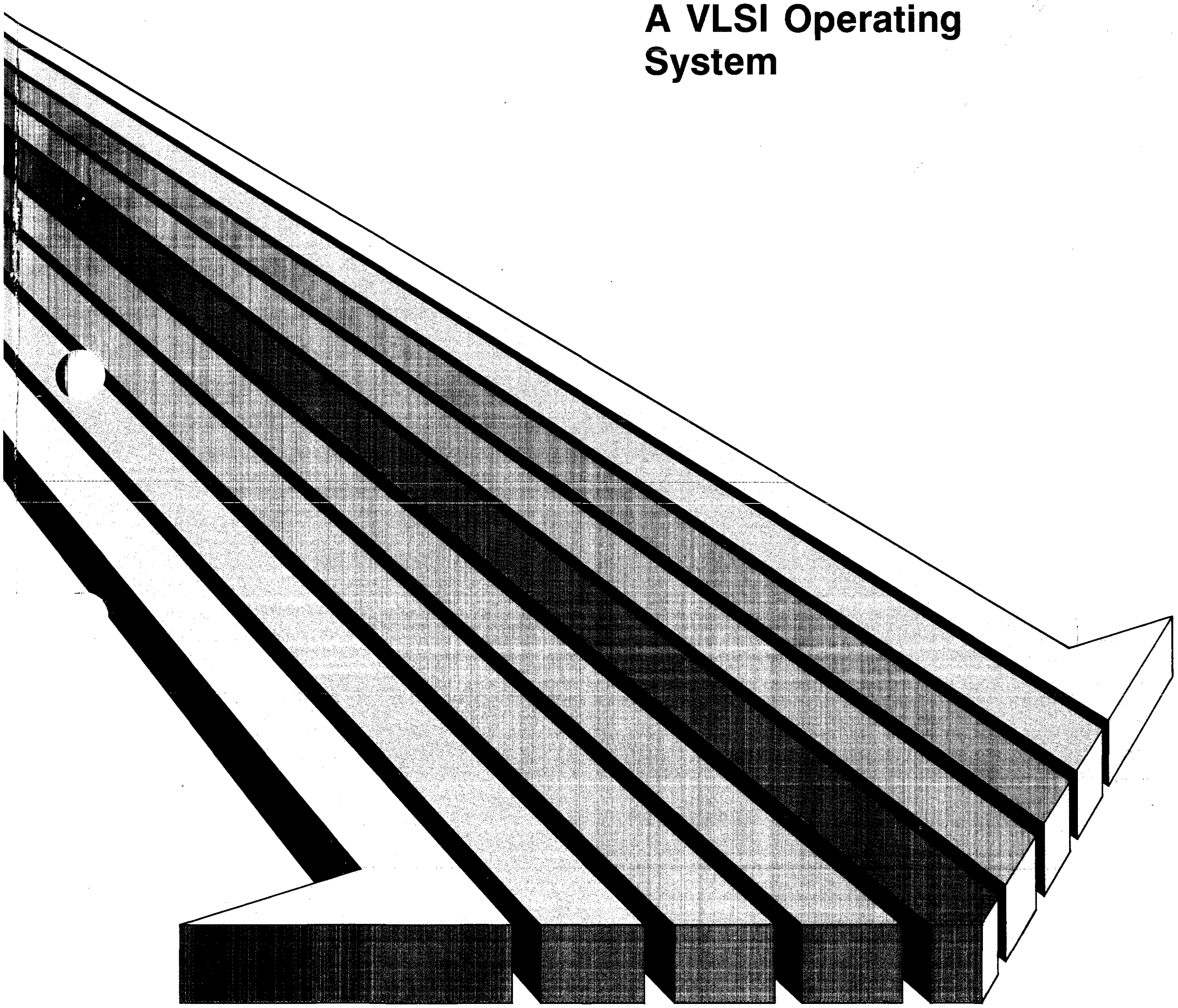
How To Select A VLSI Operating System



How To Select A VLSI Operating System

Contents

June, 1983
210341-004

ARTICLE REPRINT

AR-194

November, 1981

Choosing a VLSI-Compatible System

Peter Palm
Intel Corporation

Reprinted with permission from Mini-Micro Systems, November 1981.

OPERATING SYSTEMS

Choosing a VLSI-compatible system

PETER PALM, Intel Corp.

VLSI offers expanded modularity for operating systems; the intended application will determine the most appropriate system

By integrating more functions into silicon, very large-scale integrated (VLSI) circuitry lowers the cost of μcs and improves their performance and ease of use. But the advent of VLSI raises a question: which μc operating systems enable a user, especially an OEM, to take advantage most quickly of these hardware improvements? This question can be answered by examining the characteristics of available μc operating systems in the light of VLSI advances.

An operating system's intended application is a key consideration in its selection. No operating system is ideal for every application and customer. For this reason, designers tend to optimize operating systems for specific types of applications.

Microcomputer operating systems fall into two categories: systems optimized for efficient development of new software and those optimized for efficient execution of existing software (Fig. 1). Development-oriented systems tend to sacrifice performance to ease of use, while execution-oriented systems make the opposite trade-off. It is tempting to characterize the two types of systems as either end-user-oriented (execution) systems or OEM (development) systems. But such a categorization is inadequate because many end users are concerned with software development, and many OEMs are concerned with software execution.

Development- versus execution-oriented

In the 8-bit μc world, CP/M and Intel's ISIS (Intellec development system) operating systems are good examples of products targeted at efficient software development. In the fast-growing 16-bit μc segment, ISIS (Series III) and the UNIX operating system and derivatives, such as XENIX, fill the development need. Software developed with one of these operating systems is often executed on another machine running an execution-oriented operating system. As more development throughput is required, such as compilations per hour, users move from single-terminal, single-task systems (such as CP/M) to multiterminal systems (such as UNIX or MP/M) and to multiprocessor development systems (such as Intel's NDS-1).

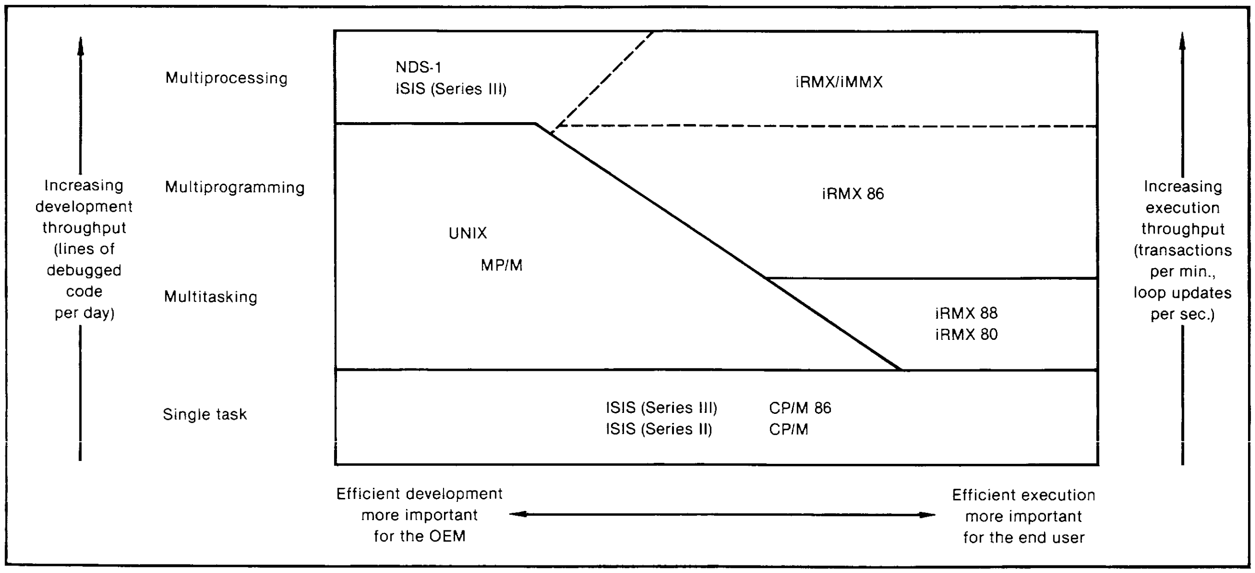

Increasing development throughput (lines of debugged code per day)
Multiprocessing
Multiprogramming
Multitasking
Single task
NDS-1
ISIS (Series III)
iRMX/iMMX
iRMX 86
UNIX
MP/M
iRMX 88
iRMX 80
ISIS (Series III)
ISIS (Series II)
CP/M 86
CP/M
Increasing execution throughput (transactions per min., loop updates per sec.)
Efficient development more important for the OEM
Efficient execution more important for the end user

Fig. 1. μC operating systems *are design-optimized for efficient execution for end users (right) or for efficient program development for OEMs (left). They range from single-task, single-terminal systems (bottom) to multiprocessing systems (top).*

An operating system's intended application is key in its selection.

Most μc systems are execution-oriented and are often most critical for OEMs who must stay competitive in price and performance. Rapid program development is usually secondary to efficient software execution. An OEM can gain a significant competitive advantage by using an operating system that provides faster execution times, less expensive investment, ease of use and the ability to be upgraded.

Operating system vendors, including Intel, have designed a range of products for different needs. CP/M, for instance, is a logical candidate for 8-bit μc applications, in which only single-task execution is required. For 16-bit applications, multiprogramming-type operating systems, such as MP/M, iRMX 86 and iRMX 88, are more appropriate because they take advantage of the 16-bit processor's power. An OEM for whom real-time execution is important might consider iRMX 86. If background program development is crucial, MP/M might be a better choice. For highest performance, users should consider multiprocessing operating systems, such as iRMX/MMX800.

Other selection criteria

Although the intended application is paramount, other considerations are important in selecting an operating system. The overriding factor in light of current trends is probably the ability of a system to keep pace with the impact of VLSI on μc performance and cost. This consideration entails several selection criteria, including transferability, multiprocessing architecture, configurability and interfacing to modules.

First, an operating system should be easy to transfer—at least in part—into VLSI silicon. Putting an operating system into EPROM, for example, can improve speed and lower cost. Vendors should state which operating system functions will be integrated into silicon, and when.

An operating system should provide support for multiprocessing, and VLSI has made multiple-μc systems practical. OEMs should look for operating systems with multiprocessor architectures or other features that ease the move to multiprocessing. Such systems typically include fast context switching, task-to-task communication, synchronization and memory message passing features. For example, the new iMMX800 Multibus message-exchange software allows 8- and 16-bit, master-to-master and master-to-intelligent-slave single-board computers to multiprocess loosely on the Multibus multiprocessor bus.

Modularity is another important criterion in selecting an operating system. The iRMX 86, for example, consists of layers of modules that can easily be moved into silicon as required (Fig. 2). This modular design has enabled Intel to develop a new component dubbed 80130 Operating System Firmware (iOSF) that integrates a timer, an interrupt controller and the iRMX 86 kernel.

An operating system should include standard interfaces to modules. For example, iRMX 86 uses a standard object-oriented format for interfaces to jobs, tasks and message primitives. At a higher layer, iRMX 86 offers standard device-independent interfaces to drivers for the new 8089/8272-based Winchester- and floppy-disk controllers and other device controllers. The interface itself eventually will move into silicon. Only standard interfaces will allow this to occur.

An operating system should also provide industry-standard interfaces to popular program-development languages, such as Intel's universal development interface (UDI) and universal run-time interface (URI). Intel's new UDI/URI-compatible languages, FORTRAN 86, Pascal 86, PLM/86 and ASM 86, can run on any UDI/URI-compatible operating system.

Operating systems should either support or should soon support the leading local-area networks, such as Ethernet, and global-area networks, such as X.25 2780/3780. In November, iRMX 86 will provide the first high-level support for Ethernet, the tri-corporate Digital Equipment Corp., Intel Corp. and Xerox Corp. local-area network standard. Prestigious firms committing to Ethernet include Hewlett-Packard Co., Siemens Corp., Nixdorf Computer Corp., Olivetti Corp. and Zilog Corp. Intel will provide high-level, data-link-layer interfaces to an Ethernet controller (iSBC 550) on the standard Multibus, via the new Multibus interprocessor protocol (MIP). Ethernet will be supported in iRMS 86 via iMMX. These are all standard modules with standard interfaces. ■

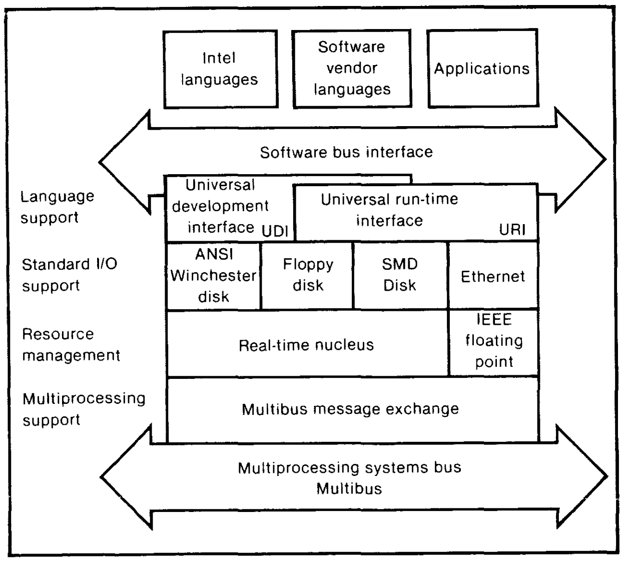


Fig. 2. Layered operating systems *with standard interfaces, standard modules and configurability are key elements in designing μc operating systems to take maximum advantage of lower cost, higher performance VLSI.*

Peter Palm is systems and software product marketing manager, OEM Microcomputer Systems Operation, Intel Corp., Hillsboro, Ore.

December 1981

Microcomputer Operating System Trends

Peter I. Wolochow
Digital Design

SOFTWARE

Microcomputer Operating System Trends

advances in VLSI and other technologies force the development of advanced operating systems

As the use of μCs expands, demand for high-level languages and human machine methods of interfacing will expand in 1982 and beyond. One major component of the expanding demand concerns μC operating systems.

by Peter I. Wolochow

An operating system performs resource management and human-to-machine translating functions. Technically, it is just another computer program — a series of instructions that tell the machine what to do under a variety of conditions. Major operating system functions include management of memory, I/O peripherals, and the central processor.

When computers were first developed, the programs (or instructions) were entered into the machine each time a particular job was started. Only with the ability to store a program in the computer's memory, over 30 years ago, did the concept of computers as we know them today truly emerge. The ability to store a program meant that a computer could perform repetitive tasks while the operator had only to enter information upon which calculations were performed.

Once it became possible to store programs, the development of operating systems began. Operating systems were stored in the computer's memory, and provided the user with a bank of stored computer instructions that could be used with a number of different application programs.

Over the years, operating systems have evolved to the point where they have three main purposes. First, they provide clear, consistent, and easily understood guidelines to users concerning how the machine works. A sub-objective is to provide an easier, more "friendly" human interface to the μC. Second, they perform initialization and start-up functions "automatically" so that — to the user — the machine is ready to perform its basic functions. This initialization function originally included only initial start-up, but now often includes methods to recover from errors in both hardware and software. Third, they provide efficient machine and storage resource management so that different users or programs can

Network Development Systems		↑ Increasing Development Throughput (Lines Of Debugged Code/Day)
Unix	MP/M	
CP/M86	Isis (Series III)	
CP/M	Isis (Series II)	

Figure 1: These microcomputer operating systems are primarily for efficient program development.

Peter I. Wolochow is with the OEM Microcomputer Systems Div. of Intel Corp. Hillsboro, OR.

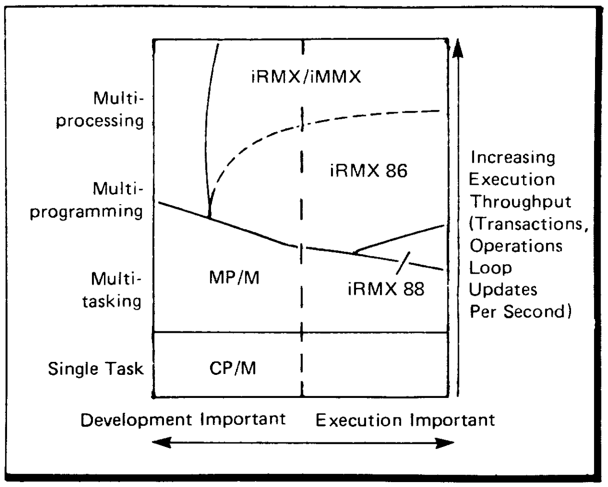


Figure 2: These microcomputer operating systems are primarily for efficient execution.

share the same resources, (e.g., memory, peripherals, I/O ports, a floating point program or the processor itself).

Although virtually all μC operating systems provide users with methods to accomplish these three main purposes, certain operating systems — designed for specific types of applications — have gone far beyond the three common objectives.

One method of viewing μC operating systems has to do with the primary purpose to which the μC is directed. Two major distinctions exist. On one hand, some μC operating systems are used primarily to develop new software systems. In this case, the operating system often includes only features and routines that are useful to persons writing and compiling software programs.

On the other hand, other μC operating systems are directed primarily at efficient execution of software programs for various applications. In this case, the operating system often includes routines and abilities targeted to fast, easy program execution.

software development OS

Within the current world of μC operating systems, examples of those oriented to efficient software development include such products as UNIX (developed by Bell Laboratories) and its related XENIX from MicroSoft, CP/M and CP/M-86 from Digital Research, and ISIS and Networked Development Systems from Intel. CP/M and ISIS-Series II are examples of operating systems designed for the technology and power of 8 bit μC systems. UNIX was designed to match the 16 bit minicomputer, (e.g., DEC PDP-11) and now is being ported (e.g., XENIX) to the newer, more powerful 16 bit μC systems.

Development oriented software systems provide growing levels of programmer support. CP/M, for example, is a single user, single terminal system. UNIX provides support to multiple terminals, so more than one person can be developing software at one time. Intel's Networked Development System provides support for both multiple users over a network, and support for development on multiple processors as well.

execution oriented OS

The second major category of operating systems comprises those primarily targeted to meet the needs of efficient execution of application software already written, developed, and tested. This category includes the largest number of systems, and is often critical to effective utilization of μCs in applications such as industrial control, communications, transaction and word processing, interactive graphics, and simulation.

Often, execution programs working in conjunction with execution-oriented operating systems are developed on a mainframe or minicomputer, with a cross-translator. The other alternative is to develop the software on a μC using a development oriented operating system that has tailored its human interface more to the trained programmer than users of the end product.

In addition to the standard purposes of an operating system — simple human interfaces, initialization, and resource management — operating systems for efficient program execution have a number of other objectives as well. These include real-time operation, multiprogramming, multitasking, multiprocessing, and effective scheduling and priority determination.

real-time operation

Many real-time operations are dedicated to specific applications, such as controlling a machine or series of machines. As such, they often run the same software programs over and over again, depending on inputs received from monitoring and measuring devices attached to the machine they control.

The primary characteristic of such applications is that the data or input that causes the μC to act is not regular, and does not occur at a particular rate. As a result, the μC program and operating system must be able to handle inputs as they occur, and to monitor or control activities based on these real-time inputs.

multiprogramming

This refers to the ability of an operating system to support several independent applications executing concurrently. If a μC, for example, is used to provide an integrated business office system, the software might be divided into a number of separate applications. One might focus on WP, another to manage the printer, and perhaps another to manage an inter-office electronic mail service. Each of these applications would involve a number of tasks devoted to different parts of the system.

An operating system that supports multiprogramming allows this division, and consequently makes it easier to develop the different applications separately, and insure that they do not interfere with each other. This is accomplished by establishing a separate environment for each application. These environments provide the basic appearance of a series of individual machines, while sharing the resources of only one. A multiprogramming operating system must manage this division, keep track of the tasks and requirements of each application, and ensure that each task is given the correct priority.

multitasking

Multitasking refers to the ability of an operating system to effectively manage several tasks, of one computer program, that are operating simultaneously. Multitasking is an important sub-set of multiprogramming, wherein several programs are concurrently being executed. Many applications require that one applications program involve different tasks. Often, these tasks must operate based on data developed in other tasks, and hence a form of task-to-task communication is required. Generally, operating systems designed for efficient program execution require some form of "executive" to manage tasks, priorities, and intertask communication. As a result, common resources such as μP memory and I/O devices can be shared by multiple tasks and can be kept as busy as possible, adding to overall system efficiency.

multiprocessing

Multiprocessing refers to the ability of an operating system to support multiple processors. In certain applications, the demands of the applications exceed the capacity of a single processor or μC. As a result, more processors may be added. When this occurs, the role of the operating system is to efficiently allocate different processing requirements to the various processors, to keep track of which jobs have been sent where, and to assure that the total system resources are effectively utilized. Multiprocessing is becoming more attractive as a way to expand the functions of particular μC applications without the need to entirely rewrite a program.

As μCs become less costly compared to software development and maintenance, many systems will use multiple processors, and operating systems will be required to efficiently manage multiprocessing configurations. A means for linkage of multiple processors and operating systems is Intel's iMMX 800 (Multibus Message Exchange) software and iSBC 550 Erthernet communications controller. They support the needs of local area network applications such as office automation, distributed data processing, factory data collection, research data collection, intelligent terminal and other EDP-related applications.

scheduling and priority determination

As μC operating systems are required to manage even more complex functions — such as multiple programs, multiple

Processor Management:	**iRMX 86**	**UNIX**	**CP/M**	**RSX-11M**	**RT-11**	**MTOS-86**
Scheduling	Realtime	Multiprogram	Batch	Realtime	Realtime	Realtime
Multitasking	Yes (64K)	No	No	Yes (64K)	No	Yes (4K)
Priority Levels	255	None	None	250	None	255
Multiprogramming	Yes (64K)	Yes (64K)	No	Yes (64K)	Foreground /Background	No
Multiprocessor	iMMX	No	No	No	No	Yes (16)
Multiuser	Release 5	Yes	No	Yes (4)	No	No
Interrupt Management	Yes	No	No	Yes	Yes	Yes
Error Management	4 levels	Yes	No	Yes	Yes	No
Powerfail Protect	No	No	No	Yes	No	No
Memory Management:	**iRMX 86**	**UNIX**	**CP/M**	**RSX-11M**	**RT-11**	**MTOS-86**
Dynamic	Yes (1MB)	Yes (64K)	No	Optional	No	Yes (64K)
# of Memory Pools	64K	One	One	8	One	32
Memory Resident	Yes	No	Yes	Yes	No	Yes
Application Loader	Yes	Yes	Yes	Yes (11S: No)	Yes (RT^2: No)	No
Bootstrap Loader	Yes	Yes	No	Yes	Yes	No
(P) ROM'able	Yes	No	No	No	No	Yes
Device Management:	**iRMX 86**	**UNIX**	**CP/M**	**RSX-11M**	**RT-11**	**MTOS-86**
Concurrent I/O	Yes	Yes	No	Yes	Yes	Yes
I/O Buffering	Yes	Yes	No	Yes	Yes	No
Reentrant I/O	Yes	Yes	No	Yes	Yes	Yes
Asynchronous I/O	Yes	No	No	Yes	Yes	Yes
Synchronous I/O	Yes	Yes	Yes	Yes	Yes	~~No~~
Device Independent I/O	Yes	Yes	Yes	Yes	Yes	Yes
Max. # of Drivers	64K		256	256	16	256
Data Management:	**iRMX 86**	**UNIX**	**CP/M**	**RSX-11M**	**RT-11**	**MTOS-86**
File Support:						
1. Sequential	Yes	Yes	Yes	Yes	Yes	Yes
2. Indexed Seq.	No		No	Yes	No	No
3. Direct Access	Yes	Yes	Yes	Yes	Yes	Yes
Swapping	No	Yes	No	Yes	No	No
Overlays	Yes	Yes	Yes	Yes	Yes	No
Hierarchical Directories	Yes	Yes	No	No	No	No
Stream Files	Yes	Yes	No	No	No	No
Mailboxes	Yes	No	MP/M	No	No	No
Critical Regions	Yes	Yes	No	No	No	Yes
Host System For Development	Yes (With UDI)	Yes	Yes	Yes (11S: No)	Yes (RT^2: No)	No

Figure 3: Comparison of microcomputer operating systems

tasks, and multiple processors — the ability of the operating system to schedule activities becomes a primary consideration. In early multiprogramming operating systems, many of these scheduling routines were time driven. That is, one program would be allowed to execute for a certain time. It would then be interrupted, and another program would be allowed to execute. This time driven scheduling, in effect, forced multiprogramming to occur, but also often involved a significant amount of operating system overhead to manage the process.

Subsequently, the approach of event driven scheduling was developed. With this approach, programs and tasks are allowed to proceed until some predetermined event causes the operating system to interrupt the running task and substitute another. In many applications, event driven scheduling is the most efficient manner of allocating resources. In addition, event driven operating systems can often be modified to include some time driven scheduling routines, where the reverse is not possible. As a result, event driven systems are more flexible and can manage the μC and other system resources more efficiently.

future issues

As the μC world continues to evolve rapidly, changes in semiconductor technology will continue to force evolution and improvement in operating systems. As this evolution occurs, a number of trends and developments will influence the user's choice of appropriate operating systems. Some of these developments include:

Very Large Scale Integration (VLSI) Trends. As more complex functions are integrated into μC chips,operating systems will be required to support a broader variety of needs and application programs. The benefits of VLSI — such as increasing density, substantial improvements in function, and rapidly declining costs per function — accrue to those who rapidly use the newest technologies. As a result, μC users on the leading edge often can build significant competitive advantages by being first to market.

With regard to operating systems, trends toward greater VLSI integration imply that operating systems should be evaluated based on their ability to most rapidly use technological advances. This ability to capitalize on VLSI trends includes:

- Operating systems with the potential to be integrated into silicon. Intel, for example, has introduced a device that integrates timers, an interrupt controller, and multiprogramming and multitasking operating system "primitives" into one device. (These operating system functions are equivalent to those of the iRMX 86 kernal.) In essence, some of the traditional software functions will become part of the hardware. Such a development opens a number of vistas for future integration.
- Operating systems organized to take advantage of new trends in μC architecture. One of the most promising developments in this area is object-oriented architecture such as that implemented on iAPX 86 processors under the iRMX 86 operating system and on Intel's new iAPX-432 32 bit micromainframe product family. Essentially, an object-oriented architecture treats different kinds of data and instructions as "objects" and provides common functions that can manipulate objects in a consistent manner. Moreover, the object oriented architecture also meshes closely with the new types of high level languages — such as Ada — now being brought to market. These developments begin to provide μCs designed to optimize both program development and execution.

The benefits of VLSI — such as increasing density, substantial improvements in function, and rapidly declining costs per function — accrue to those who rapidly use the newest technologies.

Standards for μC Languages. μC applications have evolved requiring higher level languages so less technical users can easily participate in VLSI technology. Without substantial advanced planning, the use of higher level languages can cause significant problems with operating systems. One example of how advanced planning is done is Intel's approach. Intel's iRMX 86 operating system contains both a Universal Development Interface (UDI) and a Universal Runtime Interface (URI). These two interfaces provide a standard upon which high level languages can be both developed and run. In essence, the UDI/URI approach is developing a "software bus" or series of standards for easy and consistent language development. Without such an approach to standardization, each new language could be retarded in its development and each new processor could require a completely new set of compilers.

Among the many benefits of this approach are those of particular interest to persons who have been hesitant to use μC because of the lack of standardized, high level computer languages. With the development of the UDI/URI approach in iRMX 86, for example, languages currently or soon to be available include FORTRAN 86, PASCAL 86, PLM/86 and ASM 86.

Another major advantage of software standards now being developed is that many vendors can now create languages that will operate effectively on the same operating system. The software bus concept provides these vendors assurance their languages will operate, and it also provides μC users with a significantly broader range of application languages and even pre-developed software. Among the newer computer languages currently under development by independent software vendors as a result of the UDI/URI standardization are BASIC, COBOL, and "C".

This one development — the standardization of software development and runtime environments — has the potential to be as significant to μC users as earlier efforts to standardize on communications methods (such as the IEEE 488 standard) or μC standardization such as the Multibus (IEEE P796). Once a standard is developed and accepted, many different manufacturers can develop products with the assurance that they will be compatible with other products. Hence, the user obtains a broader selection and applications innovation is enhanced. Moreover, users need concentrate only on learning one approach, with a corresponding improvement in speed and productivity of applications efforts.

Standards for μC Networks. Increasing use of VLSI means that μC costs per function are declining. As a result, many new applications will become cost-effective. One of the major new application areas VLSI is stimulating is local area networking. The costs of μCs and supporting memory are now declining to the point where local and even global area networks make more and more sense. As networking becomes more practical, however, new approaches to networking standards will be required.

Standards are particularly important for networking μC operating systems, since the operating system will be required to manage many of the networking resources. In this regard, Intel Corp, Xerox Corp, and Digital Equipment Corp have joined together to develop Ethernet, a local area network standard. (Many other firms are also becoming involved with Ethernet, including Hewlett-Packard, Siemens, Nixdorf, Olivetti, and Zilog.)

One of Intel's Ethernet responsibilities is to provide standard modules and standard interfaces for Ethernet users. These include high level, data link layer interfaces to an Ethernet controller on the standard Multibus, a new Multibus Interprocessor Protocol (MIP), and a method to support Ethernet with the iRMX 86 operating system in conjunction with Intel's new Multibus Message Exchange (iMMX). Intel intends to provide VLSI implementations of these standards, up to the data link layer. As futher Ethernet standards evolve, the major benefit to networking μC users will be the ability to take advantage of a wide variety of products with the assurances they will work together effectively.

Protection of Software Investment. Many μC users have substantial investments in μC software and are consequently concerned that these investments do not become obsolete before having provided an adequate return on the resources invested. Many current minicomputer users, for example, would like to take advantage of the density, size, and low cost per function benefits of μCs but are hesitant to redevelop their existing application software.

Recent developments in μC software are directed toward solving this concern. The trends toward software and networking standards, for example, will provide a basis for rapid development of new language compilers, and emulators that allow minicomputer users to migrate to more technically advanced μCs without a commensurate reinvestment in software development.

Other trends in μC operating systems, such as increasing modularity and configurability, also support this direction. Intel's iRMX 86 operating system, following this trend, is developed in modules. This allows future integration of certain modules into silicon as conditions warrant. Modularity and configurability also mean users can eliminate portions of the operating system not appropriate to their application without a performance penalty. In addition, Intel's approach to operating system design means that current μC systems users can easily and cost-effectively move most of their existing software from 8 bit to 16 bit μCs as their application requirements expand. This design consideration, most noticeable in the iRMX 86 operating system, guarantees substantial user software investments are protected while users can, at the same time, rapidly take advantage of the latest developments in VLSI technology.

hardware advances mean change

Rapidly advancing developments in μC technology are causing a corresponding change in μC operating systems. More and different operating systems are becoming available as users' needs evolve and become more specialized.

The rapid march of VLSI technology also pressures manufacturers, OEMs, and end users to develop and evaluate operating systems based on their ability to directly translate VLSI advances into applications.

Moreover, the need for a series of operating systems standards is clear. Without standards such as the UDI and URI, operating system development could retard the widespread and fast growing use of productive microelectronics technology. D

intel®

ARTICLE
REPRINT

AR-196

November 1981

The iRMX™ 86 Operating System

Bruce Schafer
and Jack Davis
Intel Corporation

The overall structure of the iRMX 86 Operating System . . . an O.S. designed to exploit the advantages of Intel's VLSI technology.

THE iRMX 86 OPERATING SYSTEM

Bruce Schafer and Jack Davis

INTRODUCTION

Over the past several years, microcomputers have become faster and less expensive. Accompanying this increase in raw horsepower have been enhancements in hardware architecture including high-speed floating-point co-processors, multiple processor support, and soon, Local Area Network controller chips.

This rapid increase in VLSI capability poses a question to microcomputer users: How do they keep up? Already, the cost of developing and maintaining software packages is many times more than the cost of developing the underlying hardware. Fortunately, microcomputer vendors have recognized the problem. Some of the things they can do to help include:

—Provide a wide variety of operating system features as a set of user-selectable building blocks.

—Provide software support for co-processors.

—Provide software that allows the application to take advantage of multiple processors for increased application throughput.

—Provide software that supports the use of the Ethernet* protocol.

—Integrate operating system and hardware functions on the same VLSI component.

—Support standard interfaces that make it easy to move application packages from one operating system to another and to take advantage of future generations of VLSI.

The iRMX 86 Operating System and supporting Intel subsystems provide these capabilities. By taking advantage of them, users can turn the "future shock" of VLSI to their advantage. As such, the iRMX 86 product can be described as a VLSI operating system.

This paper provides an overview of the features of the iRMX 86 Operating System. Associated papers describe how its capabilities are used to support multiple processors [10] and Ethernet [11, 12]. Another paper describes how many of the basic features of the operating system are being integrated with hardware functions [13].

PURPOSES OF MICROCOMPUTER OPERATING SYSTEMS

Reduced Application Investment

Because software costs are rising while hardware costs are falling, microcomputer customers have discovered that they must carefully account for their software development investment. This accounting must include both the original development and the maintenance of the software. Many microcomputer users have found that it is much cheaper to purchase software rather than develop software from scratch. An operating system allows customers to save a significant amount of time and money in developing their particular application program. By reducing their development costs, applications that would otherwise be unprofitable become profitable.

Improved Portability of Applications

Initial costs of development are not the only major concern for microcomputer customers. The cost of developing new products in an existing or new product line is also a key concern. When one microcomputer board is used to create an initial product offering, another member of the same board family is often used to extend the product line. This is true because Intel is constantly adding new features to its family of boards which offer new hardware functions and increased performance. In order to take advantage of new microcomputer components or boards, customers are interested in using as much of their existing software as possible. The use of an operating system will hide many of the details of the underlying hardware and greatly ease moving an application to a new board.

Maximizing Hardware Utilization

Many applications allow monitoring and controlling more than one device. An operating system can make it appear to the application programmer that he has more than one processor at his disposal. This is accomplished by allowing more than one activity to occur asynchronously. The operating system can go further in providing mechanisms for communication and synchronization between programs running on the microcomputer.

Support of Sound Development Methodology

Modular construction is a key way to control the cost of software development and maintenance. Each module ideally "hides" or encapsulates the effect of major decisions

*Ethernet is a Trademark of Xerox Corporation.

such as how data is represented. Similar to structured programming, the modular design process permits a complex design to be partitioned into a structure of cooperating modules. This both facilitates top-down development and simplifies the maintenance and evolution of large software systems. Even though users of microcomputer systems are usually familiar with these concepts, the lack of effective tools and tight development schedules often force them to take a choice between modular construction and quick implementation. High level programming languages have played a key role in allowing customers to obtain the best of both worlds. An operating system can take this one step further by adding facilities to simplify modular implementation. For instance, it can allow the customer to identify separate asynchronous activities that his applications must accomplish and write a separate program for each of these activities. The operating system can allow these separate programs to be run as separate tasks and communicate with each other as necessary. When a particular part of the application is to be modified, the change can be isolated to one or a very few number of the original application programs. When additional functions need to be added, additional tasks can be added to the system with little or no change to the existing tasks.

COMMON FEATURES OF MICROCOMPUTER OPERATING SYSTEMS

The varied purposes of microcomputer operating systems have led to some relatively standard features. Following are some of the features provided by the iRMX 86 Operating System.

Multitasking and Multiprogramming

Based on the goals summarized above, a key role of a microcomputer operating system is to allow for multiple programs to execute on the same processor. This maximizes the utilization of the hardware and at the same time provides for modular construction of applications. Most importantly, it allows the application programmer to manage the complexity inherent in real-time applications where multiple asynchronous events are occurring.

When several programs must execute concurrently (or at least appear to do so), an operating system can provide multitasking. Each executing program is called a task. When these tasks must execute in separate environments, the operating system can support multiprogramming to provide these environments: The set of tasks executing in a separate environment can then be called a job.

Interrupt Mapping

Since most real-time operating systems are event driven, they must use the interrupt structure of the underlying hardware. The operating system can provide the mechanism that transforms the hardware interrupts into events that will cause particular tasks to execute and service the interrupt. In this way each task need be only aware of the interrupt that it is managing.

Timer Support

Real-time applications often require the use of the hardware timer provided by the microcomputer. This may be because they must be aware of elapsed time in order to accomplish their purpose. Another reason is that external events may not occur when they should. The software must be aware that they have not occurred and take corrective action. In either case the programs running as separate tasks may each need a separate timer.

Memory Allocation

Many applications cannot predict their actual memory usage ahead of time. This is because they must deal with an unpredictable environment. A key part of this environment is the operator who invokes various functions of the application. The parameters provided by the operator often affect how much memory the program needs. Besides the operator, the external devices connected to the microcomputer generate a variety of events to which the software must respond immediately. These events are not entirely predictable and thus the amount of memory required to respond to and control these events is not predictable. By providing for dynamic memory allocation, an operating system can help an application adapt to its unpredictable environment without statically allocating the worst-case memory requirements to each part of the application.

Device Support

Often a significant portion of the application development time is spent writing complex code that interfaces the application to vendor-supplied devices. An operating system can provide software that does this interfacing. This feature reduces the customer's development cost and elapsed time.

File Support

Operating systems, in general, use random access devices such as disk drives or magnetic tapes to store and retrieve information. The very simplest of these applications might only store one set of data on each device. Far more common than this, however, is the situation where one disk is to store many different kinds of information. Examples of this information may be parametric information that affects the activity of the application, intermediate data required by the application to accomplish its purpose, and data that the application generates which will be used later.

An operating system must manage the secondary storage space represented by the random access device. This management will usually include support for dynamic allocation of random access space, automatic bookkeeping of this space, and naming each portion used by the application. In this way the application can treat a single random access unit as if it were many separate devices each of which can be randomly accessed.

Loading Support

Many microcomputer applications involve no mass storage devices and thus all of the operating system and application code resides in read-only memory. For systems that include mass storage devices, this allows some of the code to be loaded into read/write memory and provides some significant advantages to the programmer. These include

(1) Software can be updated and distributed on disk or similar media.

(2) If not all parts of the applications must execute concurrently, less total memory may be required if only the code required is loaded into memory.

Human Interface

The vast majority of microcomputer applications involve some interface to a human operator. Many of these applications use standard cathode-ray-tube terminals in order to accomplish this interface. An operating system can reduce the cost of using this interface in three ways:

1) By providing for common editing functions that allow the human operator to correct his input before it is seen by the application.

2) By providing for the automatic invocation of the correct application program based on input by the human operator.

3) By providing functions that make it easy for the application program to determine which parameters have been specified by the operator.

This completes a general description of the features that are provided by iRMX 86. A detailed description of the functional capabilities of this operating system follows.

MAJOR FEATURES OF THE iRMX 86 OPERATING SYSTEM

In order to provide operating system support for applications using the 8086 microprocessor, Intel has developed the iRMX 86 Operating System.[1] Because the system is modular, it allows customers to choose only those features of the operating system that are needed by their applications.

Nucleus

The iRMX 86 Nucleus provides for multitasking, interrupt control, timer support, and intertask communication.[2] While many of these concepts are the same as in other microcomputer operating systems the application interface is quite different. One reason for this is that iRMX 86 interfaces encapsulate the details of the implementation so that it can be more easily changed without affecting the application. This encapsulation is accomplished by implementing each mechanism as a type manager. Each type manager provides a set of objects that are defined by their attributes and the operations that can be performed by them. The basic object types supported are task, segment, mailbox, semaphore, region, job, and extension. One of these objects, the task, also serves as the subject in the sense that tasks are the active elements of the system and perform operations on all objects.

Tasks. All operations are performed by tasks. A task is an executing program and is characterized by a set of processor register values, a priority, a containing job, and a dispatcher state.

Segments. A segment is a contiguous portion of memory described by a base and a length in bytes. The base of a segment can be loaded into a segment register for use as a code segment, stack segment, data segment, or extra segment.

Mailboxes. Mailbox objects are used by a task when it wishes to pass an object to another task in the system. A set of tasks can use this mailbox to implement synchronization, mutual exclusion, and communication.

Semaphores. Semaphores are used in the place of mailboxes where no actual information needs to be communicated between the tasks. Semaphores have the advantage of requiring less execution time overhead and thus may be a sound alternative where performance is critical. They are also useful in solving resource allocation problems, especially when the number of units of the resource is large or if it is desirable to allocate several units at once.

Regions. The region object type is used to implement critical regions via mutual exclusion. Regions have the advantage of preventing the deletion or suspension of a task during a critical operation. They are also used to guarantee that a high priority task will not wait an excessive amount of time for a resource held by a lower priority task that is currently in a region. This is accomplished by raising the effective priority of the task holding the critical region whenever a higher priority task is waiting for it.

Jobs. Job objects represent the environment in which a set of tasks can operate. This environment is limited by the resources given to an application. Several different things can make it desirable to divide an application into multiple environments:

1) Control Dynamic Memory Allocation.

When multiple applications are implemented on a single microprocessor the extent to which each application can account for the other applications' memory needs is limited. By providing separate memory allocation environments for each application, the user can allocate portions of the total memory to each. In this way he can ensure that one application will not allocate memory to the disadvantage of others. This approach also makes it easier to avoid a key hazard of dynamic allocation, the possibility of memory deadlock.

2) Separately Invoke & Abort Individual Applications

While many applications only require a static set of tasks, some applications involve the dynamic invocation of individual subapplications and require the ability to abort these subapplications at any time. By providing separate environments, the iRMX 86 Operating System allows an individual subapplication to be aborted and have its resources returned to the system without affecting other subapplications in the system.

3) Provide Separate Name Spaces

When multiple applications are run on the same microcomputer, these applications are often designed and implemented by separate programming teams. When these teams select names for external devices that are to be used by their applications, they take the risk of choosing names also used by other applications. By providing a separate environment for each executing application, the names used for each application can be kept separate. At execution time the operator can match the individual names used by the application against the resources provided by the operating system. A particular example of this concept occurs when one program is invoked more than one time concurrently. During each invocation the operator can match a set of devices that the application uses against a particular subset of the devices available. In this way the same program can access several sets of devices, at the same time, because from the point of view of the operating system the program is running in separate environments.

Extensions. The final basic object type is the extension object.[9] The extension object is used by operating extensions to create new types. New objects of the new type can be created as a composite of existing objects. Operators are also provided for deleting a composite object and for obtaining the component objects of a composite object.

The concept of restricting access to objects is an integral part of the iRMX 86 model. When a task creates an object all tasks in its job are automatically given access to the object. A task can pass an object to a task in another job. Two mechanisms for communicating access to an object have been built into the iRMX 86 Nucleus: object directories and mailboxes. The advantage of object directories is that they allow a task to obtain access to an object by knowing only its name. The advantage of mailboxes is that they also can be used for synchronization and communication between tasks.

To increase the ease in which programs can be debugged, each iRMX 86 function returns an exception code. This code indicates whether the requested operation was successfully performed, and if not, what went wrong. These exception codes may be handled either by instructions immediately following the call to the operating system or by a separate error handler. In the latter case, an error condition will automatically cause control to transfer to a user-provided routine or, by default, to one provided by the system.

Terminal Handler

The Terminal Handler provides a real-time, asynchronous interface between a terminal and tasks running under the supervision of the Nucleus.[3] It can be used either with or without the Debugger. The Terminal Handler provides the following features:

- Line editing
- Multicharacter type-ahead
- Control characters for suspending and resuming output at the terminal
- A means of awakening the Debugger.

The Terminal Handler can be accessed either directly under the supervision of the Nucleus or through the Basic I/O System described later.

Debugger

The Debugger is designed specifically for debugging and monitoring systems running under the supervision of the Nucleus.[4] A special Debugger is very helpful in debugging such systems because their real-time and multi-tasking characteristics render useless many ordinary debugging techniques. The iRMX 86 Debugger is sensitive to the data structures used by the Nucleus and can give "snapshots" of tasks at critical moments. It can also be used to alter the contents of memory. If desired, the Debugger can be included in a debugged application system for trouble-shooting in the field. If it is included the Debugger requires only the support of the Nucleus and the Terminal Handler.

Basic I/O System

The iRMX 86 Basic I/O System provides facilities for accessing devices and files residing on random access devices.[5] By taking a modular approach, it allows the customer to choose between support for physical devices such as terminals and random access devices, and sophisticated support for files on the random access devices. It accomplished this goal by providing two types of drivers.

The first are device drivers. Device drivers allow the customer to choose from the set of Intel-provided device drivers in order to support the controllers that are being used. In addition, the customer can add his own device drivers to match custom device controllers.

The second type of driver is called a file driver. The file driver accomplishes goals similar to a device driver in that it is a modular piece of the I/O System which allows the customer just those facilities needed by his application. File drivers implement different types of files.

Named files. The most general type of file provided by the iRMX 86 Basic I/O System is that of a named file. A named file is a byte-oriented random-access file which is given a name to identify it among those on a particular volume. Files can serve as both data files and directories. Directories can point to both data files and directories. In this way a file can be designated by a sequence of file names from the root or main directory on a volume through a sequence of directory files to the actual file being designated. Once located, a file can be opened and closed as many times as desired without further directory searches. This type of file naming mechanism is called a hierarchical file structure.

The accessing of the individual data blocks of a file is designed to both minimize allocation of space on a volume and to minimize the number of disk accesses required to access an arbitrary location in a file. For small files an arbitrary data block can be read with a single physical seek-and-read operation. In large files this can be accomplished with at most, two seek-read combinations. Because there is inevitably a trade-off between space and performance, the Basic I/O System allows the application to specify to what extent space should be compromised for performance or vice versa.

Physical files. The second major type of file provided by the Basic I/O System is the physical file type. Physical files are accessed similiarly to physical devices. In order to provide a common interface to a wide variety of devices the interfaces to physical files assumes that any byte on a device can be accessed. The device driver for a particular device then chooses to what extent it supports this file. For instance, a device driver for a line printer would return an error upon a request for seek or read.

Stream files. A third type of file provided by the Basic I/O System is called stream files. A stream file is a sequence of bytes. A task can add bytes to the end of this sequence of bytes and/or read bytes from the front of the sequence of bytes. Any bytes read are consumed by the read operation and thus can only be read once. A key use of string files is to allow a program that normally writes to a physical device or to a disk file to direct its output to another program.

Because the Basic I/O System provides three basic file drivers which are accessed via a common interface, application programmers do not have to be concerned whether the data that is being output is being written to a physical device, a data file, or directly to another program.

Extended I/O System

The iRMX 86 Extended I/O System[6] builds upon the facilities provided by the Basic I/O System and provides the additional facilities to accomplish two general goals:

1) decreasing the cost of implementing applications that use the facilities of the Basic I/O System.

2) providing additional features not found in the Basic I/O System.

The fact that the Basic I/O System is designed to be very general means that some of its system calls are overly complex when used in simple situations. The Extended I/O System provides an optional interface that allows calling sequences to be greatly simplified when some of the more sophisticated features of the Basic I/O System are not needed.

One of the additional facilities provided by the Extended I/O System is the notion of logical names. Logical names allow applications to refer to devices without using the actual physical names of the devices. This allows an application to be written for a standard set of devices and then be executed with a variety of different devices. This accounts for both the fact that the user of an application program will vary over time and the fact that the set of available devices will change over time. The Extended I/O System also extends the concept of logical names to include directory files and data files. In this way an application program can refer to a device location without knowing whether it is using an entire physical device, a directory on that device, or a particular data file on that device. Consequently a data file can be substituted for a device like a line printer or a directory on a large disk can be substituted for a smaller disk.

The Extended I/O System provides support for automatic buffering as an optional facility. This support accomplishes two goals.

1) Allow the physical accesses to the devices to match its physical characteristics. In particular, it allows the number of bytes requested of the device to match the characteristics of the device such as its sector size.

2) Allow the physical access of the device to be concurrent with the execution of the application program. In this way the throughput of the system can be increased by overlapping CPU and I/O time.

The first goal is accomplished by having the Extended I/O System allocate a buffer whose size matches the physical characteristics of the device. Input and output requests are accomplished using the buffer or buffers as intermediate storage. In this way the actual request made to the device controller matches the controller, and is independent of what the application has requested.

The second goal is accomplished by providing one or two buffers where data can be moved to and from at the same time the application is executing. To use the example of sequential input, when the file is open, the Extended I/O System can initiate reads into the buffers that it has allocated for the application. When the application makes its first request, the data it needs may already be in one of those buffers and can be returned immediately to the application. By the time the data in the first buffer is exhausted the second buffer may have been filled. While the second buffer is being used the Extended I/O System can refill the first. In this way, assuming that the execution time required to process a buffer full of information is comparable to the time required to read from the disk, the application will run concurrently with the physical reading of the device. The same approach can be used for sequential output, in this case the physical writing is delayed until the buffer is filled.

By combining the notions of automatic buffer size and automatic overlap, the total throughput of the system can be increased and the execution time of a particular application can be decreased. The Extended I/O System provides both these facilities in a way that makes them appear as if the application is doing a simple sequence of reads and writes. It completely hides the buffering and the overlap algorithm from the application.

Application Loader

The Application Loader uses the I/O System that to load object files into memory[7]. With the loader, you can store some of your code on disk and load it into memory only when you actually need it. This can lower the memory requirements for your application system.

The Application Loader accepts the following types of files:

1) *Absolute Files:* The loader places absolute code into memory at predetermined locations.

2) *Load Time Locatable (LTL):* These files contain code which the loader can assign to any available memory in the job's memory pool. This version of the loader will automatically update instructions as they are loaded to account for the fact that in a multisegment program the code in one segment may refer to code and data in other segments. Since the loader is responsible for allocating the segments, it can update the code with the appropriate base values while it is loading the code.

The Application Loader also supports overlays. This allows an application to be constructed as a "root" and a set of overlays. This significantly reduces the amount of memory required to support large applications.

Human Interface

The iRMX 86 Operating System provides an additional layer of software called the Human Interface.[8] This layer makes it particularly easy for customers to add customer facilities to the system. It is designed to provide support for interactive commands whose code is usually not resident in memory. In addition to this goal it provides a standard set of commands for the manipulation of files.

In order to make it easy to add custom commands to the system, the Human Interface provides for automatically loading and invoking the appropriate program based on the commands entered by the operator. Once a program is loaded and invoked in this way, it can access its parameters by making a series of calls to the Human Interface routines. These routines will return connections to files as well as other parameters.

Rather than require each customer to implement his own set of basic commands, the Human Interface provides a standard set of file manipulation commands. This set of commands includes commands for renaming files, copying files, displaying a list of the files in a particular directory, creating files, changing access to files, and deleting files.

Bootstrap Loader

The iRMX 86 Bootstrap Loader[7] is used to load the iRMX 86 system and/or application programs into memory from mass storage and begin their execution. It consists of two stages.

The first stage provides a rudimentary device driver and uses it to read in the first part of the second stage. In addition, the first stage may provide a file name to the second stage to identify which version of the system is to be loaded. The first stage may also provide for automatic or manual bootstrap device selection.

The second stage reads the rest of itself in and then finds and loads the operating system. Using the device driver from the first stage, the second stage interprets the file structure on the disk. Since the first stage and device driver handle all the device dependent matters, the same second stage can be used on all iRMX 86 disks regardless of the type of device used. Separately located modules may be combined in a library, so that the various layers of the operating system and the application may be loaded from one file.

SUMMARY

This paper has outlined the advantages of vendor-supplied operating systems software for microcomputers. Although these advantages parallel those used for operating systems in minicomputers and mainframe computers, the specific facilities required by a microcomputer customer are often different. For example, many microcomputer applications do not require operating system support for devices and files. The iRMX 86 Operating System answers this need by providing layered software. It is organized around a basic Nucleus that supports multitasking and offers I/O facilities as optional layers above this Nucleus. A customer can choose how many of these facilities he requires and then add his application software on top of the operating system.

By taking this approach iRMX 86 reduces the investment required by the customer, improves the portability of applications from one microcomputer to another, allows one microcomputer to be used for multiple concurrent applications, and provides a model that makes it much easier to change and add features.

CONCLUSION

The dramatic increase of computing power provided by microcomputers represents a challenge. How can this power be harnessed without turning the entire labor force into computer programmers? Part of the answer is provided by vendor-supplied operating systems.

By taking advantage of the availability of both high-performance microcomputer hardware and off-the-shelf software, designers can leverage their expertise and investment. End-users can quickly and effectively put microcomputers to use in their applications. Original Equipment Manufacturers can take advantage of what they know best, their market and their technology. In this way, they can play a *leadership* role in taking the microcomputer revolution to their marketplace.

REFERENCES

1. *Introduction to the iRMX 86 Operating System,* Intel Corporation, 1980.
2. *iRMX 86 Nucleus Reference Manual,* Intel Corporation, 1981.
3. *iRMX 86 Terminal Handler Reference Manual,* Intel Corporation, 1981.
4. *iRMX 86 Debugger Reference Manual,* Intel Corporation, 1981.
5. *iRMX 86 Basic I/O System Reference Manual,* Intel Corporation, 1981.
6. *iRMX 86 Extended I/O System Reference Manual,* Intel Corporation, 1981.
7. *iRMX 86 Loader Reference Manual,* Intel Corporation, 1981.
8. *iRMX 86 Human Interface Reference Manual,* Intel Corporation, 1981.
9. *iRMX 86 System Programmer's Reference Manual,* Intel Corporation, 1981.
10. Ronald M. Smith, Multiple Processor Support with the iRMX 86 Operating System, *Proceedings of the iRMX 86 Technical Symposium,* Intel Corporation, 1981.
11. Jack Inman, Getting onto Ethernet with the iRMX 86 Operating System, *Proceedings of the iRMX 86 Technical Symposium,* Intel Corporation, 1981.
12. Ted Forgeron, Intel OEM Building Blocks Support Natural Addition of Ethernet Capabilities, *Proceedings of the iRMX 86 Technical Symposium,* Intel Corporation, 1981.
13. Phillip L. Barrett, VLSI Operating Systems, *Proceedings of the iRMX 86 Technical Symposium,* Intel Corporation, 1981.

ARTICLE
REPRINT

AR-286

June 1983

Software That Resides in Silicon

Ron Slamp
and Jim Person
Intel Corporation

Software That Resides in Silicon

Ron Slamp and Jim Person, Intel Corporation

Silicon software sounds like a contradiction in terms. The casting of software in silicon implies that the software cannot be changed; yet software does and must change. For example, it must be possible to alter a microprocessor operating system so that the system will support different hardware and software designs, as well as accommodate new hardware components and applications. And if the software has been committed to silicon, then a way must exist to overcome any bugs that are discovered later.

Design Considerations

Silicon software consists of two kinds of code: on-chip code and off-chip code (see Figure 1). In a typical case, some of the off-chip code works closely with the on-chip code, and is developed as part of the silicon software package. This special off-chip (or "support") code might contain initialization, interface, system, and version update codes. For silicon software to tolerate change and be usable in more than one system, the on-chip code must have three qualities: position independence, configuration independence and stepping independence.

Position Independence

Because the most advanced microprocessors address at least 1 megabyte of memory, system software that resides in silicon must work right regardless of its location in memory. Absolute addresses in the read-only, on-chip code or data restricts the configuration of the system. Because the on-chip code recognizes only offsets, absolute addresses are unacceptable. On-chip code cannot presume to know the location of any code or data, it can only presume to know the structure of the data which it accesses. It cannot know, except relatively, where in memory it (or any other code) resides. If the on-chip code is to be position independent, then any absolute addresses needed by the on-chip code must be obtained via the processor's registers.

Position independence is not a new concept; in fact, it is rather an obvious requirement for silicon software. Compilers and relocatable assemblers allow linking and locating, thus making it easier to produce position-independent code. But most of these tools can also produce code that is not position independent. Silicon software developers need to be aware of the position-independence requirement throughout the design, implementation and test phases for their products.

Configuration Independence

The second requirement for silicon-resident software is that the on-chip code must not depend on the underlying hardware and software configuration of the system. Instead, the on-chip code must have indirect access to other code or data, and must then check the run-time data to deduce the system configuration.

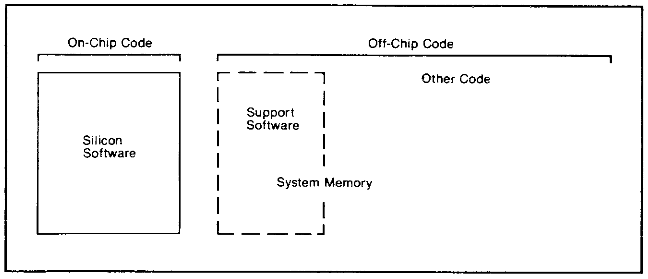


FIGURE 1. Silicon software is divided into on-chip code and off-chip code. The off-chip code either directly supports the on-chip code or contains other applications code.

Because of the read-only nature of silicon software, constants can cause problems when they are located within the on-chip code. Values representing a hardware device must not reside on-chip if that device can be located anywhere in the system, or when values support several devices having similar functions but different programming interfaces. Indirect access is necessary for all values that vary depending on the configuration of the system.

Stepping Independence

Stepping independence is an expansion of configuration independence, and is perhaps the most elusive of the requirements to be met by software intended for residence in silicon. A "step" is an updated version of the on-chip code. The on-chip code and the off-chip code must remain compatible, regardless of changes in either of them. Stepping independence exists when all versions of the on-chip code work with all versions of the off-chip code.

If stepping independence is taken into consideration when the silicon software is developed, then provisions can be made for the subsequent additions of options without changing the on-chip code. Otherwise, the static nature of the on-chip code might make it impossible to add options. Although configuration independence can be designed into software from the start, stepping independence can be achieved only if a system's existing silicon software does not include features that prevent it.

One type of data that is likely to change between steps is the value representing the size of a data area. If the software is to be stepping independent, it cannot know the sizes of the data areas accessed by on-chip code prior to run time. (No problems arise if on-chip and off-chip code agree on the size of the data area.)

But what happens if the on-chip code is not from the same version of the product as the off-chip code, and if the size of the data area has changed between versions? If the size of the data area is defined by a constant in the on-chip code, then that area might be smaller than the off-chip code expects it to be. This misunderstanding can lead to disaster as the off-chip code reads and writes beyond the data area.

This problem is solved when the on-chip code ascertains the size of the data area from off-chip data. Thus, the size of the data areas for the system becomes a configuration option.

Getting the Bugs Out of Silicon Software

Every large program contains bugs. Designers usually remove bugs by modifying the program to correct the problem, and then discarding the old program. However, a program in silicon cannot be modified without stepping the component. And even so, it is undesirable to discard the outdated component.

Software designed for silicon should include a facility for fixing bugs in on-chip code. One way to fix an on-chip bug is to prevent access to the routine containing the bug. A correct version of the routine is provided off-chip, and program execution is forced to branch to the off-chip version whenever the routine is invoked. Modular programming practices during development help reduce the cost of such off-chip duplication.

This on-chip bug-fix works well over time. Each component step has an associated collection of bug-fix modules. The collection is updated for each new version of the product, as component steps fix known bugs. During system configuration, the user specifies which component step is being used; the fixes for that step are included automatically in the off-chip code. Because of this facility, one step looks just like another to the user.

Intel's OSF: A Software Component

The Operating System Firmware (OSF) component consists of several hardware modules (see Figure 2). These modules provide two functions that are essential to operating systems: interrupts and timers. The OSF modules include a Control Store (16K bytes of fast ROM) to contain the silicon software, three programmable interval timers, an eight-input programmable interrupt controller, a bus interface, control logic, a data buffer, and address latch logic.

The 80130: The iRMX™ 86 Kernel in Silicon

Intel's first software-on-silicon product is the 80130. It provides a functional subset of the iRMX™ 86 Nucleus, which is the heart of the iRMX 86 operating system (OS). The iRMX 86 OS is a real-time, multi-tasking, multiprogramming operating system intended for 16-bit microprocessor designs. The iRMX 86 family of standard software modules includes a nucleus, a stand-along terminal handler, a stand-alone debugger, an asynchronous I/O system, a synchronous I/O system, a loader, a human interface, and options required for real-time applications. The nucleus manages the creation and dynamic deletion of all system architectural features (tasks, program environments, memory segments, data-communication managers, etc.). It also schedules tasks, based on priority, interrupt management, memory management, validation of parameters, management of exceptional conditions, and co-processor support.

How the 80130 Satisfies the Silicon Software Criteria

The iRMX 86 Nucleus provides both the on-chip and off-chip codes needed to implement the operating system. The on-chip code resides in the 16K-byte ROM space of the 80130. It is the main portion of the Nucleus code, and includes the kernel of the

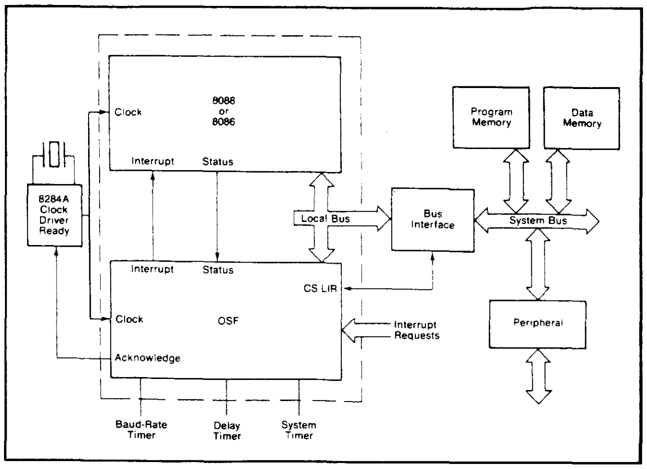


FIGURE 2. The OSF component works with systems that use the iAPX 86, 88, 186, or 188 microprocessor. Close coupling of the CPU and the OSF allows maximum zero-wait-state performance of the OSF software.

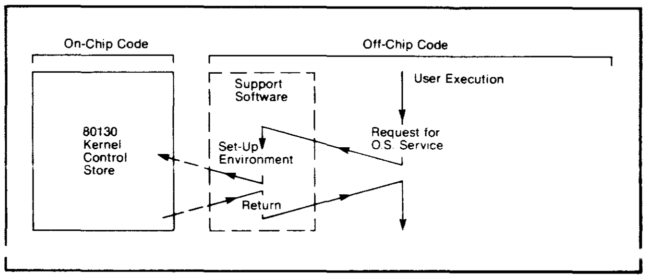


FIGURE 3. The position-independent interface supplies data location and run-time values, and starts on-chip execution of the software.

operating system and the primitives, which are present in the basic 80130 configuration. The off-chip code is stored in external RAM or ROM. It consists of initialization code, and code that either cannot be position independent or cannot be known before a given system is configured.

Position independence is guaranteed if entry to the on-chip code is possible only through an interface in the off-chip code that sets up the necessary registers. The off-chip position-independence interface (see Figure 3) provides an absolute data location and begins on-chip execution by the silicon-resident code. All run-time values can be determined based on the data location. On-chip execution gives the processor a location in the on-chip code from which other on-chip locations can be calculated.

It was relatively easy to make the 80130 configuration independent, because (like most operating-system kernels) it contains only general-purpose functions. The off-chip code contains all the drivers for particular peripheral chips. The Interactive Configuration Utility integrates the drivers with the 80130.

The interface between the off-chip and on-chip codes remains stable across component steps. The stepping-independence interface (see Figure 4) resides on the chip, and is a map of the on-chip code. This interface gives the off-chip code indirect access to all on-chip "publics" (*e.g.*, externally accessible routines, modules, and labels). It is also a chart that routes execution to the proper on-chip location. The off-chip code uses an index of this chart to specify which public should

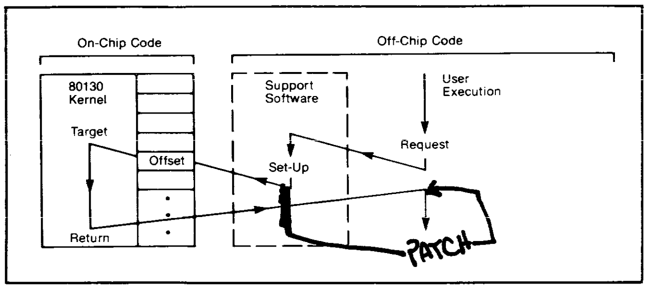


FIGURE 4. All on-chip accesses are routed through the on-chip stepping-independence interface, which provides compatibility between on-chip and off-chip code. Because the interface structure stays constant, the external reference also stays constant, while the on-chip OFFSET changes to point to the new location of the on-chip code.

be accessed. The index of a given routine remains the same across component steps, even though the actual address (offset into the component) of the public has changed. For different versions of the on-chip and off-chip codes to work correctly, all access from outside the component must be routed through the stepping-independence interface.

The 80150: CP/M-86* in Silicon

Intel's decision to implement CP/M-86 operating system in silicon (the 80150) raised a different design problem. With the 80130, Intel only had to deal with Intel-designed software. Code design, implementation, extensions, corrections, support, and the subsequent effect on the end user were all under Intel's control. The selection of an independent software system such as CP/M-86 (a product of Digital Research, Inc.) introduced new factors into the implementation.

The CP/M-86 Architecture

The CP/M-86 operating system consists of three modules. The Console Command Processor (CCP) handles command line processing, and executes built-in utilities. The Basic Disk Operating System (BDOS) performs logical disk I/O, including disk reading and writing, directory management, and sector allocation. The Basic Input/Output System (BIOS), which contains the configuration-dependent code and data, also provides I/O for specific peripheral chips.

CP/M-86 is a single-user, single-tasking operating system written in position-dependent code. The 80150 contains the *entire* CP/M-86 operating system; for many configurations, it requires no off-chip code. Intel's goal was to use the configuration-independent CCP and BDOS elements as a base, and add to them a BIOS that supported a variety of peripheral components but was still configuration independent.

The 80150 BIOS supports the following two functional configuration options:

1. *A preconfigured-mode system*, for which the system designer needs to do no operating-system code development or extension.
2. *A configurable-mode system*, for which the designer makes a selection from among the Intel drivers supplied, and makes changes as required to meet hardware needs.

The 80150 BIOS includes drivers for the following chips:

*CP/M-86 is a trademark of Digital Research, Inc.

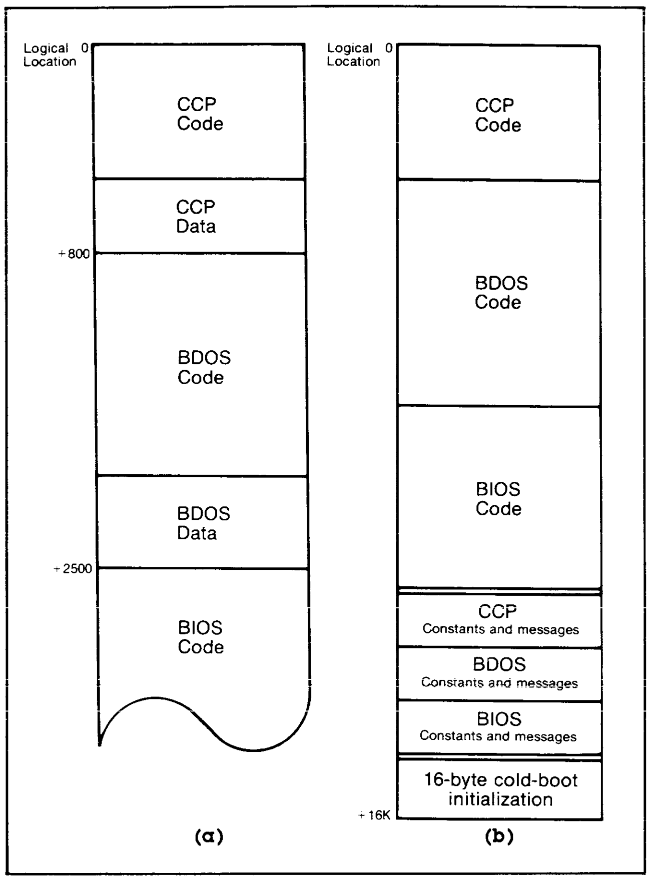


FIGURE 5. (a) The standard disk-based CP/M-86 module is one long structure containing both code and data. (b) Intel reorganized the basic CP/M-86 architecture to fit the operating system into the 80150 OS firmware component.

8251A	Universal Asynchronous Receiver/Transmitter (UART)
8274	Multi-Protocol Serial Controller (MPSC)
8255A	Programmable Parallel Interface (PPI)
8275	Floppy-Disk Controller
8237	Direct Memory Access (DMA) Controller

If the 80150 is used as a co-processor with the iAPX 186 or the 188, then the on-chip peripherals of these processors (DMA, timers, interrupt controller, chip-select logic) are also used.

Configuration independence is achieved via the Configuration Block (CB), with which whole BIOS drivers, data structures, and built-in utilities can be selected independently by the system integrator.

CP/M-86 Transformations

Intel and Digital Research together addressed the issues of position dependence and intermixed code, data, buffers, and stacks. The CCP and BDOS were reorganized to consolidate code and to use the 80150's ROM space efficiently.

CP/M-86 was originally developed using an 8080 model structure. The use of this structure implied that the code and data groups would overlap, as they do in the classical 8080-based CP/M design. Each module contained set-aside buffer areas, and included separate data stacks. Therefore, all variable areas

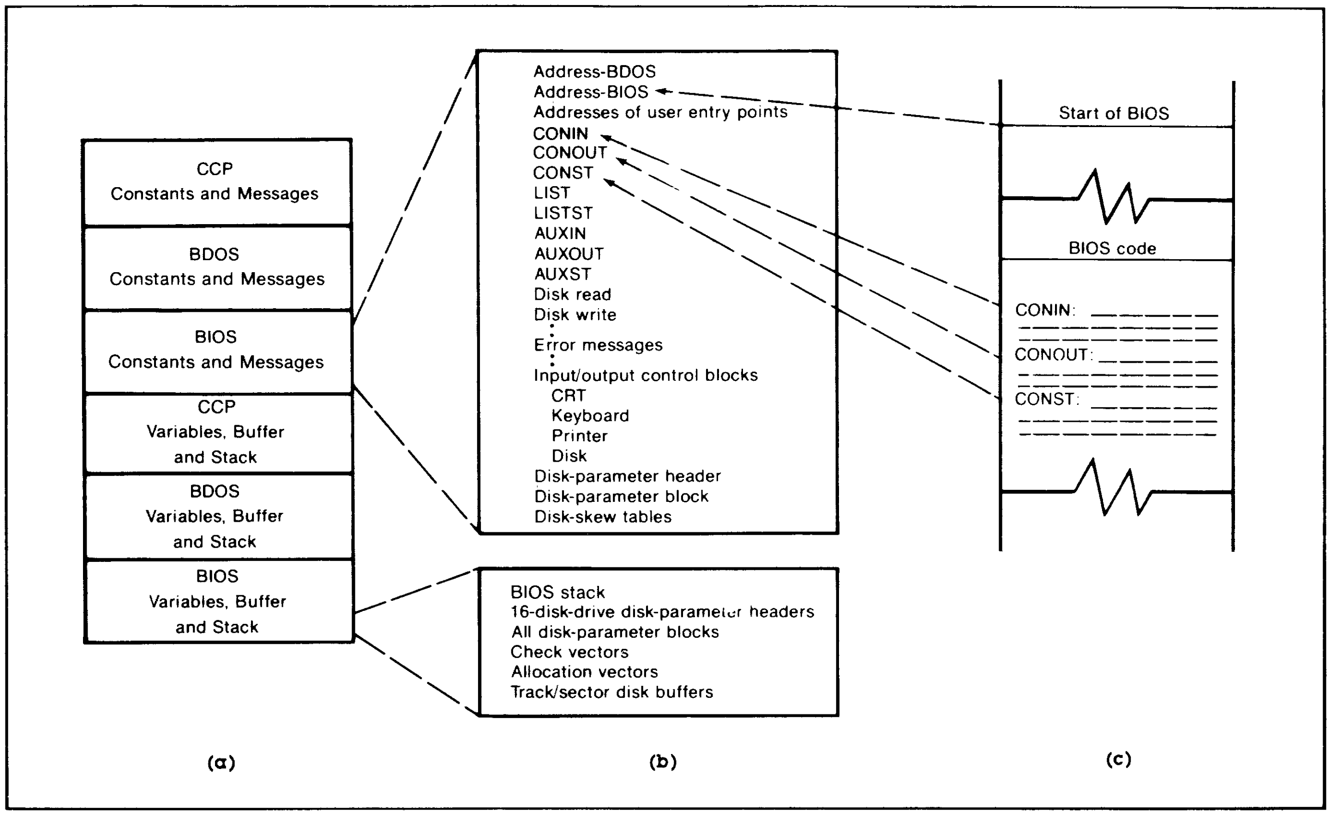

CCP Constants and Messages
BDOS Constants and Messages
BIOS Constants and Messages
CCP Variables, Buffer and Stack
BDOS Variables, Buffer and Stack
BIOS Variables, Buffer and Stack
Address-BDOS
Address-BIOS
Addresses of user entry points
CONIN
CONOUT
CONST
LIST
LISTST
AUXIN
AUXOUT
AUXST
Disk read
Disk write
Error messages
Input/output control blocks
CRT
Keyboard
Printer
Disk
Disk-parameter header
Disk-parameter block
Disk-skew tables
BIOS stack
16-disk-drive disk-parameter headers
All disk-parameter blocks
Check vectors
Allocation vectors
Track/sector disk buffers
Start of BIOS
BIOS code
CONIN:
CONOUT:
CONST:
(a)
(b)
(c)

FIGURE 6. The Configuration Block (CB) reconfigures the 80150 for specific hardware systems. a) The CB contains read down from the 80150, and variables used at run-time. b) The BIOS portion of the CB contains configuration-dependent data. c)These addresses provide access to the 80150 on-chip code, to alter execution paths for different configurations and steppings.

and stack areas had to be removed from code that would reside in ROM.

Figure 5(a) shows the general structure of the original CCP and BDOS. Although a natural separation between code and data is clear, Digital Research did not distinguish between constants, literal messages, and pure scratch storage.

Intel's first step in the transformation of CP/M-86 was to group all variables within each module, including buffers and stacks. We then placed this data grouping at the end of the constants and literal messages for each of the CCP and BDOS modules.

The new structure (Figure 5(b)) includes all code, constants, and internal messages, as well as a 16-byte initial-program- load (IPL) boot resident in the 16K-byte OSF ROM. We removed all variables from the body of CP/M-86, and put them in an external RAM-based structure.

Second, the implementation of CP/M via the Intel 8086 "small model" (separate code and data segments) rather than via the 8080 model (intermixed code and data), meant that the necessary additional variable data space would be available at 80150 execution time. The segmented architecture of the iAPX 86 family made this implementation easy, because separate CPU registers were available for data and code addresses. As part of the BIOS initialization, we moved the constant data structures for the CCP, BDOS, and BIOS to the base of a RAM-resident Configuration Block (CB). An additional amount of RAM equivalent to the total variable space was also allocated and preset to zero. This 8086 "small-model" transformation not only made it easy to separate code and data, but also made the code more efficient and eliminated approximately 2100 bytes.

We achieved configuration and stepping independence via the off-chip RAM-based Configuration Block. Figure 6(a) shows the overall structure of the CB as constructed during BIOS initialization. During initialization, the 80150 BIOS copies the CCP, BDOS, and BIOS constant and literal structures into the Configuration Block, and appends additional space for variable and scratch-pad storage. Even the location of the CB is alterable, based on the address stored in locations 0:3FE-3FF.

Figure 6(b) shows expanded portions of the CB. The data area contains pointers that can be changed to select custom off-chip code instead of the standard on-chip code. The entire BIOS can be replaced. (The BIOS code insert in Figure 6(c) and the various code labels are reflected back to the CB.) Complete I/O control block structures are provided for each CP/M logical device, including CRT, keyboard, list, auxiliary, and disk. The control block includes port addresses, protocol support, and other default data needed to detect and control the status of each peripheral. Figure 6(b) also expands the systems tables and buffers created for disk support.

The addresses in Figure 6(b) indicate how stepping independence is achieved. Any off-chip routines changed by the user can be selected by altering the address of the CB. If Intel updates an on-chip routine, the address in the CB is updated automatically when the 80150 copies its constant structures into the CB. As explained above, full stepping independence is maintained, because any ROM changes can also be implemented off-chip by having the address in the CB point to an off-chip patch. (The CB contains BDOS entry points (shown in Figure 6(b)) that make this change possible.)

The Configuration-Independent Interface

Use of the predefined configuration requires that the 80150 be installed at the top of the 8086 memory address space (FC00:0). The 16-byte internal hardware boot is activated at all POWER ON and hardware resets, and passes control to the 80150. The 80150 initialization sequence uses this positioning to indicate the default hardware configuration (floppy disk, printer port, serial console, or auxiliary port). Each device has predefined port addresses, interrupt assignments, and protocols. The iAPX 186 or 188 CPU supports programmable chip-selection and the on-chip DMA drives the floppy disk controller.

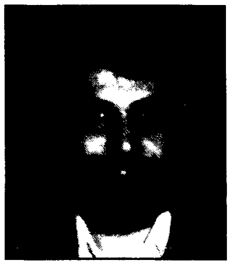
If the configuration must be altered, or if the BIOS code needs revision, the 80150 can be installed on any 16K code boundary except at the very top or bottom of memory. A PROM that contains off-chip code and data for a user's particular configuration is also installed at the top of memory.

The 80150 initializes the default system hardware tables, then calls an EPROM to complete or revise the existing data in the off-chip CB RAM area. At this point, the CB contains the addresses that select either on-chip or off-chip code. When the configuration is complete, control is returned to the 80150. The 80150 completes the CP/M initialization, displaying the familiar CP/M "A" sign-on.

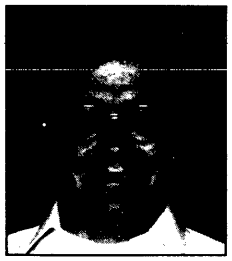
Conclusion

Converting software to silicon is not new. But redesigning software to consist of on-chip ROM code and configurable RAM data is somewhat more innovative. One silicon-related specter that haunts software designers is the fear of "committing code before its time." But software designers can *never* expect to produce bug-free code the first time. And system designers cannot always predict the capabilities or the implementation requirements of peripheral devices that have yet to be built. Nevertheless, software designers who use the general silicon-implementation strategies of position independence and configuration independence, and who provide for stepping independence, can create standard silicon hardware without fear of component obsolescence. □

About the Authors



Ron Slamp received the A.S. degree in software technology from Portland Community College, and gained much of his skill in electronics at Clark Community College in Vancouver, Washington. He has worked in Intel's OEM Module Operation in Hawthorne, Oregon since 1978 and is currently the project leader for component software.



Jim Person received the B.S. degree in mathematics in 1962 from the University of Arizona. He was the engineering project manager at Intel for the 80150 "CP/M-on-a-chip."

SPECIAL REPORT

Punching in for real-time jobs in industry, R&D, and offices, operating systems use special software structures to squeeze better-than-ever performance out of 16-bit microprocessors

by Stephen Evanczuk, *Software Editor*

□ A special class of operating systems is hard at work in the l6-bit microsystem world. For controlling environmental processes, acquiring data at high speed, or even handling transactions at a commercial bank, these operating systems contain mechanisms that enable them to respond rapidly to external events and that differentiate them from the more familiar general-purpose operating systems.

In fact, all the operating systems for 16-bit microprocessors respond in a reasonable period of time. But the general-purpose, or developmental, operating systems like CP/M, Bell Laboratories' Unix, and MS-DOS are intended for standard programming activities like editing, compiling, and file management [*Electronics*, March 24, 1982, p. 113]. As such, they lack certain software structures needed for reliable control of processes producing data at a high speed.

Real-time operating systems tend to fall into two general categories—multipurpose and embedded, reflecting the type of hardware they run on. Multipurpose real-time systems are typically built around full-fledged microcomputer systems with terminal, keyboard, plenty of system memory, and mass storage. Furthermore, in process-control or data-acquisition applications, some special-purpose hardware is usually included in these systems to serve equipment or high-speed data input operations. Besides the familiar applications for research and development, transaction-processing environments are an example of situations needing multipurpose real-time systems.

No doubt the largest class in volume because of their growing use in consumer items, embedded systems are minimal hardware systems, often just one-chip microprocessors that control limited parts of a larger system. Programmers ordinarily employ a special development system to create the software, which is loaded into the target system for use and ideally is never seen again.

To meet the needs of these two classes of applications, real-time operating systems come in three flavors for 16-bit microprocessors. Serving multipurpose real-time systems, one type—discussed in the first part of this report (see p. 106)—includes all the software development support found in their general-purpose counterparts. Furthermore, many can be stripped of the layers needed in the developmental environment and placed in programmable read-only memory for use in an embedded system.

For those who swear by Unix, the group of Unix-based operating systems discussed in the second part (see p. 111) may mean no need to swear at it in real-time applications. A growing number of vendors are starting to convert this admittedly non–real-time operating system into versions that can be used to handle external processes. Although the industry is cautious, if not downright skeptical, of real-time versions of Unix, the fact that C—the language of Unix—is so highly regarded for use in real-time applications may help swing this group into the forefront.

The potential for distributed-control systems based on embedded microprocessors hinges largely on the availability of high-performance real-time operating systems that can be plugged into the application with the same ease as an integrated circuit. Called silicon software, these operating systems discussed in the last part (see p. 114) have been designed to be stored in read-only memory. Providing a fixed set of system calls, they present programmers with a consistent set of high-level commands to perform the low-level functions usually built from scratch.

Building system-level software from scratch has long been the hallmark of real-time programmers, even a mark of honor. Fortunately, however, the increased acceptance of ready-made operating systems using well-understood algorithms (described in the first part) is helping to replace this software "random logic" with rather more standardized packages.

On still another level, the unique responsiveness and throughput demonstrated by real-time operating systems is a truly user-friendly feature. For this reason, these systems should find their way into less obvious real-time applications, such as transaction processing, word processing, and personal work stations for office automation.

Algorithms star in multipurpose systems

□ Whatever environment it finds itself in, the function of an operating system is the efficient management of shared resources by a number of users, whether these are human beings accessing a computer through terminals or programs vying for a single central processing unit. In fact, the degree of sophistication of an operating system is reflected by the number and types of physical resources it manages and by the fineness of control it exercises in their management. And operating systems targeted for control of the external environment must wrestle with the most demanding resource of all—time. The degree of care with which such software is designed to manage time is what determines its suitability for the real-time environment.

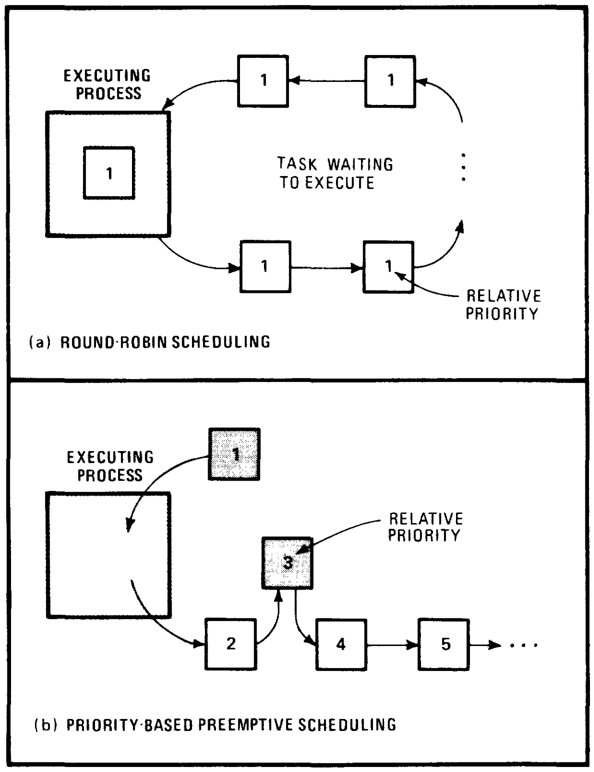
Schedulers and queues

Two critical aspects of the real-time environment are the random nature of physical events and the simultaneous occurrence of physical processes. Consequently, interrupt handling and multitasking are primary attributes of a real-time operating system. In fact, it might be argued that the mechanism for handling multitasking—the scheduler—is the heart of the operating system. The rest of the operating system lies atop this kernel and serves the specific demands of the application environment.

In particular, the lists, or queues, and their managers that surround the scheduler are constructed to deal with the different physical resources supported by the operating system. Thus, one queue may contain those tasks (processes, or programs in the course of being run) that are ready to execute on the processor, another queue may be tasks waiting for access to input/output hardware, and another queue may contain tasks waiting for some specified event to occur.

In any multitasking operating system, the scheduler uses the queues as input. Its output, on the other hand, is a single task that has been activated and allowed to execute on the central processing unit. The scheduling algorithm in large part defines the operating system.

In one system, the scheduler may simply select a task on a first-come, first-served basis, allowing it to run until completion or until some specified period of time has elapsed. This type of relatively primitive algorithm was commonly used in mainframe computers running simple batch-oriented operating systems.



1. Priorities. In round-robin scheduling (a), tasks (or processes) take equal turns executing, while a higher-priority task will supersede a lower-priority one in priority-based preemptive scheduling (b). Most schedulers employ some combination of these techniques.

In a slightly more sophisticated operating system that can be used interactively through terminals, the scheduler may select tasks on a round-robin basis and permit each of them to run for a specified period of time (Fig. 1). Once the task exceeds its time slice, it is placed at the end of the queue and forced to wait until all other tasks have had a chance to execute.

Round-robin scheduling with equal time slices is adequate if every task is no more important than any other task. However, if some are considered to possess a higher priority, then a more sophisticated scheduling algorithm must be used—one that recognizes that some tasks are more important, but that no task should be excluded from using the CPU.

One solution is the use of several queues, where the length of the time slice is related to the priority of elements in the queue. In this case, the scheduler would allow all tasks in each queue of a different priority to execute on the CPU, but lower-priority tasks would be given less time.

A further refinement permits higher-priority tasks to suspend a running task. This technique, called preemptive scheduling, is an important feature for real-time environments, in which the delayed execution of a high-priority task could have disastrous results, rather than simply disappointing the user.

In scheduling algorithms, tasks may exist in a number of logical states, depending on their readiness to run. In the Versatile Real-Time Executive (VRTX) from Hunter

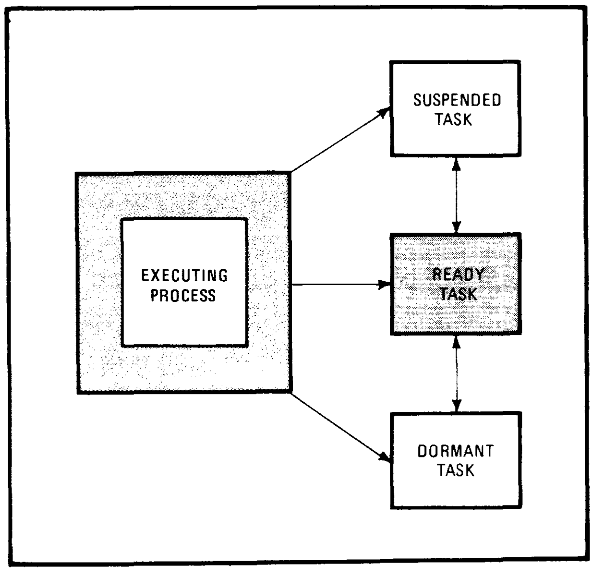
& Ready Inc., Palo Alto, Calif., for example, tasks are driven through four possible states by external events, by other tasks and system utilities, or by their own system calls (Fig. 2). For example, an executing task may delete itself—in which case it enters a dormant state—or may cause itself to be blocked either explicitly through a call to suspend itself or implicitly through a call to perform some I/O function. On the other hand, once suspended, a task may reschedule itself through a system call, or an external real-time event may bring the task back into the ready queue.

Recognizing the importance of scheduler design, at least one software vendor has made it easier for real-time users to build systems around a prepared kernel. United States Software of Portland, Ore., is offering a basic scheduler that assembles into less than 100 bytes of object code for the target microprocessor [*Electronics*, Nov. 17, 1982, p. 206]. Furthermore, in anticipation of real-time systems targeted for specific application areas, U. S. Software supplies a list of design notes detailing extensions to the basic kernel.

Another use for queues

In addition to having queues serving the scheduler directly, most systems use them as the preferred means of associating a task with a required resource. For example, one capability commonly found in real-time operating systems is the ability to suspend a task for a specified period of time. Typically, the operating system contains a special queue for this function. Each element in the queue is a task in a suspended state. Associated with each task is a counter that contains the number of clock ticks remaining until it should be reactivated.

For example, in iRMX-86 from Intel Corp., Santa Clara, Calif., the counters keep track of the incremental time remaining with respect to the previous element in the queue, rather than the total time remaining before the task may be reactivated. Thus at each clock tick only the counter in the element at the head of the queue need be decremented, rather than every counter in every queue element. This method takes longer to insert new elements into the queue and so requires slightly higher overhead for insertion than when the total time is maintained by each counter; however, that overhead is more than offset by the time saved by updating only a single counter.



2. Task states. As one task (or process) runs, others may be in various states of readiness. In Hunter & Ready's VRTX, for example, tasks can be ready (able to run immediately), suspended (waiting for a resource), or dormant (deleted by a system call).

Real-time environments pose a special set of problems for resource allocation. Besides all the more familiar problems of scheduling, a real-time operating system must maintain reliable behavior under extremes of load when it is driven by a high rate of external stimuli. From the system user's point of view, the system must maintain a predictable level of response and throughput.

In an interactive environment, users sitting at terminals measure response as the time the system needs to react to a keystroke. In general, system response is the time that the system needs to detect and collect data from some external stimulus. Throughput, in an interactive environment, is seen as the number of users able to utilize the installation simultaneously. In a more general real-time environment, throughput is the rate at which the system is able to collect, process, and store data.

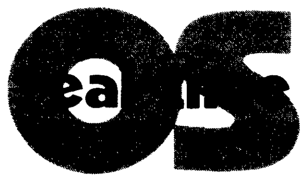
In fact, although response and throughput share some common software elements, operating-system designers will invariably find themselves forced to make choices that will tend to optimize one at the expense of the other. Often, the interrupt-handling requirements of a real-time operating system force this choice.

Interrupt processing is hardware and software integration at its most demanding (see "Handling hardware interrupts," p. 108). To handle interrupts, operating systems often place layers of software between the user and the microprocessor in order to allow different levels of performance and capability.

Intel's RMX-86 is a typical example of distinct levels of software used to perform basic interrupt processing. At the lowest level, an interrupt handler works intimately with the hardware to execute some operation, such as sending a message character by character to a printer. Code for interrupt handlers is kept compact and simple, since system interrupts are disabled during their operation. The higher level, called the interrupt task, works at a priority associated with the particular hardware it services. Interrupt tasks act as interfaces between application tasks, working with specific interrupt handlers to complete execution of operations dealing with external devices. RMX makes this interrupt-handling mechanism available to application programs through a special set of system calls.

Protection and communication

Once the interrupt software has completed its function, tasks that use the data are indistinguishable from any other task in the system as far as the operating system is concerned. Unless special care is taken, conflicts could still arise between two separate tasks that might need to use the same resource, such as the same location in memory. MP/M-86, for example, employs a special queue, called a mutual exclusion queue, that contains a unique message representing the shared resource. In or-



der to use the resource, a task must first capture the message, much as a node in a token-passing network must first obtain the token before being at liberty to transmit.

Per Brinch Hansen[1] identified such shared resources as key elements in multitasking systems. Sections of code that access critical resources are called critical regions. The simple expedient of ensuring that only one task at a time is allowed in a critical region guarantees that multiple tasks may share the same critical resource without fear that its integrity may be compromised when two of them attempt to access it simultaneously (Fig. 3).

This concept of the mutual exclusion of tasks from critical regions is implemented in a structure called a monitor, in which critical regions are gathered in one section of code and protected from use by more than one task at a time. The MSP operating system from Hemenway Corp. of Boston [*Electronics*, Jan. 27, 1983, p. 119] explicitly supports mutual exclusion through monitors in its internal structure.

Furthermore, user-written routines needing monitor protection are provided with four functions in MSP that are implemented using hardware traps for rapid access: Entermon, Exitmon, Wait, and Signal. Entermon and Exitmon serve as monitor entry and exit points, respectively, performing required housekeeping functions. Entermon disables system interrupts and preserves all registers, while Exitmon reverses these actions. Wait and Signal, on the other hand, work in tandem to control access to a critical resource. Wait queues up tasks needing an unavailable resource. Signal releases them from the queue when the resource becomes available.

Wait and Signal are examples of an intertask communication mechanism, called semaphores, found in most real-time operating systems. As noted, these commands simply queue up and release tasks needing a critical resource. Such a resource may be an I/O device, a memory location, or simply a go-ahead signal that synchronizes a pair of tasks. For example, task A may execute only after task B has completed. In this case, task A would begin with a Wait (flag) command, where the flag is used as an associated variable. Task B, on the other hand, would end with a Signal (flag) command. In this way, task A would be blocked until task B had executed its Signal command at the end of its processing. But exchanging simple go–no-go signals is not sufficient for many multitasking environments.

For longer messages, real-time operating systems offer extensive intertask communication facilities called mailboxes. Mailboxes are essentially semaphores with storage. As such, tasks needing data from another task will wait until the other has loaded the mailbox with the information. Intel's object-oriented RMX-86 transfers any of the defined objects in the system through mailboxes. Hemenway's MSP, on the other hand, provides a buffer of fixed size that may be used without restriction on its contents, as long as the 256-byte buffer is not exceeded. With its Multibus message exchange (iMMX) extension to RMX for

Handling hardware interrupts

Underlying the special software of a real-time system is the assumption that the hardware itself can respond in a coordinated fashion to external events, or interrupts. In fact, microprocessors contain subsystems whose sole function is to deal with interrupts in a way that eases integration of the interrupt-handling software.

All modern computers integrate interrupt-handling hardware and software at a very low level of design. When a user accesses a microprocessor through a terminal, the same hardware interrupt facilities come into play as when, for example, an analog-to-digital converter sends data to the same type of microprocessor. The software response, on the other hand, depends on the type of operating system, but both real-time and general-purpose operating systems must take some action, like read in the data value or the character.

Examining the details of a simple keyboard task illustrates the complex nature of real-time processing. It also serves as a vehicle for introducing some of the basic vocabulary in this field.

A standard software subsystem in a microcomputer system, called the keyboard monitor, is responsible for working with the hardware interrupt system to detect a character, collect it, and effect some action based on the input character. When a key is struck on a terminal, the corresponding byte is converted into a serial stream of bits that are passed from the terminal to a universal asynchronous receiver-transmitter. Once it receives the full character, the UART generates a hardware signal, or interrupt, that notifies the processor. Since interrupt management is a common activity, processors contain special hardware to respond to this signal.

Although the details may vary from one particular microprocessor to the next, the result is the same for all. When its interrupt-request line is asserted, the processor ceases its current processing and places values from its internal registers into system memory. Typically, the processor status and instruction-address registers are saved in the system stack, a last-in, first-out buffer located in some portion of system memory. As the figure shows, the processor responds to the original interrupt-request signal by issuing a signal of its own, called an interrupt acknowledge.

The peripheral hardware that originated the interrupt detects the interrupt-acknowledge signal on the system bus and responds by returning the memory addresses of both the interrupt-handling subroutine and the new processor status. Typically, the new processor status will provide for disabling any further interrupts. This latter action is a simple precaution, preventing a single external stimulus from causing a continuous series of interrupts that will eventually result in an overflow of the system stack.

Such an interrupt mechanism, called a vectored interrupt, allows the speediest identification and reaction to an interrupt. (An alternative interrupt mechanism used by earlier processors, called a device-polling interrupt, simply forced the processor to switch to a defined address in memory containing software that polled each peripheral device until the device that generated the interrupt was discovered.) At this point in the interrupt-handling task, all the activity was exclusively in hardware, but nevertheless resulted in extensive processor activity and bus traffic due to multiple accesses of system memory and the involved peripheral-device controller.

Consequently, it is not surprising that the time for hardware to set the processor to handle the interrupt—the hardware-interrupt latency—should be several processor cycle times in length. In general, hardware-interrupt latency is not a fixed number, but will lie within some range, since the processor will need a variable length of time to complete its current instruction and to initiate the interrupt-acknowledge signal. For example, if a processor is involved in a lengthy floating-point operation, several microseconds could elapse before the interrupt is acknowledged.

Once the processor has reached the interrupt-handling subroutine, the contents of only a minimal set of its internal registers have been preserved. However, before the real work of the subroutine may commence, the contents of other registers and variables shared by independent sections of the operating system must be preserved. The time needed to perform this action is called the context-switching time. Only after the software context is switched is the system ready to begin handling the special requirements of the device that originated the interrupt. The period of time between the occurrence of the external event and this state is the total interrupt-response latency.

In real-time operating systems, interrupt-response latency is usually a specified value—around 100 microseconds in very high-performance systems based on 16-bit microprocessors. Designers often bypass the constraints imposed by response latency by including special-purpose hardware to boost system response to external events.

Throughout all this time, system interrupts are still disabled. However, now that the context switch has taken place, the keyboard handler is free to transfer the character from the UART. Deciding where to put the character is important in terms of system throughput and overall efficiency. When it is put in some specified location in system memory, system interrupts must remain disabled; otherwise, if the handler attempted to service a subsequent interrupt, the new character would overwrite the character already in the location, but not yet fully processed.

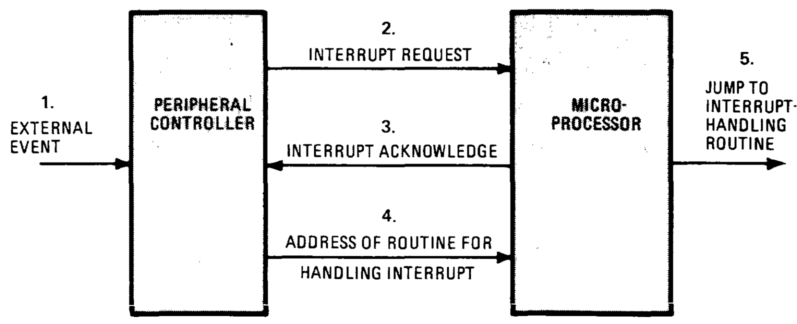
In general, there are two methods for handling this problem. In the first method, the character is simply placed on the system stack and referenced through the relevant-pointer. In an alternative method, the character is placed in a block of memory that has been reserved just for the handler and is called a context block. In this case, the character is referred to by using a specified offset from the base of the context block. Each time the keyboard handler is called in response to an interrupt, one of these context blocks is reserved from available system memory. Setting up a context block and switching the processor to it in a context switch accounts for a significant fraction of the time that is needed to respond to an interrupt.

Software code, such as the UART handler in this example, that does not contain any memory locations for variables is called reentrant because the processor may asynchronously enter it, be called away by an interrupt (even one that results in another call to the same piece of code), and return without loss of data or context. If the code is not already resident in system memory, another routine causes a copy of the code to be read from storage into memory. With reentrant code, only a single copy of the program or task need be resident at any time. Each context block, or logical copy of the task, is called an instance of the task.

Multiple instances of a task help explain some of the confusion associated with performance figures reported as a result of benchmarks. In examining benchmark figures, it should be clear just what the values are that are being reported. Total interrupt latency generally includes hardware interrupt latency, the time to create an instance of a task (plus the time to call in the task into memory if not already resident), the context-switch time, and an additional period needed to execute a variable amount of code that causes the data to be read from the peripheral registers. Creating a new task means either calling in a new task and creating a context block for an instance of it or just creating a new instance of a task already existing in memory.

Once the handler in the UART example reads in the character from the receiver buffer, it will reenable interrupts. The time between entry to the interrupt routine, when interrupts were disabled, until the time when interrupts are reenabled is an important factor in determining the effective latency of system response.

This dead time must be minimized, or the system will remain deaf to external stimuli for unacceptably long periods of time. In fact, the length of time that system interrupts are disabled is one of the criteria for determining the usefulness of an operating system for real-time applications. The longest period during which interrupts are disabled is a direct measure of the responsiveness of the system. Because of the weight of disabled interrupts on total system performance, modern microprocessors use a number of hardware-interrupt levels, or priorities, that disable interrupts at or below the priority level of the device originating the interrupt.



multiprocessor-based systems, Intel replaces the concept of a mailbox with that of a software port connecting different tasks, whether they exist on the same or different physical processor.

Unlike memory-intensive software development systems, real-time environments find less need to support a virtual address space. In fact, the increased system overhead is less than desirable, because the designer seeks to minimize response latency. A useful feature, however, that can be found in some real-time operating systems is a set of system calls responsible for dynamically allocating and deallocating memory.

For example, in the ZRTS system from Zilog Corp., which comes in different versions for the Cupertino, Calif., firm's segmented Z8001 and nonsegmented Z8002, a set of three system calls provides for dynamic allocation and deallocation, as well as information on the status of memory allocation. The system call for memory allocation allows application programs to specify the attributes of the memory block to be allocated and returns a name referring to the created structure.

Besides similar system calls, Intel's RMX adds some calls suited to its context-based architecture. In RMX, each task lies within the context of a job environment that bounds the scope of tasks within it (Fig. 4). As such, each task is allowed to draw from the memory pool of its job. In case more memory is required than that initially allocated to the job, a pair of system calls provides for querying the system on the size of the job memory pool and for dynamically changing it.

Dynamic memory allocation and deallocation is a relatively advanced concept that exacts some overhead during runtime. However, the alternative—static allocation before runtime based on expected requirements—may be less suitable for applications in which the real-time environment is relatively unpredictable.

In real-time operating systems, disk-file management is treated as just another asynchronous task possessing a particular set of critical resources—mass-storage devices. In real-time environments, file-management utilities have to meet not only the requirements of general-purpose systems but some additional demands.

In terms of system response, a requirement of real-time operating systems in heavily loaded systems is the ability to conduct asynchronous I/O operations. In such an operation, the calling task simply queues up the I/O request, then immediately returns as if the task were completed in zero time. When the I/O request is fulfilled, the operating system switches the processor to a separate routine whose address is supplied when the original asynchronous request was made. This completion routine then may continue any processing that may be required following the I/O request.

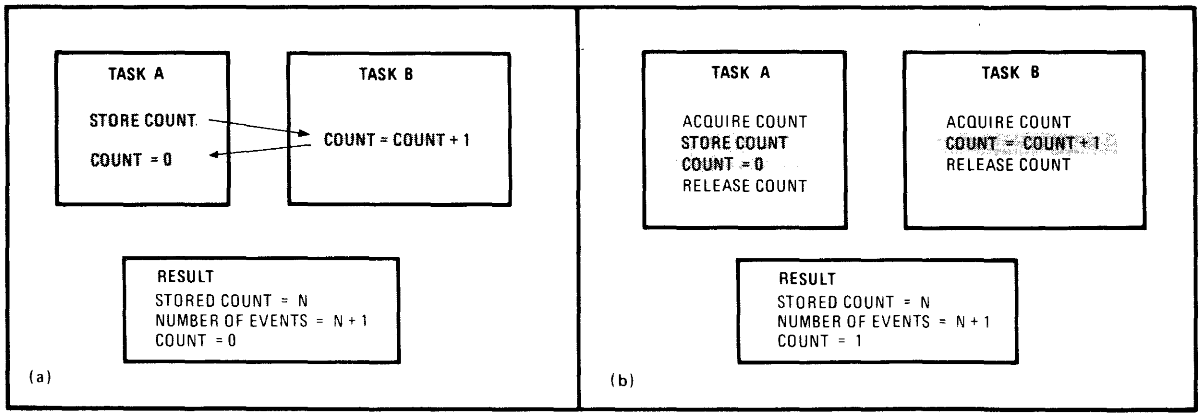
System throughput depends heavily on the efficiency and performance of the I/O subsystem. Peripheral controllers with direct memory access and the ability to move the disk's read-write head without necessarily performing data transfer can significantly reduce the overhead associated with data movement.

Reducing overhead

System software can also contribute to reduced overhead by providing a simple disk organization when high throughput is needed. One of the simplest structures is a file consisting of an unbroken series of disk sectors, such as the contiguous file in Hemenway's MSP or the physical file in Intel's RMX. By ensuring that the next block of data will be written to the next physical sector on a disk, the operating system can reduce the delay caused by head movement on the disk.

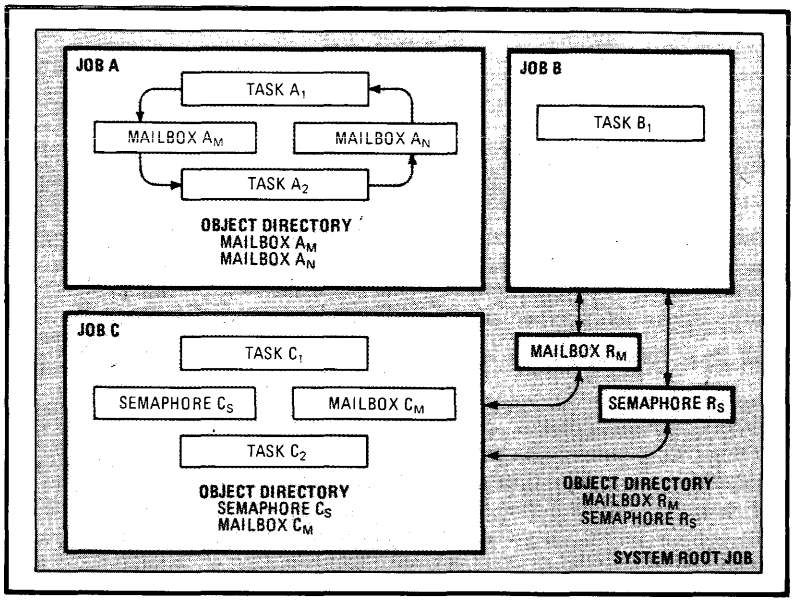
In their use of an I/O interface that is common to all system device drivers, MSP and RMX attack another important aspect of system design, though one not necessarily tied to their utility in real-time applications. In MSP, a basic I/O routine called Iohdlr serves for all operations by accessing a special block of information in memory. RMX, on the other hand, uses a number of device-independent system calls to handle communication with peripheral devices.

Next to multiprocessor-based software systems, real-time software systems are the most difficult to debug. Again, the cause is the distinguishing feature of real-time operating systems—precise management of time. Standard debugging tools for single-user general-purpose op-



3. Critical regions. If two asynchronous tasks use a counter, events can be miscounted if task B interrupts task A before the counter is reset (a). Forcing the tasks to acquire a counter before using it (b) ensures synchronization through the critical regions (tinted).

4. Job context. In Intel's RMX, all jobs exist within the context of another job. A directory defines the objects that are known to other objects in the same context. For example, all three jobs may use mailbox R_M since it is in the system's root-job object directory.



erating systems generally disable all system interrupts in various phases of the debugging routines. Since the object of a real-time software system is asynchronous involvement with the task under control, this effect makes standard debugging tools useless.

Ideally, debugging real-time software would use performance-analysis tools and troubleshooting aids built into the operating system itself. Unfortunately, the processing overhead and additional memory requirements imposed by such a technique make this an unpopular notion in the design of an operating system. However, some systems do provide some means for run-time error handling. The exception handlers in RMX, for example, are procedures that are associated with each task when it is created. If a task attempts to use a system call but encounters an error, called an exception, the operating system invokes the associated exception handler to allow some graceful recovery from the error.

Although the technique in VRTX is not true exception handling, Hunter & Ready's silicon-software system does include a mechanism to build run-time debugging software. A special location in the VRTX configuration table (see p. 115) causes a user-defined routine to be called whenever a context switch is performed. By recording information about the task as well as the processor, such a routine can be used to create a list, called a trace, of the history of task execution.

Because real-time systems often include special-purpose hardware, the accepted technique for debugging user-written routines uses the classical approach of collecting data before and after passing through a suspect region, along with a logic analyzer to monitor timing of traffic through critical regions.

Intel offers some relief to this problem through the iRMX debugger In particular, the debugger allows the user to work with individual tasks without interfering in the operation of other tasks, as well as to monitor the activity of the system as a whole without disturbing it. The debugger recognizes data structures in the RMX kernel, so the user may examine system objects. In addition, Intel's crash analyzer brings mainframe debugging power to microprocessor-based applications using RMX.

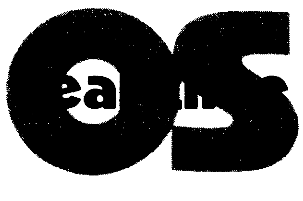
Zilog's ZRTS configuration language offers another level of support to the development of systems targeted to specific hardware complements. By defining the details of the hardware, a system designer can configure ZRTS to particular systems.

Reference
1. Per Brinch Hansen, "Operating System Principles," Prentice-Hall, Englewood Cliffs, N. J., 1973, p. 84.

Designers tune Unix for real-time use

☐ With an eye on the growing momentum of Bell Laboratories' Unix, real-time system designers have endeavored to squeeze this complex operating system into the rigid confines imposed by the demands of real-time environments. Although Unix brought advanced system capability to mini- and microcomputers, the original intent was to provide a hospitable software-development environment, rather than to include the features considered necessary for real-time uses.

Until now, data-acquisition systems employing unmod-

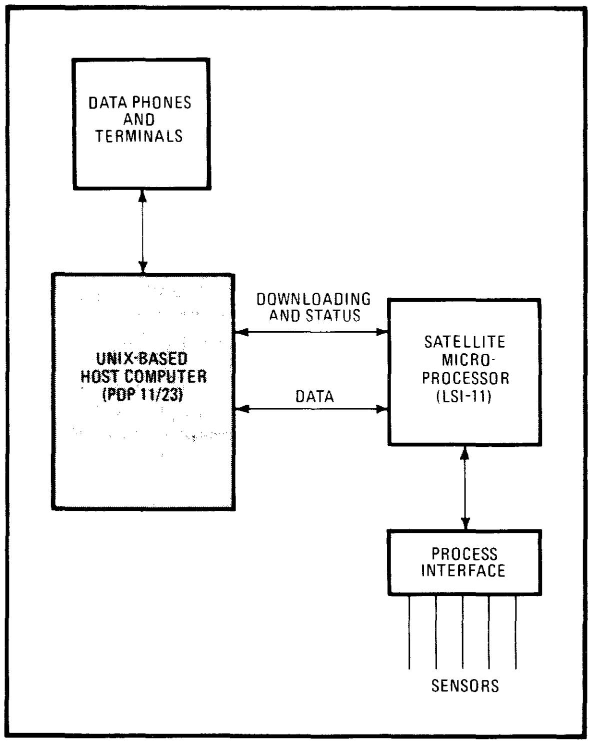


ified Unix typically used dedicated microprocessors to buffer a central computer from constant random activity caused by external events. For example, in the Conceps process-control system from Bell Laboratories, Murray Hill, N. J., a Unix-based host is linked with auxiliary microprocessors. In each microprocessor, software derived from Unix software handles the low-level details of real-time activity (Fig. 1).

Unix goes real-time

Appearing in all shapes and sizes, Unix-compatible executives, Unix lookalikes, and new Unix versions are bringing this popular environment into real-time applications. However, unlike their colleagues creating totally new operating systems (see pp. 106–111), designers of these second-generation systems are constrained by the boundaries set by the original. Caught between Unix's complex organization and the high-speed needs of some real-time applications, they have opted for preserving the basic architecture. Still, for intensive data-acquisition applications, vendors like VenturCom, Cambridge, Mass., and Masscomp, Littleton, Mass., add on dedicated hardware like high-speed peripheral controllers to link devices into the main system without losing the generality of the Unix software architecture.

For microprocessor-based dedicated systems, memory-resident kernels like the C Executive bring a measure of Unix compatibility to even dedicated systems. Offered by JMI Software Consultants of Roslyn, Pa., the C Executive combines support of an extensive C-language run-time library with many of the features considered important in real-time applications. Although not directly supporting shared data in its multitasking architecture, the executive's intertask-communication facilities include data exchange through a queuing mechanism. As befits a real-time executive, the task-scheduling algorithm allows higher-priority tasks to preempt lower-priority ones. Because it is intended primarily for embedded systems—that is, dedicated microsystems that do not have disks—the C Executive is totally contained in system memory and does not support the extensive Unix file-management subsystem.


DATA PHONES AND TERMINALS
UNIX-BASED HOST COMPUTER (PDP 11/23)
DOWNLOADING AND STATUS
DATA
SATELLITE MICRO-PROCESSOR (LSI-11)
PROCESS INTERFACE
SENSORS

1. Satellite processing. In Bell Labs' Conceps system, separate microprocessors handle low-level details of process control. Yet another processor—a host computer that runs the Unix operating system—is in charge of coordinating these satellite machines.

Controlling real-time tasks

Full-blown Unix lookalikes, on the other hand, find themselves forced to deal with some of the very internal structures that aided Unix's rise in popularity. For applications like program development where regular scheduling is more important that instant response, scheduling is aided by Unix's manipulation of the priority levels of tasks (or processes, in Unix's preferred terminology). For real-time applications, however, the slight uncertainties this feature introduces could destroy the synchrony of timed events controlled by the system.

Consequently, one enhancement commonly found in the real-time offshoots is the addition of some mechanism to ensure more precise control of real-time tasks. A technique that sits well within Unix's task-oriented (that is, process-oriented) design is the definition of a real-time class of tasks (or processes). This class earns special rights in the operating system, such as a guarantee that each task will not be swapped out of memory, but remain locked in and ready to respond more rapidly to events.

VenturCom's Venix, for example, defines a real-time priority level. The scheduler allows tasks running at this level to maintain control of the processor for as long as necessary. In contrast, Regulus from Alcyon Corp. of San Diego, Calif., speeds response to real-time events through the use of 32 user-defined priority signals.

Better I/O handling

In addition to its scheduling algorithm, Unix's method of handling input/output operations needs improvement to perform well in real-time applications. Aiding total system response, the asynchronous I/O procedure in Venix supplements the conventional synchronous procedure in Unix, in which the requesting task must be suspended until the I/O operation is completed (Fig. 2). By placing asynchronous requests at the head of the I/O request queue, Venix's manager lets real-time tasks issue a write request, for example, and immediately continue processing, assured that the request will be honored next.

Concentrating instead on improving what happens when I/O requests have been completed, Masscomp's enhanced version of Bell Labs' Unix System III adds a modified signal called an asynchronous signal trap. Similar to the concept of completion routines in other operat-

Going Forth with alternatives

Few nightmares evoke the feelings of dread experienced by a programmer who must alter code that has been developed by another programmer—worse yet if the code is all assembly language for an embedded system. Fortunately, system developers are seeing the light of day and are specifying one of the commercially available real-time operating systems, so programmers now are dealing with a set of well-defined software calls for system functions. Still, for the true diehard who feels restricted by using someone else's system or the developer trying to eliminate all processing overhead caused by the operating system, alternatives do exist.

For straightforward, yet high-performance, process-control applications, the use of a finite-state machine as the controller is an easily implemented technique. A finite-state machine is simply some device that produces a defined output state based on its input state. For example, a microprocessor may read some input register, access a table in memory using this input as the address, and send out the value contained in the accessed location. In such a system, a value could be created with a single indirect move instruction in a microprocessor using a memory-mapped input/output scheme. Clearly, using a microprocessor this way would allow only a relatively small number of states.

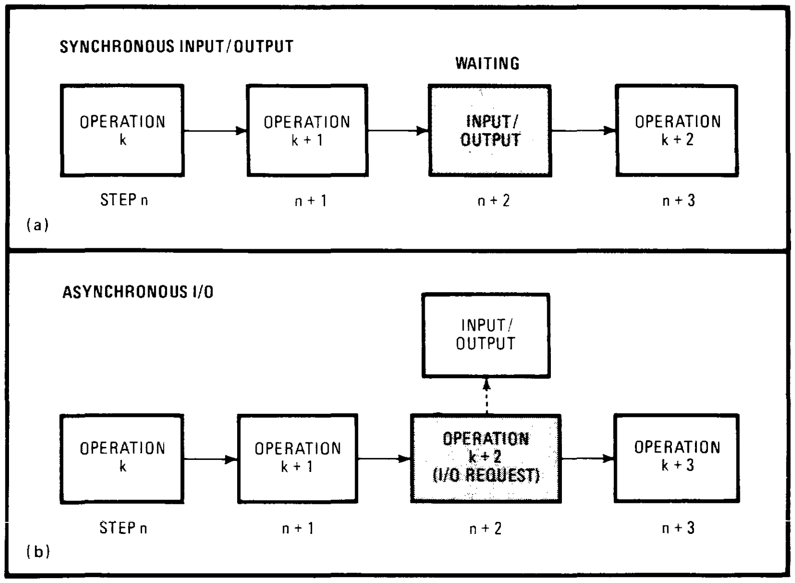
Besides this hardware approach, the software alternatives include the interpreters for high-level languages, such as Forth and concurrent versions of Pascal, that are appearing in the read-only memory of single-chip 8-bit microcomputers. For example, the CDP1804P complementary-MOS single-chip microcomputer from RCA Corp.'s Solid State division, Somerville, N. J. [*Electronics*, Nov. 30, 1982, p. 127], contains a core interpreter for Micro Concurrent Pascal (mCP) from Enertec Inc. of Lansdale, Pa. Based on Per Brinch Hansen's Concurrent Pascal, mCP contains all the constructs necessary for real-time applications, such as shared data, monitors, interrupt handling, and task queuing and switching. RCA also provides a ROM that extends the core interpreter to include full multitasking support. Software for this microsystem is developed using an RCA cross-compiler available on various host machines.

In parallel with the use of modified high-level languages like mCP, Forth interpreters are on the verge of appearing as single-chip microcomputers like the CDP1804 or the RF1/12 from Rockwell International Corp.'s Newport Beach, Calif., Electronic Devices division [*Electronics*, Jan. 13, 1983, p. 41]. After its development in the 1960s for real-time applications, Forth gained a slow acceptance among system developers. But with the inception of Forth standards committees and the spread of interpreters into more systems, this stack-oriented language is rapidly attracting the attention of larger houses.

Forth is a threaded language in which basic procedure calls, or words, are used to build up more complex words. Because of the threading, programs tend to be very compact. Once the programmer gets used to reverse-Polish notation, program development is simply a matter of building up the system dictionary with the words needed for the particular application.

ing systems, the AST mechanism allows tasks to perform operations that were contingent on the completion of a separate real-time operation. For example, by issuing an AST when it has completed its work, a read task is able to notify another task that a buffer has been filled. The other task is then free to initiate whatever calculation may be needed to make use of this new data.

Besides such modifications improving Unix's response to asynchronous events, Masscomp upgraded the system's throughput by adding support for contiguous files to the file-management system. In this way, large amounts of data may be written at a high speed to



2. No blocking. In synchronous I/O, execution of a task blocks, or waits (tinted), until the data transfer is completed (a). Since I/O is handled independently, a task need only request an I/O operation (shaded) and continue on to the next operation.

consecutive disk sectors. Since other disk accesses are locked out in this mode, the disk head will be positioned correctly, thereby eliminating unnecessary and time-consuming movements.

In addition to these I/O add-ons, Masscomp boosted intertask communication capability by enlarging the Unix standard intertask communication mechanism, called pipes, to allow tasks to transfer buffers. In an alternative approach, Charles River Data Systems of Natick, Mass., allows tasks in its Unix-like Unos system to share data directly. A number of independently constructed software tasks may use a common set of locations in memory to transfer data between themselves or to perform some sequence of calculations. However, whenever asynchronous tasks share some common resource, their use of the resource could result in corrupted data—unless some mechanism coordinates their activities, such as the monitor concept described on page 108. Unos provides a mechanism called event counts to help avoid these conditions.

Event counts are integer values that are a nondecreasing count of the number of times some particular event has occurred. By using an event count associated with some task that produces shared data and another event count for a task that consumes the shared data, programmers may ensure the correct sequencing of asynchronous data-producing and -consuming tasks. Similarly, event counts serve as primitive operations for emulating the synchronization function that is provided by semaphores and the mutual exclusion that is furnished by monitors.

Chips come to aid of embedded systems

□ Storing machine instructions in read-only memory is hardly a new concept in microprocessors. If supporting software totally breaks down, Digital Equipment Corp.'s LSI-11, for example, resorts to a basic keyboard monitor stored in a special ROM that is logically placed in the input/output address space. Using a primitive on-line debugging technique stored in the same ROM as the monitor, a software designer may read and alter memory locations and initiate a bootstrap loading operation from storage—a common provision in computer systems.

From these primitive beginnings, however, ROM-based software has evolved into complete operating systems in memory, engendering the term silicon software. Complementing hardware for distributed-processing architectures, such silicon-software systems signal a migration of application software into dedicated microcomputers previously considered unable to gain full systems capability. For developers of dedicated microcomputers embedded in some larger real-time system, silicon software spells the end of the need to reinvent the wheel to carry out the fundamental functions of a real-time operating system.

Extending the microprocessor

Functionally, silicon operating systems extend the microprocessor's instruction set to include system-level instructions that perform operations on software structures, like queues and tables, rather than on hardware registers. Application-program developers are then presented with a virtual machine—one that is perceived by the programmer as different from the actual host processor. In these virtual operating-system machines, their instruction set includes a well-defined set of system calls as well as the basic machine instructions of the host microprocessor. For example, with systems like VRTX and RMX, the virtual microprocessor has a special set of instructions for handling interrupts (see Table 1).

For system developers, however, the problems in developing reliable silicon software extend beyond resource protection, timing, and communication problems (see pp. 106–111). In fact, the development problems extend beyond the purely logistical exercise of maintaining a separate ROM-based instruction store and one for variables that need to be placed in system read-write memory. Treading a fine edge between the full function of a general operating system and the fine tuned performance of special-purpose software, silicon systems need to balance the need for a wide range of system functions with the requirement that they squeeze into a minimal amount of ROM.

Flexibility for expansion

Still, once a system meets a reasonable compromise between capability and size, it should not irrevocably lock the user into accepting its choices. For example, many real-time applications require some custom peripheral-device drivers and system-level functions. Consequently, the program should provide a mechanism for logically incorporating user-written extensions to the operating system, such as the user-defined pointers in the VRTX system from Hunter & Ready, Palo Alto, Calif.

In VRTX, a configuration table (Table 2) in system random-access memory allows specification of a custom routine that is to be executed whenever the system is initialized. For even more delicate control of system operations by custom software, a trio of pointers in the table specifies user-written routines to be accessed whenever a task is created or deleted or whenever a context switch is performed. Hunter & Ready also includes a location in this baseline configuration table for its anticipated file-management extensions to VRTX.

The 80130, an RMX-86 kernel in silicon from Intel Corp., Santa Clara, Calif., generalizes this approach through an index table containing pointers to system routines. If circumstances require the replacement of an existing system routine, the index-table pointer is merely altered to indicate the address of the new routine. In an

TABLE 1: SYSTEM CALLS FOR HANDLING INTERRUPTS

Instruction	Description
Versatile Real-Time Executive (VRTX)	
UI-POST	deposit message from interrupt handler
UI-EXIT	exit from interrupt handler
UI-TIMER	timer interrupt
UI-RXCHR	receiver ready interrupt
UI-TXRDY	transmitter ready interrupt
iRMX-86	
RQ$SET$INTERRUPT	assign interrupt handler
RQ$RESET$INTERRUPT	deassign interrupt handler
RQ$GET$LEVEL	return number of highest-priority interrupt level currently being processed
RQ$SIGNAL$INTERRUPT	signal from interrupt handler that event has occurred
RQ$WAIT$INTERRUPT	wait for occurrence of event
RQ$EXIT$INTERRUPT	relinquish control of the system
RQ$ENABLE	enable hardware to accept interrupts
RQ$DISABLE	disable hardware from accepting interrupts

TABLE 2: VRTX CONFIGURATION TABLE

Table Entry	Entry Description
sys-RAM-addr	system beginning address
sys-RAM-size	system memory size
sys-stack-size	system stack size
user-RAM-addr	starting address for available memory in initial partition
user-RAM-size	size of initial partition
user-block-size	size of memory block for dynamic allocation
user-stack-size	size of stack for user tasks
user-task-addr	address of first user task
user-task-count	maximum number of tasks
sys-init-addr	address of user-supplied initialization routine
sys-tcreate-addr	address of user-supplied routine accessed when a task is created
sys-tdelete-addr	address of user-supplied routine accessed when a task is deleted
sys-tswap-addr	address of user-supplied routine accessed when a context switch occurs
[RESERVED]	address of Hunter & Ready future extensions to VRTX

embedded system, this new routine could be placed in ROM along with application software.

Now that programs in ROM have matured into silicon systems, the development of software for embedded systems may now follow a more hospitable development cycle. The particular method used to create embedded systems will, in general, fall into one of two paths represented by the two major camps.

On one hand, kernels in silicon from systems such as RMX-86 or the MSP from Hemenway Corp., Boston, Mass., for the 68000 or Z8000 are self-contained subsets of the full operating system. Consequently, software programmers may use the full development version of the same operating system as that in the eventual target to create the application package. On the other hand, development of application programs around the ZRTS system from Zilog Corp., Cupertino, Calif., or Hunter & Ready's VRTX for the Z8002, iAPX-86 family, or 68000 relies on the use of a separate development system to create software for the target microprocessor, since this software does not have development versions.

Two approaches

The significance of these two approaches as usual depends on the intended application. Hunter & Ready views VRTX as a set of processor-independent building blocks that programmers use to construct application packages for embedded systems. As such, the programmers employ the same development systems that they might use to build application code, but now with the benefit of a sophisticated set of ready-made system-software components.

In playing its part in Intel's systematic drive toward providing an integrated environment around the iAPX-86 family, the 80130 holds the anchor position in an interlocked set of components. Able to function independently of the upper layers of the operating system, it provides a hardware base for the rest of RMX-86. Serving as a viewport into this system-software base for the central processing unit, Intel's universal run-time and development interfaces offer the mechanism for software portability needed for the next stage in the company's plan to grow into higher-performance microprocessors, such as the 186, 286, and 386.

While interlocking with the software in this way, the 80130 also must play its role in the complementary relationships being established at the hardware level. As such, it includes on-chip hardware support for system-level functions, including timers, interrupt controller, bus control, and bus interface.

Meanwhile, Intel's plan for software-in-silicon becomes evident as it gathers the other pieces of the puzzle, such as the 82730 text-coprocessor chip, the 82586 local-network coprocessor, and the 82720 graphics processor chip. Similar to the 80130 software connection, the 82720 graphics part interlocks with the rest of the system at the software level through its support of another well-defined software interface—the virtual device interface. Yet to come are pieces for voice I/O support, as well as some level of hardware support for data-base access. □

ARTICLE
REPRINT

AR-288

June 1983

Intel's Matchmaking Strategy: Marry iRMX™ Operating System with Hardware

Chappell Brown
Software Editor
Electronic Engineering Times

210341-004

Intel's Matchmaking Strategy: Marry iRMX™ Operating System With Hardware

Intel's major software product, the iRMX™-86 16-bit operating system, which is now in its fifth release, represented a three-year development investment which most independent software vendors would have found a daunting prospect in 1978 when the project was conceived.

The investment was essential. By the mid-1970s, feedback from OEMs working with Intel's hardware revealed problems with system integration—the marriage of software with hardware. It consequently slowed sales, with the prospect of even greater problems at higher levels of circuit integration. Intel management, looking for ways of coping with the ballooning software requirements of the rapidly accelerating hardware program, began stepping up software development programs in the mid-1970s.

"The RMX program illustrates a number of things one needs to keep in mind with developing a real-time operating system," explained Bill Lattin, Intel's OEM microcomputer systems manager. "Foundations must be well laid so the system can grow and evolve over time. And there is a need for the system to be open to modification by typical OEM-specific applications.

"Although the RMX program has been around since 1978, it has only recently hit its stride, as processor technology has advanced to use the full range of its features," Lattin said.

The fast-paced microcomputer market had created a new situation for systems designers in terms of a radical shift in the hardware/software cost ratio. Earlier hardware generations involved various expensive centralized facilities. Not only was software cheap in comparison, but the hardware environment changed slowly, so that it was also feasible to rewrite systems as needed.

But when the price of a computer drops to as low as $5, the hardware environment becomes volatile and software turns into a major investment. Intel was finding that customers might invest as much as two-thirds of their development costs in software, only to see it eclipsed by evolving VLSI technology.

It became evident that merely supplying components would become increasingly counterproductive. Thus, the Intel "total solution" emerged—a consistent systems approach to hardware sales, which naturally depends heavily on a viable software program.

Object-oriented programming is a method which has worked best in creating a software program blending with the component approach. By hiding data representation within an object with its own object manager, changes in the hardware environment that affect the data can be accommodated without having to change the rest of the software.

A price is paid in terms of program size with this approach, however. And it was difficult at the time to justify this kind of liability with the existing onboard memories of the 8-bit generation.

Bill Stevens, iRMX-86 program manager for release five, explained the difficult decisions that had to be made at the outset of the program. "Every engineering decision involves a trade-off. We wanted to optimize program productivity and we had to have modularity. The consequence of this was large size. It turned out that a minimum configuration was 12 kbytes wide and the full configuration was 128 kbytes. At the time we did not have 64k dynamic RAMs and 64k EPROMs, so we didn't have the technology to realize the systems of initial specifications times. Bruce Schafer has to take credit for making that decision to go ahead anyway, early on. . . it was a gutsy decision, and it turned out to be absolutely right."

Had Intel known of the difficulty it was about to encounter in producing its 64-kbyte RAM, Schafer may have had second thoughts.

Schafer joined Intel in 1976 and began working on iRMX-80. "It was a nice little system," Schafer said. "A miniature dispatcher had evolved to handle multiple asynchronous events and became a primitive OEM operating system. It was tempting to do an enlarged version of it, mainly because I was already working on it for the 16-bit generation."

Schafer soon found himself centrally involved in the task of heading off the 16-bit software crunch, laying groundwork for a system that could cover a wide range of applications, many of them unknown at the time, and a system which could also evolve with hardware advances.

"When you set out to design a system of that scope, you don't just sit down and start writing code. It's definitely a top-down process," explained Schafer. He discovered early in the project that the purely technical hurdles in writing software were minor compared to orchestrating a team of engineers on such a comprehensive project.

The iRMX-86 system is multi-layered, and the project had to be coordinated across these layers along with the sequence of planning, design and implementation. On top of that, a thorough testing program had to be coordinated with all phases.

"I had a difficult time convincing engineers on the project that documentation of their work was as important as the work itself. Specifications were absolutely crucial to the development phase," said Schafer.

Schafer began with a customer survey to discover the kind of problems OEMs were experiencing with system design. He wrote a production implementation plan, which was critiqued by marketing and engineering personnel. This was approved in June 1978 and formed the basis for engineering specifications. A critiquing process evolved as the organizing principle behind initial product design; engineers on the project would exchange documentation and then meet to evaluate the progress of the system.

The sessions were lively and the problems of coordinating implementation, testing and design along with the pressure of deadlines for the whole program generated quite a bit of excitement.

Development testing turned out to be a particularly thorny problem—the asynchronous interrupts and multiple-processing aspects of real-time applications required a special test apparatus to simulate a real-world environment.

What they came up with is a nucleus executing directly on the 8086 and 8088 processors as the basic building block of the system. Together with the next layer—a basic I/O system—a minimal operating system can be configured, which has been found useful in many applications.

However, it was necessary to develop an application on the Series-III development system even though the target was going to be RMX. "We quickly realized that users want to be able to do development work on the machine they target on," said Schafer. "This is particularly important for field maintenance . . . you can't drag a Series-III out to an oil derrick." To realize this goal, Intel built higher layers around iRMX so that program development could be done without a Series-III. Higher layers involve extended I/O and human interface facilities. After this, customer-written software can be added in high-level languages.

A major objective has been to provide a stable base for independent software vendors; with its latest release, Intel also announced an ISV program initially involving three major vendors; Microsoft, Digital Research and Mark Williams Inc.

The first release of iRMX-86 came out in April 1980. Since then, the system has been refined and released four more times, with release five appearing last December. An Interactive Configuration Utility appeared for the first time with release five, a further attempt to aid OEMs in putting their systems together. The system designer runs the ICU program on a terminal and is quizzed on his requirements, after which the program generates the unique iRMX software for his application.

"It has been a successful product in its own right, apart from its role in the hardware program, but I doubt that anyone would have wanted to invest in a three-year development process before there was a chance at some return," observed Stevens, who has been most excited by the diverse applications he has seen. "I've really enjoyed the iRMX symposiums. There is always some new system demonstrated. In Tokyo, I just saw an 8086-based scientific system with really first-class graphics put together by Seiko. Another time I saw a blood analyzer based on the system. There are even RMX-based personal computers."

ARTICLE REPRINT

AR-289

June 1983

iRMX™ 86 Has Functionality, Configurability

Bruce W. Schafer
Software News

iRMX™ 86 Has Functionality, Configurability

The iRMX™ 86 operating system provides a modular set of building blocks from which users can create a wide variety of applications. iRMX 86 features include: multitasking; interrupt support; multiprogramming support; device independence; tree-structured directories, file access control; and interactive debugging.

The iRMX 86 operating system combines the concepts of objects, types, and type extension to form a highly-functional and highly-configurable foundation for applications software. The operating system is designed for use with programs executing on the iAPX 86 and iAPX 88 processors. The 8087 numeric data processor is supported as on option.

Execution Environment

The iRMX 86 Operating System can be used with a variety of hardware configurations. Interactive disk-based systems as well as ROM-resident systems can be constructed.

Any part of the operating system's code can reside in ROM/PROM memory. Alternatively, all or part of this code can be "bootstrapped" into RAM using a small, configurable bootstrap loader provided with the product. The application code can similarly be committed to PROM or bootstrapped into RAM.

The operating system divides the execution environment into **jobs** and **tasks**. A task is described by a set of processor registers, a stack, a priority, and a state. Jobs provide resources for tasks. A job can be viewed as a task environment. In the simplest case a job represents a memory pool. Tasks executing in the same job share the same pool of memory. When a job is deleted, all tasks within the job are also deleted and all memory allocated to these tasks is deallocated.

The iRMX 86 system is composed of several layers. The innermost layer is the Nucleus, which provides multitasking, interrupt control, and multiprogramming support. The first optional layer, the Basic I/O System, supports device-independence, directories, random access, and file access control.

"On top" of this layer, users may add the Extended I/O System (providing services such as automatic buffering) or the Application Loader (which supports loading both absolute code and locatable code). The Human Interface uses these inner layers to support user-defined commands in addition to a set of standard commands.

The design of the iRMX 86 Operating System is based on a set of *object types*. The operating system supports dynamic object creation. Each time an object is created, the operating system allocates the proper resources to the object and returns a 16-bit virtual address called the object's *token*. This token is subsequently used by the application to identify the specific object.

By implementing this object-oriented approach, the iRMX 86 Operating System hides implementation details from the application software. The iRMX 86 Nucleus also allows users to add custom object types without changing the Nucleus.

I/O Devices

I/O devices can be manipulated in two ways. The first approach allows the application to receive interrupts directly from the I/O device. The second approach utilizes the iRMX 86 Basic I/O System. With this approach, a device driver must be written for initiation of I/O requests and for interrupt handling. The application software interfaces to these drivers through the Basic I/O System by making **read, write, seek,** and **special-function** requests.

The iRMX 86 Operating System currently includes device drivers for diskettes, Winchester disks, magnetic bubble storage devices, and Storage Module Device (SMD) interfaces.

The iRMX 86 Extended I/O System defines the concept of a *logical device*. Using this feature, each device is assigned a logical name. Application programs refer to logical devices without knowing which **physical device** is associated with each logical device. In this manner, the physical device can be changed without changing the application programs.

The Basic I/O System provides asynchronous I/O functions. Each asynchronous function is initiated by a procedure call that queues the request. The procedure call returns immediately with an indication of whether the request was successfully queued. When the request is actually completed, a response message is sent to the mailbox specified.

The Extended I/O System automatically synchronizes I/O requests. Again, a procedure call is used to initiate I/O. The procedure, however, does not return until the request is complete. To enhance efficiency when this automatic synchronization is used, the Extended I/O System permits read-ahead and write-behind.

The iRMX 86 Human Interface automatically parses input lines and invokes the appropriate program based on the first word in each line. A program executing under the iRMX 86 Human Interface can request command execution by providing the text for these commands to the command line interpreter.

The iRMX 86 Human Interface is supplied with a basic set of commands to manipulate files. These commands include **directory** display, **create** directory, **rename** file, **copy** file, **delete** file, and **submit** a set of commands. Users can add custom commands to this set.

The iRMX 86 Debugger provides the capability to debug one or more tasks while the rest of the system continues to execute. The Debugger allows a user to specify that a task be suspended when the task executes a particular instruction and when the task communicates with other tasks.

The most general communication mechanism provided by the iRMX 86 Nucleus is the **mailbox** object type. Each object of this type is described by two queues—a queue of messages waiting to be handled by tasks and a queue of tasks waiting for messages. An additional attribute of a mailbox is the specification of whether the queue of tasks is to be handled first-in, first-out or on a relative priority basis.

The iRMX 86 Operating System also provides a **semaphore** object type. Each semaphore is described by a queue of waiting tasks and a unit count. This unit count is equivalent to a count of empty messages at a mailbox, but, because no actual messages are involved, a semaphore is a more efficient mechanism than a mailbox. Since semaphores allow multiple units to be sent at the same time, semaphores are used to create deadlock allocation functions.

To provide additional efficiency, the iRMX 86 Nucleus also provides a special type of semaphore called a **region.**

Each iRMX 86 task has a **dynamic priority** attribute. This priority describes the relative importance of the Task's function with respect to other system functions. The iRMX 86 Nucleus always runs the highest priority ready task. When several tasks of the same priority are ready, the Nucleus arbitrarily chooses between them.

Scheduling requires changing the state of the task and placing the task in a queue of ready tasks. Whenever a task is scheduled or descheduled, the Nucleus checks the ready queue and allocates the processor to the highest priority ready task. In order to ensure event-driven scheduling, the iRMX 86 Nucleus is designed to place an absolute limit on the interval during which interrupts are masked.

One attribute of the **job** type is a **memory pool.** Each memory pool represents the memory resources available to the tasks executing within a job. All objects created by these tasks are allocated memory from the pool.

The iRMX 86 Operating System supports three **file types.** In all cases, application programs read and write data without knowing the device or the file type that is used. The following file types are supported:

1) **Physical**—A device accessed as a physical file is treated as a contiguous sequence of bytes.

2) **Stream**—Stream files do not exist on actual physical devices; rather, data is transmitted directly from one program to another.

3) **Named**—Named files represent the traditional notion of files. Named files are described by a path through a tree-structured network of directories.

The name of an iRMX 86 file is given as a path through a tree-structured network of directories. Each directory in this structure can point directly to data files and to other directories. One directory on each device is considered the **root** directory. All paths on a particular device begin in this directory.

The basic file functions for all three file types are: **open, close, read,** and **write.** When random file access is required, the **seek** system call is added to this set. For named files, additional functions are needed. These functions include the **rename** function, the **truncate** function, and the **change-access** function.

When a file is opened, the calling program specifies the type of file access required. For a data file, three types of access are permitted: **read, write,** and the **read/write** combination. The I/O system verifies that the specified access is available and grants the **open** request only if the requested access is available.

NOTES:

NOTES:

INTEL CORPORATION
3065 Bowers Avenue
Santa Clara, CA 95051

INTEL INTERNATIONAL
Rue du Moulin a Papier 51, Boite 1,
B-1160 Brussels, Belgium

INTEL JAPAN K.K.
5-6 Tokodui, Toyosuto-machi
Tsukuba-gun, Ibaraki-ken 300-26
Japan

Printed in U.S.A./OMO/0683/10K/BA/WCP/ER
Order Number: 210341-004