intel®

ARTICLE
REPRINT

AR-41

July 1977

An Integral Real-Time Executive For Microcomputers

COMPUTER DESIGN July 1977

9800579A

Single-board microcomputers offer hardware cost-effectiveness for implementing many real-time systems. A compatible, resident, real-time executive program provides savings in software development

An Integral Real-Time Executive For Microcomputers

Kenneth Burgett and Edward F. O'Neil

Intel Corporation
Santa Clara, California

Single-board computers, or microcomputers, that contain central processor, read-write and programmable read-only memory, real-time clock, interrupts, and serial and parallel input/output all on one printed circuit board, have made feasible a whole spectrum of applications which previously could not be economically justified. These microcomputers have also opened up a range of applications where the high functional density of large-scale integration provides advantages over previous solutions such as hardwired logic or relatively expensive minicomputers. While microcomputers readily solve hardware requirements, software for single-board computer applications with real-time characteristics (which are in the majority) has until now been generated individually for each application.

The Intel RMX/80* Real-Time Multi-Tasking Executive simplifies real-time application software development, and at the same time furnishes capabilities optimized for the microcomputer environment. It provides the means to concurrently monitor and control multiple external events that occur asynchronously in real-time. The program framework allows system builders to immediately implement software for their particular applications, and to avoid specific details of system interaction.

Major functions of the executive include system resource access based on task priority, intertask communication, interrupt driven device control, real-time clock control, and interrupt handling. In combination, these functions eliminate the need to implement detailed real-time coordination for specific applications.

Previously, two alternative software approaches were used to solve microcomputer applications. First, many designers created their own operating executive, individually tailored for each application. Obviously, this approach was expensive and time-consuming. The second approach was to use a minicomputer executive which had been adapted to a microcomputer. Since this software was designed for a different processing environment and then "stripped down," it suffered from major inadequacies when executed on microcomputers. The alternative, RMX/80, has been designed specifically to provide a general-purpose real-time executive tailored to Intel SBC 80 and System 80 microcomputers.

Real-Time System Requirements

All software design approaches for use in real-time applications include capability for concurrence, priority, and synchronization/communication.

Concurrence—Real-time systems monitor and control events which are occurring asynchronously in the physical world. Microcomputer software does not know exactly when external events will occur; however, it must be prepared to perform the necessary processing upon demand, whenever the events actually do occur. Typically, interrupts are used to inform the microcomputer that an event has occurred. At interrupt time, system control software determines what processing to perform, as well as the relative sequence in which processing must take place.

*RMX/80™ is a registered trademark of the Intel Corp, Santa Clara, Calif.

Programs related to external events are processed in an interleaved manner based on interrupt occurrence and priority. For instance, one routine is executing when an interrupt activates, signaling that a higher priority event has occurred. At this point, the routine related to the priority interrupt is started, while execution of the less important routine is discontinued temporarily. When the more important routine is completed, or temporarily halted for some other reason, execution of the less important routine is resumed. In this manner, multiple programs execute concurrently in an interleaved fashion.

Priority—In a real-time environment, certain events require more immediate attention than others because of their significance within the physical world. Immediacy is relative to other processing, and is determined by application requirements. The concept of immediacy or priority, however, is common throughout all real-time microcomputer applications. In priority-based systems, the most important program (one that is not waiting for some physical or logical reason) is the one executing.

A classic illustration of program priority in real-time systems is found in the area of plant control. When the plant begins to fail in a nonrecoverable manner, it is imperative that the plant be shut down as quickly as possible. For this reason, shutdown processing takes priority over all other system demands. Software priority enforces this hardware concept of physical operational events.

Synchronization/Communication—Another common similarity in most real-time systems is the need for synchronization between various events in the physical world which are under microcomputer control. Synchronization is defined as the process whereby one event may cause one or more other events to occur. Communication is the process through which data are sent between input/output (I/O) devices or programs and other programs within the microcomputer system.

An example of the need for synchronization and communication is a microcomputer system for weighing and stamping packages. One part of the system weighs the package, calculates pricing, and releases the package onto a conveyor belt. Price and weight data are communicated to another part of the system which stamps the data onto the package after it arrives at a sensor station. Synchronization is demonstrated by the occurrence of one event—package arrival—causing another event—package stamping—to occur.

Compatible Benefits

To satisfy real-time microcomputer software requirements, the RMX/80 Real-Time Executive software (Fig 1) was designed. This program differs from existing software systems by offering capabilities directly related to the single-board microcomputer environment in which it operates. These capabilities have two major bottom-line benefits compared with equivalent minicomputer systems. First, the executive code is compact enough to allow a large number of real-time applications to be processed on a single microcomputer board. To accomplish this capability, its nucleus is optimized to reside in less than 2k bytes [ie, in a single 16k programmable read-only memory (p/ROM)], thereby allowing up to 10K of onboard memory for application-related software and storage.

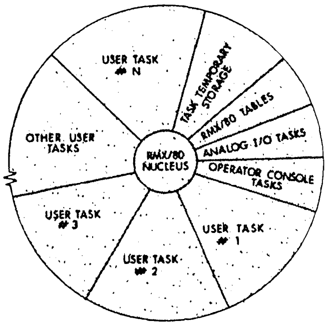

Fig 1 A typical RMX/80 system. Multiple tasks control a given application. Nucleus controls execution of both user and executive tasks through task-to-task communication, real-time clock, priority resolution, and interrupt handling facilities. All tasks within an RMX/80-based application use at least some of these capabilities; other optional executive tasks include debugger, free-space manager, and device control for operator's console, diskette file system, analog subsystems, and high speed mathematics unit

Second, the executive may be p/ROM-resident. When the microcomputer system is powered on, the software system (executive plus application programs) is automatically initialized and begins execution of the highest priority application task. Typical major real-time executives, however, are totally random-access read-write semiconductor memory (RAM)-resident, which means they must be initialized (booted) from a peripheral device, such as diskette, cassette, or communications line, into microcomputer memory. The need for peripheral devices significantly increases the total cost of traditional real-time executive-based solutions.

Sample Application

Functioning as a real-time executive for microcomputers, this software system provides facilities for orderly control and monitoring of asynchronously occurring external events. Although these events may differ widely from application to application, facilities are adaptable to nearly all processes where the microcomputers are used, including process and machine control, test and measurement, data communications, and specialized online data processing applications (where one or more terminals access diskette-based data). The executive is particularly useful in dedicated low cost applications which were not economically feasible before the advent of microcomputers. For example, consider the requirement of gas pump control in a service station (Fig 2).

In this station, a microcomputer system operating with RMX/80 concurrently monitors and controls multiple gas pumps, and sends price and volume information to one central location. At the same time, information about station operation is being transmitted over a communications line to a regional computer.

Individual tasks are developed independently to measure gas flow, calculate and display price information, transfer data to the central computer, and monitor levels of gasoline in underground storage. All this processing takes place concurrently under program control. (Credit verification, charge slip printing, and billing can also be controlled by additional software tasks.)

Efficient gas station operation demands that the hardware/software system be highly reliable. The compatible benefits of compact code, p/ROM residency, and self-initialization on a single-board microcomputer system all combine to ensure functional integrity.

Software Structure

RMX/80 simplifies the effort for developing a real-time system, first, by providing many commonly required software functions. Second, its software structure promotes efficient program development. Programmers who are familiar with structured programming will find task orientation both natural and easy to use.

Tasking means that a larger program is divided into a number of smaller, logically independent programs or tasks. The key is to identify functions that may occur concurrently. For example, consider the tasks required for a terminal handler—real-time asynchronous I/O between an operator's CRT terminal and the executive.

Input Handler Task—One task must be ready to accept a data character from the terminal at any time. This is done by responding to an interrupt signal from the terminal and then accepting the data character. The task immediately passes the input character to a subsequent task automatically and then goes back to wait for another interrupt.

Line Buffer Task—As characters are received from the input handler they must be placed into a buffer to form a line. Eventually, the buffer will be filled or the logical end-of-line will be signaled by a carriage return character. At this point, the line buffer must be sent to some other task for processing.

Echo Driver Task—For a full-duplex terminal, it is necessary to return each input character to the terminal for display on the CRT screen. This task waits for a character, which could be sent by either the line buffer or input handler task, and then sends the character to the terminal. It then waits for the next character.

Note that input handler and echo driver are described as waiting for an event. Within the RMX/80, that is literally the case. While they wait, however, system resources are available for other tasks, such as that of the line buffer. Thus, effective processing may occur concurrently with necessary waiting periods. Notice also that a number of other tasks may also be active within the system. In fact, the greater the number of tasks running concurrently, the more effectively system resources are used. Concurrent operation eliminates many time wasting procedures from a real-time system. For example, the executive can eliminate the need for many timing loops where the processor simply executes a no-operation instruction repeatedly while waiting for an event to occur.

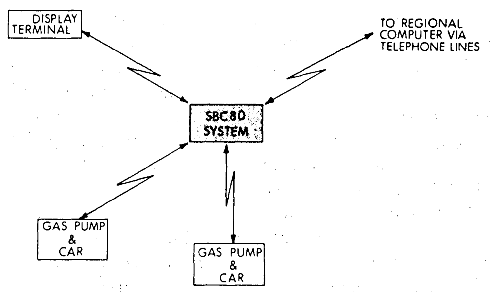

Fig 2 Microcomputer control for gas pump automation. In this example, executive-based system simultaneously controls two pumps, displays information on operator's console, and communicates with regional computer. At a given time, more or fewer functions could be operating concurrently. System expansion can be easily accomplished by adding tasks and modular hardware

Within the executive, tasks not only are logically independent, they are also physically independent, actually contending with each other for the use of the processor and other system resources. The executive resolves this contention based on the priority of each task.

In the terminal handler example, it is clear that the input handler must have highest priority, since acceptable performance cannot tolerate the loss of data. Second highest priority is given to the echo driver, so that data appearing on the screen remain coordinated with the input. Lowest priority goes to the line buffer, since that function does not depend directly on an external asynchronous event. There are no particular real-time constraints on the line buffer as long as the input characters are eventually processed.

It is possible to write the entire terminal handler as a single large task instead of as several smaller tasks. However, consideration must be given other high priority tasks operating within the system which may not be able to gain control while a low priority portion of the terminal handler, such as the line buffer task, is executing. Therefore, tasks assigned as high priority are generally kept as short as possible. If the terminal handler were written as one large task, it could tie up the entire processing system for a relatively trivial function.

Task States

Two task states have been implied—running and waiting. A running task is always the task which currently has the highest priority and is not suspended or waiting. A waiting task remains in the wait state until it receives a message or an interrupt for which it is waiting or until a specified time period has passed. The wait period can be timed using the system clock.

A running task may suspend itself on some other task in the system. A suspended task cannot begin execution again until some running task orders it to resume. As an example, a password routine might temporarily suspend the echo driver of the terminal handler so that the password is not displayed. (The password routine must

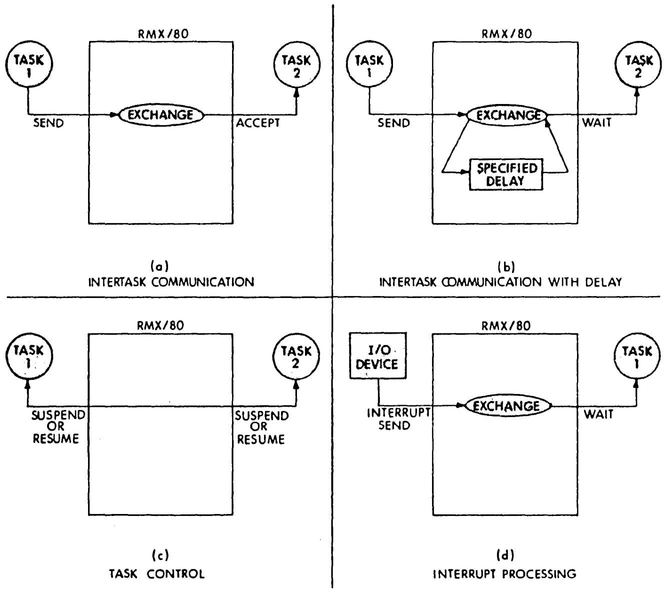

Fig 3 System message exchanges. In intertask communication (a) task 1 sends a message to an exchange, where it is held until task 2 requests message via accept. In intertask communication with delay (b), task 2 waits for a message from task 1 until data are available or until a certain time period has passed, whichever occurs first. In task control (c), any task may suspend or resume any other task. In interrupt processing (d), an I/O interrupt is transformed into a message that task 1 receives via a wait command. Task 1 then performs appropriate interrupt processing

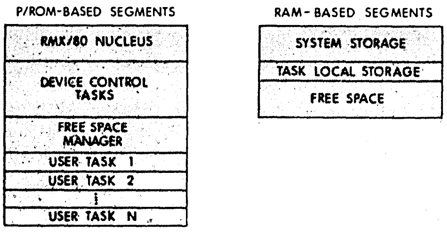

Fig 4 Memory utilization. RMX/80 nucleus, device control task, and free-space allocation modules are linked with user tasks to form a real-time system. Although executive may be RAM-resident, it is designed to reside in p/ROM and uses RAM only for temporary storage and free space. User tasks are provided by user at generation time. RAM may be used by RMX/80 and all associated tasks for temporary storage, including stack.

remove the password from the line buffer, or it will be displayed as soon as execution of the echo driver is resumed.)

A task may also be in the ready state. A ready task is one that would be running except that a task with higher priority temporarily controls the system resources. The executive maintains a list of all tasks that are ready to run. The next task to be run is always the task with the highest priority in the ready list.

The running task relinquishes its control of the system by

(1) Putting itself into a wait state

(2) Suspending itself

(3) Sending a message to a higher priority task, which if it has the highest current priority, becomes the running task

(4) Being preempted by an interrupt to a higher priority task

In the case of an interrupt, the executive saves the status (contents of registers, etc) of the interrupted task so that it will be restarted correctly.

Message Exchanges

Tasks communicate with each other by sending messages (Fig 3). The sending task constructs the message to be sent in RAM or uses a previously assembled message.

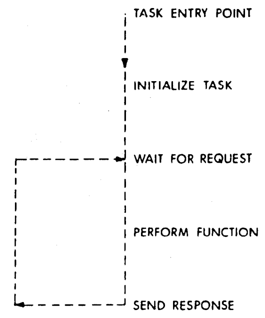

Fig 5 Consumer task flow. Consumer task performs initialization and then drops into cyclic loop, alternately waiting for messages, performing functions requested by message, and sending an acknowledgement in form of a response message

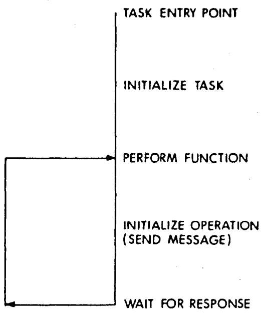

Fig 6 Producer task flow. Producer processing flow is opposite to that of consumer task. Instead of passively reacting to requests from other tasks, producer task issues requests to which other tasks must respond

The sending task then issues a SEND command that posts the address of the message at an exchange.

An exchange is simply a set of lists maintained by the executive. The first list contains the addresses of messages available at that exchange. The second list consists of a list of tasks that are waiting for messages at that exchange. When a task enters a wait state, it specifies the exchange where it expects eventually to find a message. The task may wait indefinitely, or it may specify that it will only wait a specific period of time before resuming execution.

Messages, together with the exchange mechanism, provide for automatic intertask communication and also for task synchronization. For example, a message to a particular task may specify that the task is to send a response to a certain exchange. Thus, the original task may request an acknowledgement response to its message, or it may specify that a message is to be sent to a third task. RMX/80 treats interrupts like messages, the only difference being that interrupts have their own set of exchanges.

Note that the sending and receiving of messages classifies tasks into two types—message consumers and message producers. A consumer task waits for a message, performs an action based on the message, and then returns to the wait state until another message is received. A producer task initiates its function by sending a message to another task, waits for a response, and then sends another message. Figs 5 and 6 graphically illustrate the processing within these two tasks. The distinction between consumer and producer tasks is relative since many tasks act as both consumer and producer.

Executive Modules

RMX/80 is supplied as a library of relocatable and linkable modules. These modules are added selectively as required when the user-supplied tasks are passed through the link program. Only modules actually requested by the application are linked in. For example, if the application program does not specify use of the free-space manager, that module is not linked into the system.

One module, the nucleus, provides basic capabilities (concurrence, priority, and synchronization/communication) found in all real-time systems. Additional, optional modules may be configured with user programs (tasks) to form a complete application software system. These modules include:

Terminal handler—Providing real-time asynchronous I/O between an operator's terminal and tasks running under the RMX/80 executive, the handler offers a line-edit feature similar to that of ISIS-II and an additional type-ahead facility. (ISIS-II is the supervisory system used on the Intellec Development System.)

Free-space manager—This module maintains a pool of free RAM and allocates memory out of the pool upon request from a task. In addition, the manager reclaims memory and returns it to the pool when it is no longer needed.

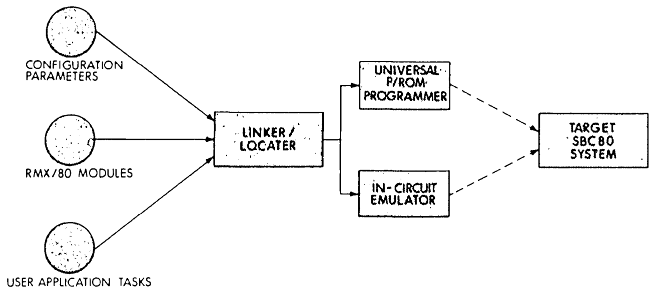

Fig 7 Target microcomputer system. Configuration parameters are linked together with appropriate RMX/80 and user task modules. Resulting program is then transferred to its target SBC 80 system via programmed p/ROMs or is debugged using in-circuit emulation and then transferred

Debugger—Designed specifically for debugging software running under the RMX/80 executive, the debugger is used by linking it to an application program or task. Thus, it can be run directly from the single-board computer's memory. In addition, an in-circuit emulator, such as ICE-80, can be used to load and execute the debugger, providing all resources of the Intellec development system to simplify debugging effort.

Analog interface handlers—Consisting of RMX/80 tasks, these handlers provide real-time control for SBC 711, 724, and 732 systems.

Diskette file systems—Giving RMX/80 users diskette file management capabilities, the diskette driver allows users to load tasks into the system and to create, access, and delete files in a real-time environment without disrupting normal processing. All file formats are compatible with ISIS-II for both single and double density systems.

In addition to application program module or task requirements, the user also supplies a set of generation parameters. These parameters are a set of tables that inform the executive of the number of tasks and exchanges in the system. Fig 7 illustrates the system generation process.

Summary

The significance of RMX/80 to software design parallels the significance of the single-board computer to hardware design. Microcomputers allow designers without extensive experience in digital systems to bring computer processing power into their applications. Similarly, the executive relieves the hardware designer of much software design required for real-time applications. Designed to facilitate growth, since new software needed to support hardware expansions can be supported easily by the addition of new tasks, this executive also substantially reduces recurring costs because it requires a minimum of memory and does not require peripheral bootstrap loading devices. RMX/80 results in economical, shorter, and more flexible software development efforts when designing, building, and verifying real-time user applications.

Bibliography

C. G. Bell, A. Newell, *Computer Structures: Readings and Examples*, McGraw-Hill, New York, 1971

P. Brinch-Hansen, *Operating Systems Principles*, Prentice Hall, 1973

E. W. Dijkstra, "The Structure of the *THE* Multiprogramming Systems," *Communications of the ACM*, May 1968, pp 341-346

E. I. Organick, *The Multics System: An Examination of Its Structure*, MIT Press, Cambridge, Mass, 1972

D. M. Richie, K. Thompson, "The UNIX Time Sharing System," *Communications of the ACM*, July 1974, pp 135-143

A Small-Scale Operating System Foundation for Microprocesor Applications

KEVIN C. KAHN

Abstract—Sound engineering methodology, which has long been valued in hardware design, has been slower to develop in software design. This paper uses a case study of a small real-time system to discuss software design philosophies, with particuar emphasis on the abstract machine view of systems. It demonstrates how the currently popular software design axioms of generality and modularity can be used to produce a software system that meets severe space constraints while remaining relatively portable across a family of microcomputers. These sorts of constraints have often been used to justify *ad hoc* design approaches in the past. The results of the project suggest that the use of such techniques actually make the meeting of many constraints easier than would a less organized approach. In addition, the reliability and maintainability of the resultant product is likely to be better.

I. Introduction

A PROCESSOR, as defined only by its hardware, is typically not an adequate base upon which to build applications software. Broad classes of applications can be examined and found to share more than the hardware defined instruction set. To avoid the reengineering of this common functionality, we would prefer to build such common parts once and thereafter treat this base software as though it were part of the machine. For example, a software system sometimes called an operating system, an executive, a nucleus, a kernel, or some similar term, is often supplied with a hardware product and can be viewed in exactly this way. In this paper, we examine a small-scale system to demonstrate this approach to bridging the gap between the hardware and the application. That is, we will view the software as a direct extension of the hardware—a view which may indicate future directions in microprocessor integration of function.

This paper is meant as both a case study of a particular system design and as a suggestion of the proper approach to such design situations in general. We will first discuss the abstract machine view of computer systems and attempt to demonstrate that this is a useful philosophical approach for building systems. We will then apply this approach to the discussion of a system to coordinate programs performing real-time control functions— RMX-80^{TM} [18]. The emphasis of the paper will be on techniques and methodology rather than on the particular functionality of RMX. Special attention will be given to such issues as the use of modularity to enhance the adaptability of the system and the use of design generality to achieve global rather than local optimizations.

II. The Concept of Abstract Machine

What is a computing "machine" or processing unit? We generally identify a processing unit as a particular collection of hardware components that implement the instruction set of the machine. This very physical definition of a computer dates from mechanical processors. Even with modern computers, before large-scale integration, it was easy to physically point at the processing elements as distinct from memories, peripherals, and programs. Continued integration of function has at least made this physical distinction more difficult with single chips subsuming processing, memory, and peripheral interface functions. Microprogramming (i.e., replacing hardwired instruction logic with a more elementary programmed processor) as an implementation strategy has logically blurred this distinction as well. That is, when the basic visible instruction set of a processor is itself implemented in terms of more primitive instructions it is more difficult to identify "the machine." It is clear that this narrow physical definition of a processor is not adequate for current technology levels and is likely to become even less viable as the technology continues to develop.

Actually we have been using alternative definitions of a processor for some time. All of the theoretical work in finite state machines, for example, deals with conceptual processors. Likewise applications programmers seldom really regard the machine they program as much more than collection of instructions found in a reference manual—the physical implementation of the machine is of little concern to them. Indeed, they may never come physically near the hardware if they deal with a typical time-sharing system—rather, the terminal is the only physical manifestation of the computer such users may see.

More to point, perhaps, are the numerous interpreters that have been written for languages such as Basic. Each such interpreter actually produces a conceptual machine with one instruction set targetted to a specific application. With standard compiled languages such as Fortran, Algol, or Pascal, a higher level source statement is translated into the instruction set of the physical hardware. In contrast, interpreted language systems translate the source into the instruction set of some conceptual machine that is better suited to the running of programs written in the language. For example, the hardware may not provide floating-point instructions or define a floating-point data representation. In such a case it may be easier to define a machine that recognizes a particular floating-point data format with an instruction set that includes floating operations. These interpreters are high-level machines that have usually been implemented in software. Likewise, it should be readily apparent that, just as these interpreters provide high-level machines to their associated translators, any programming language, compiled or interpreted, provides one to its users.

Interpreters of this sort typically may examine and decode a stream of instruction values in a manner analogous to the hardware. Alternately, the new instructions may all be executed as subroutine calls using the appropriate hardware instruction. That is, the entire bit pattern for CALL X (where X is the address of a

Manuscript received September 1, 1977; revised October 11, 1977.
The author is with the Intel Corporation, Aloha, OR 97005.
TM Intel Corporation, Santa Clara, CA.

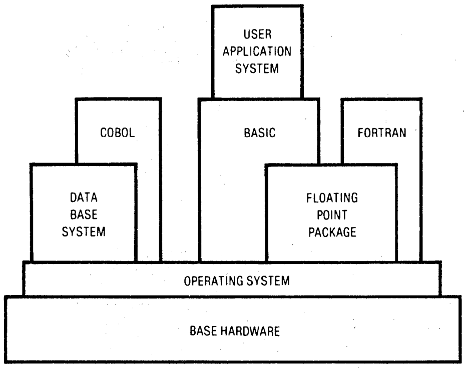

Fig. 1. Typical collection of abstract machines.

routine that implements a part of the new instruction set) can be regarded as a new operation code rather than as the hardware operation CALL. In either case the programmer using these extensions can view the harware–software combination as though it were a new machine with a more useful instruction set. Microprogrammed machines such as the IBM 5100 or Burrough's 1700 have simply optimized the performance of such interpreters or subroutine packages by committing them to a faster storage medium.

Viewed in this light we can identify any collection of hardware and software that provide some well defined set of functions as defining an *abstract machine* [10],[12]. This machine has an instruction set that consists of the functions provided by the hardware–software combination. For a particular application it may be possible to view multiple such abstract machines by taking various pieces of the whole. For example, the physical machine provided by a set of components is just one abstract machine. It is of particular interest since it is the greatest common abstract machine that can be identified as being used by any application running on that computer system. A Basic interpreter running on this machine might then constitute a second virtual machine. A Basic program running on this interpreter that accepted high level commands and performed according to them might be a third level machine usable by people with no knowledge of either the hardware or Basic. Whenever we can identify functions of sufficient commonality among a number of applications, it may be worth viewing the software which provides these functions as extensions of the base hardware machine which define some augmented or even different machines. Users programming such an application can then view this abstract machine, rather than the base machine as the vehicle that they are programming, and in doing so avoid reengineering the functions that it provides. Fig. 1 illustrates an example of such machines. It is important to remember that at any time, many abstract machines may be thought of as existing on the same base hardware.

III. Operating Systems as Abstract Machines

The terms operating system or executive have been used to describe software systems of widely different functionality. These machines generally provide for the management of some machine resources such as input, output, memory space, memory access, or processor execution time. We might then attempt to define an operating system as some collection of software modules which defines an abstract machine that includes resource management functions as well as the hardware supplied computational functions [2],[6],[8],[11]. With such a broad definition, however, large-scale multi-user time-sharing systems and small single user microprocessor development systems both may claim to have operating systems. Clearly, the range of software systems covered by this definition is large, encompassing products which differ by orders of magnitude in complexity. Rather than become involved in trying to resolve this disparity, we will qualify our use of the term and refer to an operating system "foundation." That is, we will describe a software system which provides a minimal base for the construction of real-time applications. We will avoid the somewhat irrelevant question of whether the system comprises a complete "operating system."

The important item to realize from the above discussion is that any operating system functionally enlarges the processor seen by the programmer. The functions that it provides become as much a part of the machine's functionality as jump instructions. Indeed, it is functionally unimportant to the user desiring to read from a file whether it requires a single hardware instruction or a large software routine to accomplish it. In terms of the abstract machine discussion above, we will examine a software package which defines an abstract machine that includes functions required to coordinate programs performing real-time control applications [1],[9],[12].

The key overall requirement of the operating sytem foundation that we discuss in this paper will be that it supply a minimal covering set of functions to permit coordination of asynchronous tasks. To determine this set we will need to further examine the needs of its users and environment of its use. In describing this foundation, we are defining an abstract machine that must be programmed to be of use; that is, like the instruction set of the base machine the foundation by itself performs no work but rather provides an environment within which useful tasks can be run.

We should note, here, some of the limitations of the system which differentiate it from large-scale operating systems. First, it is not primarily intended for a multi-user environment, particularly because the underlying hardware does not provide the necessary support to protect users from one another. Also, it will often be used to control functions of specialized devices and therefore is "close" to the I/O devices. That is, it does not supply the sort of high level I/O control system which is often present in larger systems for controlling more conventional I/O devices. Finally, it does not assume a backing store from which program overlays can be loaded (but it can easily support such an extension).

IV. Design Considerations

A. Use Environment

The foundation system we will describe is RMX-80 [5] which was designed to be used with members of Intel's Single Board Computer (SBC) family of products. This family includes a wide range of bus compatible processor, memory, and peripheral boards. Of most interest to this discussion are the processor boards which are based on the Intel 8080 or 8085 microprocessors and include varying amounts of on-board ROM and RAM memory and I/O interfaces. In addition, the boards vary in the sophistication of their interrupt structures and timing facilities. In terms of abstract machines we might characterize these computers as essentially the same machine at the processor level but different machines at the computer system level. It was desired that the abstract machines defined by adding RMX to the underlying computers be as much the same as possible.

During the design of RMX, we expected that its users would span the entire broad range of applications across which the SBC hardware was being put to use. This implied that it might see uses ranging from minimal single board systems that functioned as single device controllers to complex multiboard applications implementing involved real-time process or industrial control functions. In particular we expected that many user-built I/O boards and peripherals would be used with the system. It was important for us to allow full use of these unknown devices with RMX while still providing as much assistance as possible in the building of the controlling software systems.

As is the case with most processors, the concrete (i.e., physical) machines represented by the SBC family do not themselves include any facilities to permit multiple asynchronous functions to be programmed, to provide for the coordination of such functions, or to provide time information needed for real-time applications. Typically, users of these products have directly programmed these functions in an *ad hoc* manner within their applications. An examination of the sorts of functions necessary to such applications reveals that at the very least this reengineering is a waste of resources. Worse is the high probability of error in programming such critical functions.

The SBC hardware products were designed to eliminate the complexities of board engineering, particularly for those users without the necessary expertise to handle the task, by functionally integrating individual components into complete boards. The programming of functions to coordinate parallel software activities is, likewise, an area which should be carefully engineered in order to avoid subtle errors. The development of RMX was therefore viewed as a process of functional integration analogous to the integration of LSI components into boards. That is, just as a well designed board relieves the user of component level hardware engineering, RMX relieves the users of low-level software engineering.

B. System Requirements

The hardware environments and anticipated uses of RMX defined a stringent set of requirements for it. Foremost among these were its memory constraints; indeed, for the anticipated uses, memory size considerations dominated execution speed ones over a considerable range. Since we expected applications that would reside entirely on a single board with 4K bytes of PROM, the maximum size of the RMX foundation code was set at half of this or 2K bytes. Further, unlike larger minicomputer systems, many, if not most, applications of the SBC boards would not have available any mass storage or other program loading device. It was thus important that RMX be designed to be ROM (or PROM) resident and capable of automatically initializing the system when powered on.

We also anticipated that the expertise of many RMX users would be in areas other than programming systems. We therefore felt that the RMX machine needed to provide a fairly simple set of concepts, avoiding where possible those constructs most likely to cause errors. For example, we felt that a very frequent source of programming difficulty lay in dealing with interrupts. Many latent errors in programming systems stem from the occurrence of an interrupt at an unexpected time. We therefore decided to attempt to minimize the need for users to deal with hardware interrupts or with the interrupt-like occurrences found in many minicomputer operating systems. At the same time we had to accommodate the needs of the sophisticated user who still desired to take advantage of RMX but had a specific need to directly control the hardware via the interrupt facility.

Finally, to define the general functionality of RMX we examined its anticipated applications. Real-time applications commonly need to perform a number of tasks of differing importance logically in parallel, with preference always being given to executing the most critical ones first. While these tasks may be relatively independent, they may need to periodically synchronize themselves with one or another distinct task or with the outside world. For the latter, interrupts are the usual hardware supplied mechanism. Some tasks may also need to communicate data with one another. For example, a task servicing a sensing device may take readings from the device which need to be communicated to two tasks: one task which reacts to the reading by controlling some other device, and another task which logs or tabulates the readings. Ranked in order of importance these might be control, sensing, and logging. Finally, the tasks must have the ability to control themselves relative to real-time, either by delaying their execution for certain periods or by guaranteeing that they are not indefinitely delayed by, for example, a faulty device.

Requirements on the system design were also generated by considerations internal to the design project. One of these was the need to provide a single RMX abstract machine on a variety of underlying SBC boards. While separate versions of RMX for each board could have been designed with the same external appearance, this approach would have led to an unnecessary amount of internal engineering. Additionally, without careful initial design, the differences in the base hardware would have had visible effects on the RMX abstract machine for each of the boards. This requirement demanded that we partition the structure of RMX into two parts. One part would implement those aspects which were independent of the particular hardware. The second part would interface the first part to the underlying hardware of the specific boards [7].

We also wished to minimize the software development costs by applying the best available software engineering techniques. Historically, tight space constraints have often led to a very *ad hoc* approach to software design in the belief that more generally designed external features or more modularly built internal designs would lead to inherently larger systems. As a result of this philosophy, each needed function is designed to be as small as possible. Unfortunately, while each function may be locally optimized, it is possible that the overall design suffers from duplication or overlap between such individual elements. Current work in programming methodology stresses modularity, generality, and structure (most often for their side effects in producing more maintainable, less error prone systems).

We felt that there was more to gain, both in development cost and space performance, by avoiding optimized specialization of function in favor of more general designs [17]. This reduced the number of separate functions that RMX had to supply. The resulting external design therefore has a single mechanism that provides task communication, synchronization, time references, and standard interrupt-like control. To do so it incorporates the operating system design approaches favored in much of the modern computing literature. Likewise, the internal structures are highly modular and designed to be as uniform as possible so as to avoid replicating similar, but nonidentical internal management routines.

V. The RMX Machine

B. General Concepts

The abstract machine defined by RMX augments the base microprocessor by introducing some additional computational concepts. We define a *task* to be an independently executable program segment. That is, a task embodies the concept of a program in execution on the processor. RMX permits multiple tasks to be defined which can run in a parallel, or multiprogrammed, fashion. That is, RMX makes individual tasks running on one processor appear to be running on separate processors by managing the dispatching of the processor to particular tasks. The registers on the processor reflect the activity or state of the running task. Other tasks may be ready to execute but for some reason have not been selected to run yet and so have their processor states saved elsewhere in the system. From the point of view of the program that is a task, execution proceeds as though it were the only one being run by the processor. Only the apparent speed of execution is affected by the multiprogramming. From the point of view of the system, every task is always in one of three states: running, ready, or waiting. The task actually in execution is running. Any other task which could be running but for the fact that the system has selected some other task to actually use the processor, is ready. Tasks which are delayed or stopped for some reason are waiting, as will be discussed below.

Each task is assigned a *priority* which determines its relative importance within the system. Whenever a decision must be made as to which task of those that are ready should be run next, the one with the highest priority is given preference. Furthermore, in the spirit of unifying mechanisms, the same priority scheme replaces a separate mechanism for disabling interrupts. Interrupts from external devices are logically given software priorities. If the applications system designer deems a particular task as of more importance than responding to certain interrupts, he can specify this by simply setting the RMX priority of that task to be higher than the RMX priority associated with those given hardware interrupts. It is thus possible to maintain a high degree of control over the responsiveness required for various functions.

As mentioned above, tasks may desire to communicate information to one another. To this end the RMX machine defines a *message* to be some arbitrary data to be sent between tasks. To mediate the communication of messages it defines an *exchange* to be the conceptual link between tasks. An exchange functions somewhat like a mailbox in that messages are deposited there by one task and collected by another. Its function is complicated by the fact that a task may attempt to collect a message at an exchange that is empty. In such a case the execution of that task must be delayed until a message arrives. Tasks that are so delayed are in the waiting state. We chose this indirect communication mechanism over one which directly addresses tasks because it permits greater flexibility in the arrangement of receiver and sender tasks. The anonymity of the receiving task implies that the sender need know only the interface specification for a function to be performed via a message to a particular exchange. The task or tasks which implement that function need not be known and hence may be conviently changed if desired.

The conventional mechanism used by programs to communicate with external devices is the interrupt. Unfortunately, interrupts are by nature unexpected events and programming with them tends to be error prone. The essential characteristic of an interrupt is that a parallel, asynchronous activity (the device) wishes to communicate with another activity (a program). Since this communication is essentially the same as that desired between separate software tasks it seems conceptually simpler to use the same message and exchange mechanism for it. The unification of all communications functions is analogous to the idea of standardized I/O found in systems such as UNIX [17]. The RMX machine eliminates interrupts by translating them into messages which indicate that an interrupt has occurred. These messages are sent to specific exchanges associated with particular interrupts. Tasks which "service interrupts" do so in RMX by attempting to receive a message at the appropriate exchange. Thus, prioritized nested interrupts are easily handled. An advantage of this unfied treatment of internal and external communication is that hardware interrupts can be completely simulated via another software task. This facilitates debugging and permits easy modification of a system by allowing rather arbitrary insertion of tasks into a network of communicating tasks and devices.

Note that with this scheme unexpected interrupts do not cause particular difficulty. For example, if the servicing task is still busy with some previous message, the interrupt message will be left at the exchange and will not affect the task until it is ready for another interrupt; i.e., until it waits at the exchange. In an application designed to properly handle the actual interrupt rate, the task will service interrupts quickly enough to always be waiting when the next one occurs. In this case, response to an interrupt is immediate. Thus this mechanism provides no loss of facility relative to the usual interrupt scheme but it does make the proper controlling of such events simpler. Multiple occurrences of the same interrupt which indicate the processor has fallen behind in its servicing are logged as such by a message which indicates that interrupts may have been lost. These interrupts do not, however, disrupt the running task or complicate programming.

The last concept embodied in the RMX abstract machine is that of time. The RMX machine defines time in terms of *system time units*. It then permits tasks to delay themselves for given periods of time so that they can synchronize themselves with the outside world. It also permits tasks to guard against unduly long delays caused by attempting to collect a message at an empty exchange by limiting the length of time that they are willing to spend waiting for some message to arrive.

B. Data Objects and Functions

These concepts are realized in RMX by introducing some new data objects and instructions. Just as the base processor can deal directly with such data objects as 8 bit bytes or unsigned integers, the RMX abstract machine can deal directly with the more complex data objects: task, message, and exchange. Each of these data objects consists of a series of bytes with a well defined structure and may be operated upon only by certain instructions. This is completely analogous, for example, to a machine that permits direct operations on floating-point data objects which consist of four bytes with a particular internal structure to represent the fraction, exponent, and signs. In each case there are only certain instructions that can be used correctly with the object and the internal structure of the object is not of particular interest to the programmer.

The new instructions provided by RMX are: SEND, WAIT, ACCEPT, CREATE TASK, DELETE TASK, CREATE EXCHANGE, and DELETE EXCHANGE. The create instructions accept blocks of free memory and some creation information to format and initialize the blocks with the appropriate structure. Each corresponding delete instruction accepts one of the objects and logically removes it from the system. The remaining operations are of more direct interest to the operation of the RMX machine.

The WAIT instruction has two operands: the address of an exchange from which a message is to be collected and the maximum time (in system units) for which the task is to await the arrival of a message. The result of the operation is the address of the message which was received. A special message from the system indicates that the specified amount of time elapsed without the arrival of a normal message. From the programmer's point of view this instruction simply executes and returns the specified result. Actual execution of the instruction will involve the delaying of task execution if no message is available, by queueing it in a first-come-first-served manner at the exchange. Any such delay is not visible to the programmer, however. This approach unifies the communication and timing aspects of the design. It directly provides reliability in the face of lost events due to hardware or software failure. Tasks can be guaranteed not to be indeterminately delayed due to such failures and can thus attempt recovery from them. It also permits tasks to use the same mechanism to delay themselves for given time intervals by waiting at an exchange at which no message will ever arrive.

The ACCEPT instruction is an alternate way to receive a message. It has a single operand specifying the exchange from which the message is to be received and immediately returns either the next message at the exchange or a flag indicating that no message was available. The task is never delayed to await a message in the ACCEPT operation.

SEND also has two operands: the address of a message and the address of an exchange to which the message is to be sent. The instruction queues the message in a first-come-first-served manner at the exchange if there is no task already waiting there. If a task is waiting at the exchange then the instruction binds the message to the task and makes the task eligible to execute on the processor. When the receiving task resumes actual execution the address of the message will be returned to it as the result of its WAIT instruction.

VI. The RMX Implementation

A. Methodology

In this section and the next, we consider some (but certainly not all) details of the actual implementation of the system as illustrations of the design of such software products. We turn first to the methodology applied to the effort and then to some samples of the mechanisms.

To provide the abstract machine just described and meet the other requirements for the system, RMX was implemented as a combination of ROM resident code and some RAM resident tables. Just as a hardware designer uses LSI devices in preference to more elementary TTL components, we chose to use the leverage of a high level programming language rather than elementary assembly code. The system was, therefore, designed using PLM [14], Intel's high-level implementation language. The operations described above appear as procedure calls using the standard PLM calling sequence. The space constraints and a good level of internal maintainability were achieved by maximizing the modularity of the design. The broad independent functions of multiprogramming, communications and control were completely isolated from the board dependent timing and interrupt handling functions. As a result, movement of the system to a new member of the SBC family requires only the reimplementation of these board dependent functions. In addition, data structure of internal and user visible objects were generalized so that single algorithms could deal with any of them. Individual optimizations could have been made in the local design of many parts of the data structures to improve their space or time costs slightly. Such optimizations, however, would have cost considerably more in code space and code complexity [3].

The module feature of PLM was used to simulate the abstract data type concept [4],[13] and enforce information hiding [15], [16]. That is, every data structure used by RMX is under the exclusive control of a single module. The modules supply to each other restricted sets of public procedures and variables. It is only through these paths that agents outside a module may access the data structures maintained by the module. The only assumptions that such outside agents may make about a module and its data structures are those specified by the definition of the public paths.

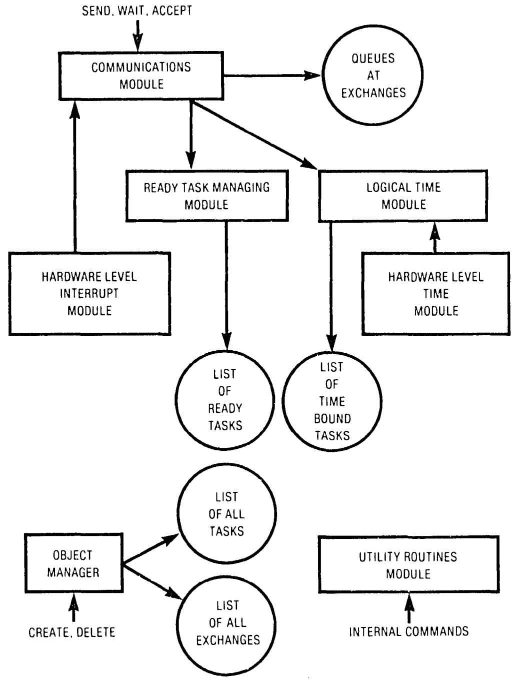

Fig. 2. Major modules (boxes) and data structures (circles) of RMX.

As a result, so long as these interface specifications are maintained, any given data structure may be reorganized by redesigning its controlling module without affecting other parts of the system. This approach improves the understandability of the implementation and facilitates the debugging and maintenance of the system. Fig. 2 illustrates the general structure of the RMX implementation.

Finally, the original version of RMX was completely coded in PLM using the resident PLM compiler of the Intellec® Microcomputer Development System. This version was functionally complete but slightly exceeded the space constraints, occupying about 2.5K bytes of program space. There were a couple of cases where the language structure of PLM did not permit the direct expression of the best way to compile the code. For these modules, it was sufficient to hand optimize the code output by the compiler. The original structure of the PLM program was maintained and the majority of its generated code was used intact. The final RMX system occupies less than 2K bytes of program space. This high level language approach coupled with selective manual optimization permitted far quicker and more error free development than could have been achieved using assembly language.

The approach to handling interrupts did introduce additional software overhead. For a typical configuration of the hardware, the realistic minimum interrupt latency would be about 200 μs. Using the message mechanism it is about 800 μs. For the targetted process control applications, this is entirely acceptable. RMX does make provision, however, for direct handling of selective interrupts which require better response time without disturbing the use of the message mechanism for the others. For normal task communication, the performance is relatively better. For the typical hardware configuration, the transmission of a message takes about 800 μs, which is comparable to the time that would be required for any synchronization primitive (e.g., P and V or enqueue and dequeue) on such hardware.

B. Engineering for Hardware Dependencies

The two functions which vary most significantly across the SBC product line are the timing and interrupt facilities. To accommodate these variations, the implementation separates the logical and physical parts of these functions.

The interrupt facilities are split between the module which implements the communications operations and a hardware interrupt handler module. The communications module provides a special "interrupt send" operation which performs the logical translation of the interrupt event into a message. This facility is independent of the interrupt structure of the processor board and remains the same in any version of RMX. The hardware dependent interrupt module deals directly with the hardware interrupt structure and invokes the send operation at the logical level. Only this module need be redesigned when generating an RMX version for a different SBC board. With this approach we take full advantage of the hardware vectored priority interrupt structure on high performance products and can simulate this desirable structure at slightly higher software cost on low performance products.

The same sorts of variations are faced in providing a source for the system time unit. Again, one module provides all of the logical time functions associated with providing time delays and time limits to the user system. This module is independent of the type, frequency, or location of the physical time source. A separate module is responsible for clocking the logical level by invoking it once every system time unit. Once again, this permits a consistent definition of time in RMX systems regardless of the sophistication of the available time source, and it limits the amount of reimplementation that is needed to support new SBC products.

C. Example Data Objects

As an example of the complex data objects defined in the system we will consider the task and exchange objects illustrated in Fig. 3. The task object is 20 bytes long and embodies the execution state and status of a task. It consists of pointers used to link it onto various lists of tasks in the system. These lists are used to queue a task at an exchange, link it to other ready tasks, or keep track of its maximum delay when waiting. It also contains the stack pointer of non-running tasks which is sufficient to supply the remaining task register values when the task next executes. Finally, the object contains the task priority, some status information describing the state of the task, and a pointer to auxiliary information about the task.

The exchange object is 10 bytes long and implements the mailbox concept described earlier, primarily by serving as the source of header information for lists of messages and tasks. Each of these singly linked lists is addressed with head and tail pointers located in the exchange object. All exchanges in the system are also linked together.

The exchange objects are operated upon by the SEND, WAIT, and ACCEPT instructions of the RMX abstract machine. These instructions generally alter the "value" or contents of these complex data objects. The task object is not the direct operand of any RMX instruction described above. Rather it is indirectly altered as a side effect of various instructions. Just as the user of floating-point objects on most machines needs to know the length and existence of instances of the object, but not its internal structure, so the internal structure of these objects is generally unimportant to the users.

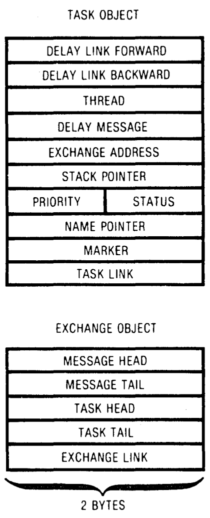

Fig. 3. Example data objects in RMX.

D. Global Versus Local Optimizations

We have already discussed some aspects of global versus local optimizations at the overall design level in terms of avoidance of redundant features. A good example of this tradeoff in the implementation is provided by the linked list data structures within RMX. Like many such systems there are a number of singly linked lists which must be maintained to reflect the status of the system. Local optimizations on the placement of links within data structures or in the form of the headers used for the lists would be guaranteed to save a few bytes of data space across the various lists. Further, the list insertion, scanning, and deletion algorithms could be specially tailored to the individual list structures to save microseconds of execution time for some operations on some lists. Indeed, any one such tailored algorithm might well use less code space than a single more general one.

On the other hand, many of the list operations are in no sense time critical. Generalizing all the list structures to use a single form replaces multiple algorithms with one, thus saving code space. The particular form can be chosen to favor those operations that are frequent, thus limiting the impact of the generalization on the execution speed of the system. Perhaps most important, however, is that, by reducing the number of algorithms and structures used, we decrease the potential number of errors and improve the maintainability of the resultant product. Since there are, for example, at least six distinct singly linked structures in the system, we reduce overall code size and engineering cost by supporting only a single mechanism. We improve product reliability at the price of a small increase in fixed data space and a small execution speed penalty of infrequent and nontime-critical operations.

It is interesting to note as an aside that this is really an example of software engineering: that is, applying engineering discipline to software development. Such discipline is highly valued and understood in other engineering fields. Standardized mechanical or electrical components are virtually always preferred to special designs; PLA's often replace random logic. Unfortunately, an appreciation of the overall benefits of such structure has been slow to develop in software engineering. Too often, we have seen special purpose designs and overly complex structure used in programs supposedly to save space or improve speed. The true costs in development time and reliability of such approaches have often been underestimated; the true time savings attributed to them often overestimated. The high percentage of end product cost due to software is finally forcing a general awareness of these issues.

VII. LSI and Abstract Machines

It seems natural at this point to ask how the abstract machine view of systems in general and our experience with RMX might be affected by the continuing development of LSI technology. Once we view any complex software system as defining a collection of abstract machines, it becomes obvious that it is simply an engineering decision as to which machines should be committed to hardware. We are constrained in this choice by the densities of our solid-state technology, the performance we desire, the applications that we are attacking, and perhaps most severely, by our understanding of software systems and of the machine structures that they require.

We might build an entire final application (e.g., a cash register) as a very-high-level single-chip machine. The specialization of such a design would, however, severely limit its application beyond the one for which it was specifically meant. On the other hand, we could build exclusively bit slice microprogrammable machines with utmost generality but which, due to their very low level of functional integration, would have no technological leverage for attacking complex problems. Actually, both these extremes have their well developed roles and will continue to be reasonable approaches for high-volume low-cost, and special-purpose tailored systems, respectively. It is in the middle ground —the area of the traditional computer—that directions are less clear.

If the 8080 type processors are generally somewhat less powerful than we actually need and as a result we always build operating systems of some level to support them, perhaps some of these functions can be integrated into the hardware. That is, if we can identify a broad range of systems which include essentially the same abstract machine implemented in software, then that abstract machine is a good candidate for hardware integration. The engineering difficulty is in understanding these software structures well enough to confidently and correctly commit them to hardware.

Attempting to build all of some very large and complex operating system onto one or two chips is, no doubt, out of the question with current technology. On the other hand, the final RMX system which we described resides in a small amount of ROM within the 65K address space of the 8080 processor. Once we view RMX as an abstract machine, the placement of the code which implements its functionality becomes immaterial. In particular, we could build an augmented 8080 type processor directly by defining the additional instruction codes of RMX as hardware operations and moving the RMX implementation into microcode on the chip. The resultant component would indeed be an "RMX machine" which dealt directly with the complex data objects and tables described above. It would have the advantage of not using any of the address space for operating system code. More importantly, it would not waste bus cycles and memory access time fetching operating system instructions. Such a machine would have the same advantages over a conventional one that a machine with floating-point hardware has over one without it.

Should we then try to build the RMX machine—ignoring for the moment whether our hardware technology is capable of it quite yet? Is the simple task model of RMX sufficiently general to be of use over a wide class of applications? Is the RMX machine the complete tool that we would like? Clearly, the answer is not a wholehearted yes. For one example, RMX provides no isolation or protection of one task from another. Indeed, no solely software system can provide such protection at any reasonable cost. Such isolation would be desirable at the least because it would limit the damage that one task could do to another due to errors. The conclusion to be drawn, therefore, is not that this particular abstract machine should be built in hardware, but rather that some such machine would provide more of the facilities needed for building microprocessor applications than do current processors. Further, the design principles discussed above are the ones that appear most likely to be fruitful in creating such a machine.

VIII. Conclusions

In this paper, we have attempted to use a case study of a particular small operating system to illustrate both a philosophical approach to viewing computer systems and some important aspects of software development methodology. Many of the subtle aspects of desiging software to control quasi-parallel activities have not been discussed in detail, nor have we fully described the implementation. Nevertheless, we hope that this description suggests the practicality and necessity of disciplined approaches to software system design. Until software implementation reaches a level of engineering commensurate with that applied to other aspects of computer system design, our products will be very much bound by software costs. Only discipline and structure within our software efforts will ultimately permit microprocessor applications to reach their full potential.

Acknowledgment

The author acknowledges the effort of codesigner K. Burgett in the original development of the system. In addition, thanks are due for the detailed suggestions received from J. Rattner, S. Fuller, R. Swanson, G. Cox, and J. Crawford, which greatly improved the content and clarity of the paper. The author also thanks his other colleagues at Intel and the reviewers who contributed to the final form of the paper.

References

[1] P. Brinch Hansen, "The nucleus of a multiprogramming system," *Commun. ACM*, vol. 13, no. 4, pp. 238-241, Apr. 1970.
[2] ——, *Operating System Principles*. Englewood Cliffs, NJ: Prentice-Hall, 1973.
[3] F. P. Brooks, Jr., *The Mythical Man-Month*. Reading, MA: Addison-Wesley, 1975.
[4] W. L. Brown, "Modular programming in PL/M," in. *Proc. IEEE Conf. Computer Software and Applications*, Nov. 1977.
[5] K. Burgett and E. F. O'Neil, "An integral real-time executive for microprocessors," *Computer Design*, vol. 16, no. 7, pp. 77-82, July 1977.
[6] E. G. Coffman, Jr., and P. J. Denning, *Operating Systems Theory*. Englewood Cliffs, NJ: Prentice-Hall, 1973.
[7] G. W. Cox, "Portability and adaptability in operating system design," Ph.D. dissertation, Purdue Univ., Lafayette, IN, Dec. 1975.
[8] P. J. Denning, "Third generation computer systems," *Computing Surveys*, vol 3, no. 4, pp. 175-216, Dec. 1971.
[9] E. W. Dijkstra, "The structure of the 'THE'-multiprogramming system," *Commun. ACM*, vol. 11, no. 5, pp. 341-346, May 1968.
[10] J. H. Fasel, "Abstract machine hierarchies for programming language implementation,"Ph.D. dissertation, Purdue Univ., Lafayette, IN, Dec. 1977.
[11] A. N. Habermann, *Introduction to Operating System Design*. Chicago, IL: SRA, 1976.
[12] A. N. Habermann, L. Flon, and L. Cooprider, "Modularization and hierarchy in a family of operating systems," *Commun. ACM*, vol. 19, no. 5, pp. 266-272, May 1976.
[13] B. Liskov and S. Zilles, "Programming with abstract data types," *SIGPLAN Notices*, vol. 9, no. 4, pp. 50-59, Apr. 1974.
[14] D. D. McCracken, *A Guide to PL/M Programming for Microcomputer Applications*. New York: Wiley, 1977.
[15] D. Parnas, "A technique for software module specification," *Commun. ACM*, vol. 15, no. 5, pp. 330-336, May 1972.
[16] ——, "On the criteria to be used in decomposing systems into modules," *Commun. ACM*, vol. 15, no. 12, pp. 1053-1058, Dec. 1972.
[17] D. M. Ritchie and K. Thompson, "The UNIX time-sharing system," *Commun. ACM*, vol. 17, no. 7, pp. 365-375, July 1974.
[18] *RMX/80 System Users Guide*. Santa Clara, CA: Intel Corp., 1977.