1964 with a BS degree in Mechanical Engineering from the University of Arkansas and received an MS degree in Engineering Mechanics from the University of Kentucky in 1968. His past assignments were in the development and design of Austin's Mag Card I, Mag Card II, Office System 6, and Displaywriter. He received an Outstanding Contribution Award in 1973 for design work on the Mag Card II and he received an IBM Invention Achievement Award in 1985.

International Business Machines Corporation

Engineering Systems Products
Independent Business Unit
472 Wheelers Farms Road
Milford, CT 06460

SA23-1057-00

SA23-1057
Printed in U.S.A.