To the Graduate Council:

I am submitting herewith a thesis written by Peter V. Andreae entitled "An Environment for Process Control Tasks Under RSX-11M." I have examined the final copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Science.

Kevin C. O'Kane, Major Professor

We have read this thesis
and recommend its acceptance:

Accepted for the Council:

The Graduate School

AN ENVIRONMENT FOR PROCESS CONTROL TASKS UNDER RSX-11M

A Thesis

Presented for the

Master of Science

Degree

The University of Tennessee, Knoxville

Peter V. Andreae

June 1984

ACKNOWLEDGMENTS

I would like to thank Dr. Kevin O'Kane for his guidance, support, and friendship during the course of this project. I would also like to thank Dr. David Straight and Dr. Robert Heller for their coursework and support as committee members.

Thanks must also be expressed to Tech-Con Inc. and its employees. In particular, to Becky Duncan who has encouraged and helped me throughout my professional career and to Bill Allen and Bruce Dusseault who provided their ideas and suggestions throughout the design and implementation of this project.

The terms DEC, PDP-11, Unibus, and RSX-11M are trademarks of the Digital Equipment Corporation.

ABSTRACT

Computer programs for the control of industrial processes require specialized functions for the synchronization of their actions in real time. A real-time operating system can provide sertain of those functions but often only at a low level. A library of functions and auxiliary tasks was designed an implemented to allow tasks operating under the RSX-11M operating system to conveniently and consistently control respond to and control events in the an industrial environment.

Of particular concern were functions to serialize and synchronize many interrupts generated by a diverse set of sources. A means of presenting these interrupts to the user tasks was devised which masks the differences in the sources of interrupts while making available all the information represented by each interrupt.

The implementation was successful although it suffered from some inefficiencies inherent in the design of the underlying operating system.

TABLE OF CONTENTS

SECTION	PAGE
I. INTRODUCTION	1
II. THE PROBLEM OF PROCESS CONTROL	4
III. PCS AND RSX-11M OVERVIEW	8
PCS Concepts	8
RSX-11M Concepts	17
IV. PCX IMPLEMENTATION	21
Device ID's	22
Basic Intertask Communication	23
Interrupt Structure	24
S1500 I/O	29
Inventory System	34
Process Interlock Functions	36
The Menu System	38
V. CONCLUSIONS	44
Problems	44
Enhancements	48
LIST OF REFERENCES	51
APPENDIXES	53
APPENDIX A. DEVICE IDENTIFIER STRUCTURE	54
APPENDIX B. PCLIB FUNCTION DESCRIPTIONS	58
APPENDIX C. PCLIB FUNCTION DEFINITIONS	77
VITA	98

I. INTRODUCTION

This paper describes an environment for process control tasks operating under Digital Equipment Corporation's RSX-11M operating system. This environment provides the following services:

1. A set of high level task synchronization and communications functions.
2. An interface to process hardware.
3. Functions for controlled access to a set of shared variables.
4. A menu style operator interface.

This software was designed and written by the author while he was employed by Tech-Con Inc. of Knoxville, Tennessee. It is currently being used in two installations, a fluid milk processing plant and a soft drink bottling plant. There is no standard name under which this product is marketed so for purposes of clarity it will be referred to as PCX in this paper,

Tech-Con sells manufacturing control systems to the food processing industry, particularly operations involving fluid products. In 1980 the primary system marketed by Tech-Con was implemented on a PDP-11 processor running under a proprietary operating system called PCS. The systems used a line of process I/O hardware, manufactured by Tech-Con, called System 1500 (S1500). The PCS operating system was written several years ago and was designed to run on PDP-11's which do not support memory management. As a result, system memory size is limited to 28 KW. PCS is disk based and requires either a dual floppy disk or a single RL01 disk drive. PCS programs are written in Macro-11 and developed under the RT-11 operating system.

Over the years of use of PCS the processing operations controlled grew in complexity and size. The memory limitations of PCS began to have an adverse effect on the performance of the systems and indirectly, decreased their reliability. In addition, the task of writing and maintaining, in Macro-11, the large number of programs required for a single installation became monumental. As a result it was determined that a new system should be designed with the following goals in mind:

1. Use of a standard, proven operating system to relieve Tech-Con of the task of operating system support.
2. Use of a high level language for both systems and applications software to speed the development process and to enhance the reliability of the system.
3. The ability to do software development on-line while the control system was in operation.
4. Support for the S1500 line of process hardware.
5. Use of PDP-11 processors to protect Tech-Con's investment in hardware.

RSX-11M was the operating system of choice. It is a standard DEC product which has been in use for several years and as a result has become fairly stable and reliable. It is designed for multitasked, real time applications yet it supports a full multiuser program development environment. This is in contrast to RT-11, another DEC operating system which is also designed for real time applications but which only supports a single user.

The languages supported under RSX by DEC are Macro-11, Fortran, and Cobol. Macro-11 and Cobol were dismissed immediately, leaving Fortran.

Fortran is widely used for process control applications and enjoys widespread recognition and acceptance. These factors make it attractive from a marketing point of view. Nevertheless, Fortran lacks the versatility of some more modern languages and the search was extended to compilers produced by others than DEC.

The two languages which received the most consideration were Pascal and C. Both were enjoying a great growth in popularity in the computing community at the time. A Pascal compiler to run under RSX had been available for some time from OMSI and in fact had just been released in an enhanced version. One feature of this compiler was the ability to embed Macro-11 code in a Pascal program. This capability could be useful, particularly in writing such things as hardware drivers. To its detriment, Pascal suffered from an intuitive distaste among the programming staff at Tech-Con which ultimately led to its dismissal from consideration.

One member of the programming staff had had considerable experience with a C compiler from Whitesmith. It supported the full language, including the preprocessor features, which was rare at the time. It had been through several updates and had proven to be reliable. C is a compact language, yet it contains a full complement of modern control structures. Its type checking is relaxed, allowing the knowledgable programmer a substantial amount of flexibility in the manipulation of data with incompatible formats. In view of these attributes and relying on the experience of one person, C was chosen as the language for implementation of the PCX system.

II. THE PROBLEM OF PROCESS CONTROL

To help us better understand the requirements of a control system let us examine the types of processes to which Tech-Con applies its control systems. The classical control problem is as follows. The state of the process to be controlled is defined by a set of input variables which may change at any time. The goal of the control system is to maintain the values of these input variables within a specified range by modifying the values of a set of output variables. A typical example is the control of a furnace. In a simplified case the problem may be stated in the following manner: Every ten seconds read the temperature of the furnace. If the temperature is too high, reduce the flow of fuel to the furnace. If the temperature is too low, increase the flow. The important problem in this example is designing an algorithm for determining the change to make to the flow rate, given the size of the deviation from the desired setpoint, which will result in a stable system.

While this is a common sort of control problem let us examine a process more typical of those encountered in the food processing industry. We want to control a carton filling machine which may be either on or off. The operator has a switch which allows him to control the process. When the switch is on and the tank supplying the filler is not empty we want to turn on the machine. Otherwise the machine should be off. In this problem the output of the process is simple, either turn the output on of off. The majority of the problem to be solved lies in monitoring the operator input and the tank level.

The simplest solution to this problem requires that we periodically test the state of the operator's switch and the level in the tank. If we find that either one has changed since the last scan, we adjust the state of the output which controls the machine. This is an adequate solution if we have a small number of machines to control or if we have unlimited resources in our control system. In general, however, we have many processes to control and a limited amount of computing power. Obviously we can support a much larger number of processes if we eliminate the time spent monitoring the inputs. In view of this, let us reword the problem statement: When the operator's switch changes state or the amount in the supply tank changes, update the state of the output to reflect the new state of the inputs. Now our system is interrupt driven. The control procedure lies dormant until some change of state occurs in the inputs from the process. At that time, the actual control algorithm is executed.

Although the preceding example is simple it is sufficient to point out some of the important factors which we must consider in the design of an environment for process control programs.

The most important data for a program controlling a discrete process such as the filling machine described above are interrupts or changes in state of the input variables. Therefore, the environment in which the program operates must provide a simple and convenient means of specifying interrupts and communicating them to the control programs. Because interrupts can come from many sources--sensors and switches in the machinery being controlled, timers, operator controls--we must also be careful that our interrupt scheme is consistent for all types of

interrupts. At the user's level the differences in the types of interrupts must be masked so that all events can be handled in a similar manner.

Once a program has become aware of an interrupt it must have some means of producing outputs which can influence the state of the process being controlled. In particular, there must be functions provided which can effect any of the operations available in the process hardware. We would like these functions to be "high level" in the sense that they appear in programs in terms which resemble those used in the specification of the process. For instance, it would be nice to be able to turn on output valve 100 with a command like
"ON (V100)" or to test the state of binary input 3 with "TEST (INP3)".

Another important communications channel which must be defined carefully is the operator interface. Every process in a plant must ultimately be controlled by a human operator. Therefore operator communication is very important. It can also be one of the most difficult functions to provide. While a programmer is often content to admit that he cannot predict the way in which a sequence of events will occur in the plant, he will often fool himself into thinking that he can predict what an operator will do. Unfortunately, the possibilities are legion and are not subject to the laws of reason. We must build a structure for the operator interface which provides reasonable functionality while constraining the possible inputs which the operator can generate.

Finally we must realize that a plant consists of many subsystems which are essentially independent but which must share some data. We

would like to reduce the complexity of the software by implementing each subsystem in a separate program. This means that there must be some means of communication between programs. In particular, we need to be able to transmit two kinds of data. First, the progress of one process may depend on the state of some other process. Therefore, there must be a means for a program controlling one process to inspect the state information of a process controlled by some other program. Second, it is often important that one process be informed of the change in state of some other process. Therefore, we would like to have some communication means similar to an interrupt from a physical device by which one process can communicate a change in its state to another.

III. PCS AND RSX-11M OVERVIEW

Before we look at the new environment which was created for operation under under RSX let us examine some of the features of PCS and RSX.

PCS Concepts

PCS is worthy of study because it was created exclusively for process control applications. A great deal of experience was gained in using it over the years and many of the solutions to process problems which evolved may be useful. Some other solutions can be described as bad and we study them so that they may be avoided in the future. In addition, many of the assumptions were carried by the author, whether intentionally or not, to the new system.

PCS is a disk based operating system designed for dedicated use in a process control environment. It runs on an unmapped PDP-11. The operating system proper occupies approximately 36 Kb, leaving the rest of the available 56 Kb for applications programs and the dynamic memory needs of the system. Because the processors on which it is able to run do not have memory management capabilities, PCS provides no memory protection for either the operating system or the applications programs.

Development of applications software is done under RT-11 in Macro-11. Once a program has been assembled and link edited in the normal fashion it must be built onto a PCS system disk by a special build program. This program maintains a directory on the system disk which is incompatible with the RT-11 directory structure. PCS offers no on-line development facilities beyond a disk patch program.

The independent unit of applications software is a task which has a maximum size of 512 bytes. A PCS task image is normally called an overlay and is identified by a ten bit overlay number. A task is an instance of a task image which has been loaded into memory and given a context. The code in a task image may be designated as reentrant, reusable, or non-reusable. Because tasks may be reentrant, multiple tasks may share one copy of a task image in memory. On the other hand, if an image has been designated as not reentrant then multiple tasks using the same image must have individual copies of the image in memory. Once a task is initiated using a particular copy of an image in memory, that image remains in the memory block which it was allocated when first read in until all tasks using it have exited. Only then may the block be reclaimed by the system and used for another task or for some other purpose. Because memory blocks are assigned to tasks on an as available basis, the code in a task image must be position independent.

Each task image has a name, called its overlay number, which translates into a ten bit binary value. These overlay numbers are associated with symbolic ASCII names at assembly time so that applications programs may refer to tasks by name. A flaw in the PCS task naming scheme is that a task has the same name as the task image which it executes. There is no way for a program to distinguish among several tasks with the same name. In particular it is impossible to abort one instance of a task without aborting all instances of it. This is a concern because, under PCS, aborting is often the best way to ensure synchronization between, for example, the start routine for a process and the the stop routine for that process.

Interrupts

The systems controlled by PCS are designed to be almost entirely interrupt driven. Therefore, an important function of the operating system is the routing of interrupts to applications tasks. Interrupts can be generated by process I/O hardware, inventories, timers, and in some sense, tasks. There are three important points to keep in mind concerning PCS interrupts. First, the prime purpose of an interrupt is to initiate a task. In fact, the only way to initiate a task is via an interrupt. Second, any interrupt can be directed to any task. Third, each I/O point in a S1500 interface is treated as an individual device and each device has its own interrupt which is independent of all other devices.

When a task is initiated it is provided with the name of the entity which caused it to run, whether it is a hardware device, an inventory, or another task. (Delay timeouts look like starts by the task which set the timer running.)

The function which associates an interrupt with a task and tells the system that an instance of the task should be initiated when that interrupt occurs is called activate, in its generic form. It has specific forms which depend on what sort of device is being activated. There is also the inverse function, deactivate, which erases the association between an interrupt and a task.

S1500 I/O

As was mentioned before, PCS supports the S1500 line of process I/O hardware. The system macro library provides an extensive set of functions for all of the operations performed by the hardware.

The name of each individual I/O point connected to the system is defined when the system is generated and can be used in a program to refer to the point. From the point of view of the programmer, a hardware name is a global symbol which can be the argument of an operating system request. The value of such a symbol is called a binding number. Rather than being a description of the physical location of an individual piece of hardware in the racks or on the Unibus, a binding number is an index into tables maintained by the operating system which describe the physical structure of the hardware.

The value of a binding number is dependent only on the type of device and the order in which that device is defined in a table relative to other device of that type. In this way an applications program is made independent, in some sense, of the physical arrangement of hardware on a system. This is particularly useful if one has an in-house system used for testing of software for several different installed systems. The same applications software can be run on the in-house system as is run on a particular on-site system without any modifications to the software itself. Furthermore, the hardware on the in-house system need not be rearranged to mimic each on-site system. The definition table for each system need only be modified to reflect the physical arrangement of the in-house hardware.

The concept of configuration independent naming of process hardware is one which we would like to retain from PCS. But while we are examining it, let us note an improvement which may be useful. Notice that the binding of a hardware name to its value is done at link time. This means that if there is a change to the hardware table every task must be relinked to reflect the change. If we defer the binding until run time, the task image is made completely independent of any hardware configuration.

Inventories

An entity maintained by the operating system which should be considered with the I/O and interrupt system is the inventory. Since PCS was developed to control operations concerned with the processing of fluids it has many functions to support data structures which describe storage tanks for those fluids. These data structures are called inventories and are contained in the software table. Each inventory has a name whose value as a global symbol is actually the address of its first word. The applications programs can refer to inventories by name and the references are resolved at link time.

Inventories are rather unusual data structures in that they can generate interrupts. The main components of an inventory are the accumulator and one or more limits. The system accumulates counts from some input device such as a turbine meter in the accumulator of an inventory. The programmer can place values in the limits of the inventory and request that the system generate an interrupt when the value in the accumulator exceeds the value of the limit. In general an

accumulator can be set up to count up or to count down and limits can be endowed with any of the standard comparison types, GT, GE, LT, LE, EQ, or NE. The implementation of inventories under PCS is well thought out and the concept itself is very useful. We would like to transport as much of the system as possible to our new implementation.

Operator Communications

PCS provides very little support for communication with the operator beyond the output of messages to one or more serial terminals. It is possible for a task to read unsolicited characters from a terminal but that feature was added to the system as an afterthought and is very inefficient and error prone. The operating system does provide some command processing but adding a new command is a difficult process and requires a regeneration of the system.

Rather than using serial terminals for operator communications, most operator commands are generated by pushbuttons and rotary switches. Both devices are supported by the S1500 hardware and the operating system. Most plant processes are assigned a pushbutton and a multi-position rotary switch. The positions of the rotary switch are labeled to correspond to one of several tanks that the process may use. To start a process the operator positions the rotary switch to the desired tank and presses the pushbutton. The interrupt from the pushbutton initiates the startup task for the process. To stop the process, the operator turns the rotary switch, generating another interrupt which starts the stop task.

The problems with this approach are several. First, the operator can provide only a very limited amount of information about how the process is to be run through the rotary switch. Second, the only feedback the operator gets to indicate that the process has started or stopped successfully is through a pilot light mounted in the pushbutton. Third, the hardware to implement this approach can be very expensive. Not only is the cost of the S1500 rotary switch module high, but the switch itself is expensive and there must be a significant amount of field wiring to each of them. We would like to discard this method of operator interaction for something which is more versatile, and is cheaper.

Task Communication

PCS tasks communicate by two means, through the system "software table" and through the task initiation functions.

The software table is a portion of memory set aside for use by all tasks. At system generation time variables can be declared within this area. The definitions of these variables are then available to the applications tasks as global symbols. There is no means of protecting individual locations within the software table. Nor are there any functions for synchronizing access to a particular location. Nevertheless, this is the most convenient place a task can store data which must remain across invocations of the task.

The items of system-wide significance which are stored in the software table are process status, inventories, and process variables. Most updates to the inventories are done by the operating system or by

the tasks via system calls so write access to them is fairly well controlled. The process variables are a group of variables defining such things as delay times which can be changed by the operator to affect the operation of the plant processes. These values are fairly static and are never changed by applications software, so synchronization of access to them is not critical. This leaves us with process status to consider.

Due to the small size of PCS tasks it is common that several tasks are required to control a single process. As a result several tasks may make changes to a single process status. Since the PDP-11 has no hardware mechanism for synchronizing access to shared variables (e.g. Test and Set), tasks must use some other means for achieving this. They rely on the priority manipulation functions available in the system. A task may set its own priority dynamically to any value in the range of 0 to 7. Seven is the highest level and it assures the task uninterrupted use of the entire machine. Thus, when a task needs to make some modification to the state of a process it must raise its priority to 7 before doing it to get some assurance that it will be unmolested.

Although this technique will work, it must be carefully applied to be reliable. There is the ever present danger of the programmer leaving a task at level 7 or forgetting to change its priority at all. In addition, several critical system services will lock up the system if invoked by a task at priority 7.

Task communication via the task initiation functions depends on the fact that when a task is started by another task it appears to the requested task that it has received an interrupt from the requesting

task. The requested task has available the name of the task which started it and the values in its registers 3, 4, and 5 are the same as those for the starting task.

There are two types of task initiation functions. The simpler one is called START. It simply requests the initiation of a new instance of a task using the specified overlay or task image. The requesting task continues execution in parallel with the new task. The more complex version is the CALL. This performs the same function as the START except that a link is created between the two tasks. The CALLing task may the execute a WAIT which suspends its execution until the CALLed task exits or executes a RETURN. Tasks may be treated as either subroutines or coroutines in this manner.

Several rather complex techniques were developed to give structure to a group of PCS tasks controlling a single process and to avoid some of the problems of synchronization. In these techniques a "scheduler overlay" was written to coordinate the activities of the tasks. All interrupts within the process were channeled through the scheduler who queued them up and dispatched tasks serially to process them. In this way it was hoped that it could be guaranteed that only one interrupt was serviced at a time by a process and that as a result there would be no conflict in access to the process status.

The PCS schedulers were only moderately successful. They achieved their goal of serializing interrupts within a process and for this reason we will remember them and return to the concept later. The goal was achieved however at a large expense. Because there is no sharing of code among tasks, a great deal of code had to be replicated in each task

in a subsystem. As a result each task could do less real processing and the number of tasks required to manage a single process grew. The increased number of tasks led to higher demands for memory and higher levels of disk activity. The absolute limit on memory size imposed by the lack of memory management support and the lack of any facilities for checkpointing waiting tasks soon led to unrecoverable deadlocks.

RSX-11M Concepts

We must also examine RSX since it is the baseline from which we must work. There may be many approaches which we would like to take in solving the problems at hand which are made difficult or impossible because RSX does not provide a means by which to implement them. Any solutions we choose will be colored by the underlying structure provided by RSX. Our task is to select the useful functions from those available and from them to build a set of functions which can express more naturally and precisely our view of the operations of the real world.

Interrupt Structure

The two important means by which an RSX task may be informed of some external event are the Asynchronous System Trap (AST) and the setting of an event flag. The AST resembles the classical interrupt. When it occurs, the normal execution of the task is interrupted and restarted in an AST service routine. When the task has completed servicing the interrupt it performs and AST exit and returns to its old execution path. AST's can be generated by several types of events. In particular, we might mention timer expiration, I/O completion, and

receipt of a message from another task. A nonprivileged task cannot generate an AST in another task directly.

There are several problems associated with using AST's directly. Each type of AST requires a service routine. When the routine is entered none of the general purpose registers except for the PC are saved. In addition some data associated with the event are pushed onto the stack. The AST routine must be careful to preserve the state of the registers and maintain the stack configuration. Failure to do this can result in obscure errors.

The setting of an event flag can also signal an external event to a task but unlike the AST it does not interrupt the execution of a task. The task must explicitly test the state of an event flag to determine its state. A task may also request to stop until an event flag becomes set, in which case execution continues at the instruction immediately following the request when the flag gets set. Event flags are set by the completion of almost all of the system directives. In addition, they may be set and cleared directly by tasks.

Event flags are no less prone to misuse. In fact Digital prints a warning about their use. "Take great care when setting or clearing event flags, especially common flags. Erroneous or multiple setting and clearing of event flags can result in obscure software faults."[Digital. P. 2-2] The programmer is saddled with the task of assigning real world events to specific event flags. He must also coordinate those assignments with other processes in the system when common flags are used. In addition, the number of event flags is limited and the supply could easily be exhausted before every event has been assigned a flag.

Intertask Communications

The primary means of transmitting substantial amounts of information (beyond the single bit represented by an event flag) between RSX-11M tasks is the Send Data directive. This sends a 13 word block of data from one task to another. The receiving task may inquire of the system whether there are any messages waiting or it may request than an AST be generated when a message is received. The only criticism of this mechanism which we can mention at this point is the fixed and limited size of a message packet. We will return to the Send Data concept later and expand upon it to provide a basic mechanism in our new environment.

I/O Structure

The last aspect of RSX which we will examine is the I/O structure. All I/O within the system is controlled by device drivers. These are asynchronous system processes, not tasks, which are mapped to the Executive and to the I/O page. Normally, any program which wants to process hardware interrupts directly must be written as a device driver.

The structure which the system imposes on I/O processing through the device driver concept is well suited to device types characterized by a relatively small number of similar units such as disk drives and terminals. This is reflected in the data structures which must exist in every device driver. The system performs some preprocessing on all I/O requests before passing them to the driver and expects all devices to be identified in a standard way.

This rigid structure is not as convenient for devices such as digital inputs and outputs however, which may have many similar units.

It is particularly inconvenient for a subsystem such as the S1500 hardware which consists of a variety of of units which are related at the interrupt level but which are not related from the point of view of the programmer. The standard data structures are not suitable for describing the hardware. Furthermore, the RSX device driver concept is not oriented towards generating asynchronous interrupts.

Happily, RSX does provide another mechanism to allow a task to handle hardware interrupts directly. It is called Connect to Interrupt. When a task connects to an interrupt it need specify only the hardware interrupt vector address and the address of an interrupt service routine within the task. The task must be privileged and of course must map to the I/O page.

The price that must be paid for this feature is a small amount of overhead in the Executive to process each interrupt and the loss of the I/O request queueing functions which are provided to device drivers. On the other hand, the interrupt processing task is free to impose any logical structure it desires on the devices it controls.

IV. PCX IMPLEMENTATION

Now that we have established some background and motivation for the PCX system we can describe the implementation. The important points to cover are:

1. Device ids and their binding.
2. Intertask communications functions.
3. Interrupt structure and functions.
4. S1500 I/O functions.
5. Inventory system.
6. Process interlock functions.
7. The menu system.

These items describe the majority of the structure which was imposed on top of RSX.

An applications task gains access to the functions provided by the PCX system via a function library called PCLIB. Each task must include this library in its task build step. It contains all the functions which implement the PCX scheduler as well as the those required for control of the S1500 process I/O system. While these operations comprise the majority of PCLIB, there are also various utility functions for operations unique to the applications environment. Each task must also link to a standard C language library provided by Whitesmith.

While the functions in both libraries are in general, reentrant, each task contains its own copy of each function it uses. RSX-11 does provide the ability to have shared libraries which greatly reduces the memory requirements of each task. At the time the system was created,

however, Tech-Con did not have access to the source programs for the standard library and thus it was impossible to build a shared version.

Each PCX task is also build with standard module which defines the entry point for the task and defines and initializes the scheduler data structures.

Many of the PCLIB functions generate requests to independent tasks which provide services to the applications tasks. In particular the S1500 I/O interface, the inventory system, and the interlock system are implemented in this way. These tasks will be described below.

Device ID's

The purpose of the device id is to provide a consistent means of describing any type of device or event which can be manipulated by a task. A description of its format appears in Appendix A. Device id's can describe S1500 devices, tasks, timers, inventories, and asynchronous serial terminals. Most devices can generate interrupts although that is not always the case. For instance, all timers can generate interrupts but not all variations of the inventory device can.

Device ids used by a program are bound to actual values at run time. Because of the diversity of internal structures there are several functions for this purpose. These are: "s15init" for S1500 devices, "ivinit" for inventories, "tskinit" for tasks, and "timinit" for timers. Each function takes two arguments, the address of the device id which is to receive the value and the literal representation of the name with which it is to be initialized. For instance, to load a variable called valve with the device id of an S1500 output called V100 we could write:

```
DEVID valve;
     .
     .
     .
s15init(&valve, "V100");
```

The actual values for device id's are generated in two ways. Task names and timer numbers are created directly from the literal values supplied. Id's for S1500 devices and inventories are requested from a symbol lookup program called SLU. SLU operates from two definition files read from disk at system boot time.

Basic Intertask Communications

Tasks communicate by sending messages to one another. Sending a message from one task to another transmits 13 words of data and generates an interrupt in the receiving task. In addition to sending the data, this function also "requests" the receiving task so that it will become active if it is not already. The message initiating function is called send and has two arguments: the device id of the receiving task and the address of a buffer containing the message text. The following code fragment illustrates a task which sends a request to print an error message called BADVAL to an error printing task called ERRPRT.

```
DEVID errtsk;
COUNT buf[13];
     .
     .
     .
tskinit(&errtsk, "ERRPRT");        /* id for receiving task */
buf[0] = ERR_PRT:                  /* function code */
buf[1] = BAD_VAL:                  /* message name */
send(&errtsk, buf);
```

The message packet may contain any values. None of its fields are constrained by the system. In fact, the message may be null in the case

that a task simply wants to generate an interrupt in another task. A null message is sent in the following way:

```
send(&errtsk, NULL);
```

The message sent by this invocation of send will consist of 13 words of 0.

Interrupt Structure

The primary goal of the PCX interrupt handling mechanisms is to allow a task to synchronize and serialize the processing of interrupts. In other words, a task should not have to process an interrupt until it is ready to. In particular, while it is processing one interrupt it should not be forced to deal with another one.

As we have seen, the main RSX interrupt handling mechanism, the AST, is inherently asynchronous. A task's AST service routines are presented with each interrupt as it occurs. The system does provide functions which allow the programmer to control AST processing by enabling and disabling all AST's, thereby making the process somewhat more synchronous.

We could force each application task to include its own unique, internal AST processing schemes. Interrupt processing is a complex operation, however. Not only is it prone to errors but each programmer who attacks it will probably produce a different solution to the problem. This leaves future generations with the problem of maintaining multiple unreliable versions of solutions to the same problem. Furthermore, the real task of the applications programmer is to solve the problem of controlling a process, not that of queueing up interrupts.

The interrupt structure which has been implemented for PCX takes into account the following items:

1. Interrupts come from a variety of sources. We would like to mask the differences from the programmer so that all interrupts can be handled by one mechanism in the user task.
2. Because there is no precise way to control when an AST occurs, we must be prepared to process one at any time, including while the task is doing other processing. This implies that there must be some sort of standard, internal queueing mechanism for interrupts.
3. It is often convenient to be able to "look ahead" at the queue queue of pending interrupts.

The different types of devices are made to look alike, in part through the use of the device id. Any interrupting device can be represented by a device id. This is an important concept not only because it allows the user to employ similar mechanisms for all interrupts, but because it makes it easy to define new types in the future without disturbing the existing scheme.

The amount and type of data actually generated by an interrupt varies with the device type. For instance, the receipt of a message from another task produces 13 words while the expiration of a timer produces none at all. As it turns out no event produces more than 13 words so that has been fixed as the standard size of the data buffer passed to a task when it dequeues any type of interrupt.

As interrupts arrive at a task through the AST mechanism, they are queued up by standard internal routines which must be included in every

task. The queue of pending events is referred to as the hold list and is built of elements containing a device id and a 13 word buffer. These elements are called event queue elements and have the following format:

```
typedef struct eque                    /* event queue element */
    L
    DEVID eqdev;                       /* interrupting device */
    COUNT edv[13];                     /* interrupt data */
    1 EQUE;
```

Event queue elements are placed in the hold list in the order they are received. There is no prioritizing of the list. The functions which build event queue elements and place them in the queue are "dlyast", "_nextc", and the various versions of the receive function. Source code listings and detailed documentation of each of these functions appear in appendixes B and C.

An examination of these internal interrupt processing functions reveals that some interrupts are handled asynchronously and some synchronously. This is dictated by the way in which RSX presents the events to the task. For example, each message received from another task generates an AST. The message text itself remains queued up by the operating system until the task explicitly reads it. Thus the task's interrupt queueing mechanism can essentially ignore the AST and read all pending messages from the operating system before it scans its internal queue for the next interrupt.

This mechanism can be seen in the "rcv" and "rdast" functions. All messages from other tasks are read by "rcv" which is the dequeueing function for user level interrupts. Before "rcv" scans the hold list for the next event to process it reads all incoming messages being held by RSX, builds event queue elements for each one, and enqueues them. If

there are no pending events in the hold list, "rcv" issues a Receive Data or Stop (RCST) directive which attempts to read the next message and suspends the task if there are none there.

Receive data AST's are processed by "rdast". When a message is received from another task, "rdast" is initiated by the system. The only action of "rdast" is to unstop the task. This has no effect if the task is not stopped. On the other hand, if the task has been stopped by "rcv" because it had no pending events, it will wake up and "rcv" will repeat its previous sequence of actions.

Asynchronous interrupt handling can be seen in the "dlyast" function. In the case of timer expirations, RSX keeps no queue of unacknowledged timeouts. It simply generates one AST for each one. Thus the AST handler, in this case "dlyast", must enqueue the event in the hold list immediately. It cannot come back later and retrieve the event at its leisure.

In retrospect, this inconsistent handling of the two types of interrupts was probably unwise. In fact it led to a problem in an early version of "rcv". Consider the following situation. A task sets a delay and waits for it to expire. When it expires the task restarts a new delay and does some processing. While it is processing, a message arrives from another task and the delay times out again. The new delay timeout will be enqueued immediately in the hold list by "dlyast" but the message will not be. When the task calls "rcv" to enqueue the next interrupt, the delay event is returned immediately. The message from the other task has still not been read. If the task continues to issue a new delay each time one expires, it is possible that the message will

never be read. This situation explains the presence of the while loop which reads messages at the beginning of "rcv". The more consistent solution would probably have been to enqueue messages when their AST's are received but for some reason that choice was not made.

Now that we have seen the internal workings of "rcv", let us look at how it appears to the user. It returns the next interrupt from the hold list. If there is nothing in the hold list, it suspends the task until an interrupt occurs. There are two arguments, the address of an area in which to put the device id of the interrupting device and the address of an event status block. An event status block contains the 13 word data block and a pointer to the interrupting device id. It is defined in the following manner.

```
typedef struct esb          /* event status block */
     {
     DEVID *esbdev;              /* address of device id */
     COUNT evdat[13];       /* interrupt data */
     } ESB;
```

Three additional variations on "rcv" exist. They are "rcvoc", "rcvex", and "rcvw". "Rcvoc" is similar to "rcv" except that if no interrupts are pending in the hold list it returns immediately, or continues, instead of stopping. The function returns the value of its first argument if there was an event pending and NULL otherwise. This function is useful for a task which performs some background operation and must check periodically for an interrupt, returning to the background function if there is none.

"Rcvex" differs from "rcv" in that if there are no events pending it exits. To understand how this is useful recall that the "send" function sends a packet of information to a task and then requests the

task to run. The "rcvex" function uses the Receive Data or Exit (RCVX) directive. If a user task tried to simulate the "rcvex" function through a "rcvoc" and a separate exit there is a possibility that the following sequence of events could occur:

1. Task 1 calls and returns from "rcvoc", finding nothing.
2. Task 2 sends a message to task 1.
3. Task 2 requests task 1 to run.
4. Task 1 exits.

At this point the message from task 2 is lost. The RCVX directive guarantees that steps 1 and 4 are indivisible thereby assuring that although task 1 may exit, it will be run again immediately when task 2 generates a request.

"Rcvw" implements the look-ahead feature which was mentioned previously. In this case the caller specifies the device id of the device from which it wants to receive an interrupt. "Rcvw" searches the hold list for instances of interrupts from that device. It returns the first one it finds. If there are none, it suspends the task as in "rcv". Notice that no interrupts are thrown away by this function. The "depck" function which implements the reception of variable length messages is an example of an application of this function. The source code for this function appears in Appendix B.

S1500 I/O

The interface to the S1500 hardware is implemented in a privileged task called IOS. The name is borrowed from PCS where its letters stood for Input Output Supervisor and it referred to the portion of the

operating system responsible for all I/O. IOS is privileged so that it may map directly to the I/O page and so that it can use the RSX Connect to Interrupt directive to trap hardware interrupts.

The interface hardware which resides inside the processor comes in two forms, a parallel interface and a serial interface. The names are derived from the manner in which signals are transmitted from the Unibus interface to the process I/O modules. The parallel interface is intended for installations in which the process hardware is in close proximity to the computer so signals are transmitted from the Unibus interface to the modules along a pair of ribbon cables. The serial interface, on the other hand, is intended for use when the modules must be located remotely from the computer. Signals to the module are transmitted on a serial line in this case.

IOS is designed to allow 2 parallel interfaces and two serial interfaces on one system. The current implementation supports only a single parallel interface.

The parallel interface occupies 1 KW on the Unibus. Each physical I/O module appears as 8 words within that space. Thus a single interface can accomodate 16 racks of 8 modules each.

The interface generates four physical interrupts, referred to as interrupt levels A, B, C, and D. Each module is assigned individually to one of these interrupt levels by a selector switch on the Power Supply/Address module in the rack which the module occupies. When it receives an interrupt from the parallel interface it is IOS's job to poll all the modules on that interrupt level to find the one which generated the interrupt.

An important design feature of IOS is that each I/O point on a module can be treated as an independent entity by the applications programs. In particular this means that each point has a unique name and device id by which it can be addressed and that the interrupt from each point can be directed to a different task.

The hardware names and device ids for a system are defined in a file referred to as the hardware table. This file defines the position of each module in the racks and gives a name to each point. It is processed by a program which produces the device id definition file used by the symbol lookup program SLU. The same file is also processed through the Macro-11 assembler to produce an object file which defines the tables used by IOS to describe the hardware.

In actual practice there may be multiple hardware table files. For instance, one file could describe the configuration of the process I/O at a customer's site. A second file could describe a different physical configuration, with the same names, which represents the hardware on an in-house test system. By keeping two versions of the symbol definition files and the two corresponding versions of IOS the programmer can boot the same system disk on either of the two physical systems and declare which configuration is in use.

Programmed I/O

A large portion of the function library used by PCX tasks is taken up with I/O functions which call upon IOS. Before examining any of them in detail, let us look at the way in which tasks communicate function requests to IOS. All requests to IOS from other tasks, whether they are

applications tasks or system tasks, are defined by a formatted version of the 13 word message buffer. This buffer is defined by a structure called "piobuf" which has the following definition.

```
struct piobuf
     {
     int p_fcn;                    /* function code */
     int p_retc;                   /* return code */
     DEVID p_dev;
     int p_dat1, p_dat2, p_dat3, p_dat4, p_dat5;
     int p_res1, p_res2, p_res3;  /* reserved */
     };
```

For every request it receives, IOS returns a message to the requesting task when it has performed the function. This message contains a status code indicating the disposition of the request. Certain functions also return data in this packet. For instance a read of an input returns the state of the input.

This policy of returning a response for every request to IOS has two purposes. First of all it allows a task to verify that its request was successful. In practice, it is extremely rare for any task to check the return codes from IOS functions. Once an applications program has been tested one normally assumes that it will continue to function correctly. One of the few exceptions is a utility program which allows an operator to do S1500 I/O functions from a terminal. In this case the program checks all return codes to verify that the operator has applied a function to a compatible device type.

The second reason for the two way operation of IOS requests is that since some functions, by definition, must return a message it seems better to be consistent and force all functions to. This policy allows IOS to treat each request in a consistent manner and allows the functions in the library to share code which submits requests to IOS.

Since communication with IOS is through the normal send and receive data directives, some special handling is required in the interrupt processing system. All requests to IOS from a task are transmitted through a function called "submit". "Submit" sends the packet it is passed directly to IOS with the SDAT directive. It does not use the "send" function so as to avoid the superfluous request of IOS which that would entail. It then performs what is essentially a modified "rcvw" function, waiting for the return of a message which contains a function code the same as the request just sent out. All other messages received are assumed to be asynchronous messages from other tasks and are queued up in the hold list. When "submit" finds the desired message it returns to its caller. Note that "submit" could wait forever if IOS fails to respond. This is not considered a serious deficiency since the entire system relies on the proper operation of IOS. If IOS fails completely, there is very little that can be done to restore the system to operation, short of rebooting.

Interrupt Processing

For each module defined in the hardware table, IOS keeps a Unit Vector Block which is a table containing the name of the task which is to be notified of the interrupt from each point on the module. Each point may be vectored to only one task at a time, although it may have a null vector specified, in which case any interrupts from the device are ignored.

A task declares its desire to be notified of an interrupt from a device by "activating" the device with the "act" function. A device

must be reactivated each time it interrupts since its vector is cleared when the interrupt is dispatched to the receiving task.

If two tasks activate the same device, only the task name from the last request will be saved. The decision to handle this situation in this way was made in favor of simplicity. It would of course be possible for IOS to check for cases of a task stealing an interrupt from another task and rejecting the second request. Since that operation is often done intentionally, particularly when a piece of equipment is changing from one process to another, it was decided not to spend time on checks such as that.

IOS sends messages which use a format similar to the programmed I/O request format to user tasks to inform them of interrupts. These messages must be intercepted by the interrupt enqueuing functions in the user task and transformed from task message interrupts into S1500 device interrupts. This is done by the function "eqbuild" which builds event queue elements. When a task eventually receives the interrupt, the interrupting device id is the one specified in the message from IOS and the data buffer contains the complete contents of the message received from IOS.

Inventory System

The inventory system is responsible for maintaining information describing the contents of the tanks in the process environment. It is implemented by a task called INV. INV is closely associated with IOS which is responsible with supplying it with the input data received from turbine meters and counters. Certain installations may also use process

devices which read tank levels directly from sensors installed in the tasks. A separate task must be written to monitor special purpose devices such as tank gauges.

Inventory data is maintained in a memory partition which contains an array of with data structures called inventories. The structure of these elements is derived, with a few modifications, directly from PCS. Each inventory consists of a 32-bit accumulator and a set of data fields which contain information such as temperature, product code, and percent milk fat for the product in the associated tank. Each inventory may also have up to 4 limits. A limit is another 32-bit quantity which may be used to establish setpoints on the tank level. INV can be requested by a task to generate an interrupt when an inventory accumulator passes a setpoint.

The inventories in a system are defined at system generation time in a file called INVDEF.MAC. This file is processed to create a device id for each inventory. One field of the device id contains the offset of the accumulator within the partition so that INV can locate it. Device id's for other fields in the inventory can be created at run time by modifying another field.

PCLIB contains functions for manipulating the accumulator, limit, and data fields of the inventory. All of these functions are serviced by INV in the same way that IOS processes requests, since the applications tasks do not map to the inventory partition.

In a system which reads tank levels directly the most important functions are those which manipulate limits. The value of a limit can be set as well as its comparison type. Once the limit value has been

set, the exception interrupt for the limit can be enabled using the "act" function.

If there is no direct reading of tank levels an applications task must explicitly request that a meter or counter input be "attached" to an inventory accumulator. An input may attached to add or subtract. These functions require both IOS and INV. The meter and counter modules generate interrupts each time they measure a gallon or some multiple of a gallon. IOS informs INV each time it receives an interrupt from an attached input. INV in turn, adds or subtracts the amount counted from each of the inventories to which the input is attached.

Process Interlock Functions

Under PCS the software table provides a global common region where process status can be stored and is accessible to all other processes. This global accessibility allows tasks to perform what is known as interlock checking, looking for conflicts in process equipment usage, before starting a process. Some similar capabilities were required for the new system.

The message passing scheme for intertask communication is not adequate in itself to implement an interlock system. If it were used each task would be required to contain routines to process requests to return the current status of the process it controls. This is an overhead which seems unnecessary and the scheme seems prone to errors if each task has a unique system for maintaining status. Furthermore, this scheme does not define a secure place for each process to store its status. It is not adequate to store process status within the

controlling task since that task may exit while the process is still running, thereby destroying the information.

RSX provides facilities for static memory areas, independent of particular tasks. These are called shared regions. In particular, we are interested in static common regions which are created at system generation time and whose addresses are linked into a task when it is built. Any number of tasks may link to a static common. Each task may be built with different access rights to the region, either read only or read and write.

There are some drawbacks to placing the process status region into a static common and including it within the addressing space of each task. First of all, due to the limitations of the PDP-11 memory management hardware, each such region requires 4 KW of virtual address space in any task which maps to it. A large system may require 2 KW of status information, of which only a small fraction is used by any single task. Thus, the amount of useful space in each task is reduced considerably. Second, by giving each task immediate access to all process status, you greatly increase the chance of an incorrect task inadvertently destroying information which it should not even be using. Third, as was the case in PCS, there is no way to synchronize access by multiple tasks to such a region. This is due to the lack of hardware support in the PDP-11.

All three of these problems were overcome by placing process status in a static region but forcing all access to it to go through a single task called ILK. ILK defines the 4 KW interlock partition as an array of Process Status Blocks (PSB). It accepts request to read, read with

lock, and write with unlock for individual PSB's. These requests are serialized by the standard scheduling functions, thus ensuring orderly access to the process status data base by all tasks. This scheme does not prevent a task from randomly destroying process status but does greatly reduce the chance of it by requiring two calls, a read with lock followed by write, to change any data.

PSB's are assigned system-wide names in a file called PROCDEF.H. A PSB name is simply a number in the range of valid subscripts of the PSB array to which a symbolic name has been assigned by the C language preprocessor. Since the definitions are made using the compiler's preprocessor there is no way to check for multiple uses of the same name. Any task which will refer to PSB's by name must include this definition file when it is compiled.

There is a standard structure which describes a PSB, although there is no way to check for proper usage. A standard of usage was adopted which requires that if a PSB is to be used for storing of process information which is read by more that one process, certain fields in the structure must have specific significance. In particular, if the process state field has a value of zero, indicating that the process is not active, no process may assume that the other data in the PSB are valid.

The Menu System

A very important part of the design of the PCX system was the operator interface. This was true particularly in view of the fact that the primary means of communication was to be through standard terminals

rather than the pushbuttons and rotary switches used by PCS. There was therefore, no real base of past experience on which to fall back.

Several considerations influenced the design. First, a menu approach was desired because it is a popular style of command interface and should be familiar to the customer. It is also fairly easy to implement and it was hoped that it would be easily understood by the plant operators.

Second, whatever approach was used must be able to run on a standard DEC terminal, in particular some model from the VT100 line. This was primarily a maintenance consideration, since it was presumed that DEC service is easier to obtain in obscure locations. This restriction precludes any features such as "soft keys" which appear on the screen and are actuated by a special set of function keys which parallel them on the keyboard. All selections must be made with numeric keys. It also rules out such features as a touch screen.

A minimal system must be able to be run from a single terminal. This means responses to commands must be able to appear on the menu terminal and that a terminal cannot be dedicated to a single process since many processes may be active at any time. This has the effect that messages produced by a process may appear at a terminal while the operator is entering commands for some other process. Additionally, several messages may appear in quick succession, necessitating a fairly good sized buffer on the screen.

Screen Format

The menu system divides the 24 line screen into 4 sections. The top 11 lines are used for the menu display which is divided into two columns of 10 entries. The operator's selections are echoed in the upper right corner. This portion of the screen is controlled by a program called MNU which is evoked once for each menu terminal in the system. When it is initiated the menu program displays a root menu. All other menus in the system are arranged in a tree structure below this root.

The line below the menu area is reserved for process alarm messages. The messages are composed by the process tasks and sent to an alarm task which writes them to the screen. The alarm task is responsible for queueing up alarm requests and displaying them one at a time. An alarm message remains on the display until an acknowledgement is received from the operator.

The next line on the screen is used by the process tasks to display interlock messages. When the operator makes a menu selection which initiates some process the menu task sends a message to the appropriate process task. The first job of such a task is to check interlocks against other processes to see whether the selected process may be run. If some interlock failure is detected the process task prints an appropriate message on the interlock line of the terminal from which it was invoked. The menu task normally waits until this checking is complete before prompting for the next input. If this protocol is followed we can guarantee that no more than one task will attempt to print interlock messages at a particular terminal at any one time.

The remaining lines on the screen are reserved for other types of process messages. Messages are written to the bottom of the screen and scroll off at the interlock line. This is accomplished using the split screen capability of the VT100 terminals. With this function the screen can be divided into a scrolled and an unscrolled portion without any special software support.

Implementation

A significant problem was encountered in the implementation of the menu system. The design above essentially allows multiple tasks to use one terminal--one input task, MNU, and several output tasks. The structure of the RSX I/O system requires that a task, such as MNU, which wants to process unsolicited characters from a terminal, must "attach" to the terminal. Unfortunately, this prevents anyone else for writing to the terminal. Two rather unpleasant solutions presented themselves. A new terminal driver to be used for menu terminals could be written to replace the standard one. If a significant part of the functionality of the standard terminal driver was to be maintained this could have been a substantial project. A second approach was to expand the menu task to accept requests from user tasks to print messages so that all I/O to a terminal came through one task. This approach creates a significant amount of overhead in message passing and effectively doubles the demand on the operating systems dynamic memory which is notoriously scarce on RSX systems. Furthermore, it means that no task other than the menu task can ever solicit input from a menu terminal without adding functionality to the menu task.

The solution to this problem, when it arrived (from outside Tech-Con), was remarkably simple. Rather than being allocated only one serial line, each menu terminal was given two lines which are physically adjacent. A special cable with one connector at one end and two connectors at the other end is built for each terminal. All input from the terminal goes to one serial port and all output to it goes to the next higher numbered port. In this way the menu terminal can attach to the input port without interfering with any other task's output. The cost for the additional hardware is small in comparison to the programming effort which would have been required for a software solution and is probably a great deal more reliable in the short run.

Other Communications

A menu is often not adequate to handle all the data entry needs of a plant operator. For instance, when specific numeric data must be entered by the operator there is no way to do it through the menu. Therefore, the definition of a menu entry also includes the ability to specify that in addition to sending a message to a task, the menu task is to relinquish control of the menu terminal until it receives a reply from the user task. Once the menu task is no longer attached to the terminal, the user task is free to attach itself and establish another style of dialog with the operator.

When a task has gained control of a terminal it may use any of the standard RSX terminal driver functions. The only caveat is that the dual port arrangement is still in effect and so one must be careful in assigning logical unit numbers to physical terminal numbers.

PCX provides a number of functions to help a task deal with terminal I/O. In particular, the "attaio" function allows a task to receive individual characters from the terminal as interrupt data. The RSX terminal driver provides an asynchronous character AST service which presents a task with individual characters received from a task. The "attaio" function integrates these AST's into the PCX interrupt scheme.

V. CONCLUSIONS

In this chapter we will outline some of the problems which were discovered in PCX and fixed and some which still remain. We will also discuss some possible enhancements which could be made to the PCX system. In particular, we will examine another message based system and consider how some of its design principles could be applied to PCX.

Problems

The implementation of PCX was successful on the whole. Some blind alleys were followed in the original design and new approaches had to be found. However, several nagging problems remain. These are primarily due to basic inefficiencies in PCX and RSX-11. In some cases, simple changes can be devised which could improve the performance of PCX.

The original implementation of the PCX scheduler did not use the functions "rcv" and "act". A different approach was implemented first and abandoned. This approach used a pair of functions, "wait" and "enable", in place of "rcv" and "act". These functions were derived directly from certain of the "schedulers" written under PCS.

The principle behind the "wait" function is that at all times each task keeps an internal table of the device id's for which it is willing to process interrupts. This table is hidden from the programmer, who can make entries in it using the "enable" function. In other words, a task declares its willingness to process interrupts from a device by enabling that device. No call to the "act" function is required since that operation in invoked by "enable" if it is required. When the task

has enabled all the devices it needs, it can "wait" for the next interrupt.

This approach presents a problem in that it imposes an unnecessary overhead on the internal scheduling mechanism. When "wait" attempts to dispatch an interrupt it must scan the table of enabled devices to verify that the interrupt is currently valid. This is in many ways a superfluous operation since in all likelihood the user program must still perform some sort of validation on each interrupt to determine how to process it.

A second problem with "wait" and "enable" is that they do not allow a task to process random requests for service from other tasks unless the names of all tasks which will ever require service are known at compile time. This means, for instance, that a task such as the alarm task, which processes requests from all tasks to print alarm messages, cannot practically be written in terms of "wait" and "enable".

Third, and perhaps most bothersome, is that if a task exits, its current enable table is lost. Thus, when a task is initiated, it must enable all legal devices before it can process the interrupt which caused it to run. This condition is significant because it creates a motivation, real or imagined, for the programmer to write programs which do not exit, even when they have a significant length of time to wait. This leads to more tasks being active at a given time than are necessary, consuming resources and slowing the response of the system.

A peculiarity of the "enable" function is that it saves the device id pointer that is passed as an argument, rather than copying the device id itself. The "wait" function, in turn, returns the pointer when that

device interrupts. The result of the approach is that the event status block also contains a pointer to the interrupting device id. When the "rcv" function was developed, the same structure for the ESB was retained for the sake of compatibility. "Rcv" now has two arguments, a pointer to a buffer into which the interrupting device id is to be copied and a pointer to a buffer into which the ESB is to be copied. The device id pointer in the ESB is set to the value of the first argument passed to "rcv". If "rcv" had been created without the influence of its predecessor wait it would probably have been obvious that the ESB should contain the complete device id. In other words, it would look just like an event queue element which is placed in the hold list. The step of creating the event status block from the event queue element could have been eliminated. The concept of an interrupt being defined by a single data structure rather than a pair would also have been much clearer.

A number of the problems in the implementation of PCX seem to be ones of efficiency. Perhaps because so much of the design is based on message passing, that is where most of the problems seem to be. To some extent, the problems may be blamed on RSX which does not really appear to be designed with heavy use of the message functions in mind. The flavor of the documentation suggests that the primary means of intertask communication was intended to be via shared regions and the event flags.

One inefficiency which has already been mentioned is in the protocol for communication between user tasks and IOS. Each request to IOS requires that one message be transmitted in each direction. In practice, the reply from IOS is almost always ignored by the requestor.

Perhaps the wiser approach would have been to make the reply from IOS optional or to discard it altogether.

Another poor use of resources appears in the way received messages are handled by a task. When a task executes an "rcv" and there are no pending events in its hold list, it tries to read the next message from its receive data queue. If there are no messages, the task stops itself. When the next message is received, the receive data AST routine is activated, which unstops the task. The receive data AST routine could be completely eliminated from every task if the send function included an unstop of the task to which it sends a message. This approach is certainly less expensive than activating an AST routine for each message.

Some experiments can better illustrate some of the efficiency problems. The first test is for task A to send a message to task B and wait for a reply. This is repeated 1000 times. On an 11/24 processor with no other activity this requires one minute and six seconds. If we modify the send and receive functions slightly we can reduce the time for this sequence of operations to 54 seconds. The modification required is that we remove the receive data AST handler from the tasks and add an unstop request of the receiving task to the "send" function. The number of calls to the system is not reduced since the unstop was previously in the AST handler but we have saved the overhead of processing an AST in the receiving task.

A very dramatic result is found if we generate 1000 write requests to the null device driver. The write operation is very similar to a send and receive in that some form of message must be passed from the

requesting task to the driver and a reply returned which allows the task to continue. However, this program will complete all 1000 request in three seconds. The difference is that write uses the QIO mechanism instead of Send Data. The vast difference in performance is probably due to the fact that RSX preallocates I/O packets so that they are available for immediate use. There is no such preallocation of standard message packets so the system must search its dynamic memory free list before performing the operation.

It is certainly practical to make modifications to the send and receive functions to take advantage of these observations. We would almost certainly not see the full 18 percent gain in performance that is observed on an unloaded system but some improvement can be expected.

For reasons outlined in the discussion of the RSX I/O structure it is not practical to implement IOS as a device driver and reap the possible savings implicit in the use of the QIO system as opposed to the Send Data directive. The complexity of such an implementation precludes that. However, the observation of the potential improvement is worthwhile to the student since it points out the benefit of such a simple operation as preallocation of a commonly used resource such as message and I/O packets.

Enhancements

While "rcv" and its variations are adequate to implement most protocols between tasks, substantially more sophistication can be achieved. An example of a richer set of message passing primitives is

represented in the University of Rochester PLITS (Programming Language in the Sky) project.

PLITS was designed and implemented within the context of Rochester's Intelligent Gateway computer network. It defines processes called active modules whose only means of communication is via messages. The protocols and structures specified by PLITS are independent of any language but have been implemented in at least one case by augmenting Pascal.

PLITS messages are constructed from pairs of slot names and values. A message may be manipulated as a whole by a module which knows nothing of its internal structure. But any module which wants to inspect or change the values in a particular slot of a message must know the name of that slot. This is significantly different from a PCX message in which the structure is purely implicit. Under PLITS the structure of the message is a part of the message. Besides the obvious advantage of allowing message contents to be hidden from unauthorized modules this message structure also allows PLITS to provide primitives that can insert and remove slots from a message as well as test a message for the presence of a particular slot.

A PLITS message is sent in much the same way it is under PCX, by specifying a message and the name of the module to which it is to be sent. An additional feature is added to the send command whereby a transaction identifier can be appended to the message so that it can be identified in the future.

The reception of messages is again similar to PCX in the simplest case. The next message in the module's first in first out queue of

messages is returned by the receive function. Options in the receive allow either or both of the name of the sender and the value of a transaction key to be specified in much the same way that a device id may be specified in the PCX rcvw function.

The only radical difference that comparison of PLITS and PCX reveals is in the structure of messages. It is easy to see the value of more sophisticated message structures but implementing them under RSX could prove impractical. That added structure in a message carries quite a bit of overhead with it and we have already seen the relative inefficiency of RSX message passing.

At the present there is no universal structure to PCX messages. Each process defines the types of messages it is willing to process. The standard length of a message is 13 words but in practice we see that all 13 words are seldom used. If PCX were being reimplemented, it would be worth giving some thought to reserving some portion of a message for standard system-wide use. In particular, a transaction key field could be defined. This sort of field is very useful in client/server type arrangements where it is good to have some convenient means of attaching a unique identifier to a request so that it can be recognized when it returns.

LIST OF REFERENCES

LIST OF REFERENCES

Computer Concepts Corporation. PCS System Reference Manual. Knoxville, Tenn.: 1978.

Digital Equipment Corporation. RSX-11M/M-PLUS Executive Reference Manual. Maynard, Mass.: 1981.

Feldman, Jerome A. "High Level Programming for Distributed Computing." Communications of the ACM, 22 (1979), 353-367.

Kernighan, Brian W., and Dennis M. Ritchie. The C Programming Language. Englewood Cliffs, N.J.: Prentice Hall, 1978.

APPENDIXES

APPENDIX A

DEVICE IDENTIFIER STRUCTURE

A device id is described by the following structures:

```
typedef struct devid
     {
     COUNT bd_dev;                  /* device type */
     union
          {
          struct
               {
               COUNT bd_unt;        /* module number */
               unsigned bd_bit:4; /* bit number */
               unsigned bd_pte:4; /* pte number */
               unsigned bd_ofs:6; /* offset */
               unsigned bd_ifc:2; /* interface number */
               } bd_s15;
          struct                    /* timer id */
               {
               COUNT bd_tmn;        /* timer number */
               } bd_tm;
          struct                    /* inventory id */
               {
               COUNT bd_lim;        /* limit type */
               COUNT bd_addr;            /* address */
               } bd_iv;
          struct                    /* task id */
               {
               COUNT bd_tnl;        /* low order name */
               COUNT bd_tnh;        /* high order name */
               } bd_tsk;
          struct                    /* terminal */
               {
               COUNT bd_term;            /* terminal number */
               } bd_asynt;
          } bd_b;
     } DEVID;
```

A device id is three words long. The first word (member bd_dev) contains a one word device type. The currently defined device types are:

PTIN — Point input. The point input module provides eight 110V inputs referred to as points, each of which can generate an interrupt when it changes state. Each point may also be tested to determine its present state.

PTOUT — Point output. The point output provides eight 110V output points.

LAIN — Latching input. The latching input module has eight 110V inputs, each of which can generate an interrupt on a transition from low to high. A latching input can also

be attached to the inventory system so that input pulses to the module can be counted.

MTR — Meter. The meter module counts pulses received from two turbine meter inputs.

CTR — Counter. The counter module counts 110V pulses received on each of four input channels.

BCD — BCD I/O. BCD I/O modules are connected to a diverse group of physical devices such as operator control stations and proximity probe switches.

RS — Rotary switch. The rotary switch module allows a program to read the values of up to 16 sixteen position switches and to get an interrupt when one of them is turned.

RTD — Analog/RTD. The analog and RTD modules allow reading of temperature data from RTD temperature probes.

TASK — Task message. The TASK device type identifies messages from other tasks in the system.

INV — Inventory object. The inventory device type can be used, in combination with the other fields of the device id to describe any field in an inventory.

DLY — Timer interrupt.

ASYNT — Asynchronous terminal. An interrupt from a terminal signals the receipt of a single character from that terminal.

The union which describes the rest of the DEVID has five members, one for each of the major classes of devices: S1500 hardware, timers, inventories, asynchronous terminals, and tasks.

The members of the S1500 hardware descriptor are:

bd_unt — The unit number of the device. Unit numbers are assigned by the order in which modules are defined within a module.

bd_bit — This field identifies the bit number of the specific point input, latching input, and point output. It contains the control station address for BCD I/O. It contains the probe number for Analog/RTD modules. It may also contain the meter or counter number.

bd_pte — This field identifies the bit number within the specified

offset for BCD I/O PTE's and toggle switches. It identifies the specified 1554 module for an RTD.

bd_ofs This field identifies an offset within a particular BCD I/O remote.

bd_ifc This field contains the number of the interface to which the device is attached.

The members of the timer descriptor are:

bd_tmn Timer number. Up to 10 timers may be defined for each task.

The members of the inventory descriptor are:

bd_lim This field identifies the specified field within the inventory referenced by this DEVID.

bd_addr This field contains the offset of the start of this inventory into the inventory table.

The members of the task descriptor are:

bd_tnl First three characters of the task name in rad50.

bd_tnh Last three characters of the task name in rad50.

The terminal descriptor contains one member:

bd_term The physical terminal number.

APPENDIX B

PCLIB FUNCTION DESCRIPTIONS

NAME

act - activate an S1500 device

SYNOPSIS

```
act(dev)
     DEVID *dev;
```

FUNCTION

This function activates the interrupt of an S1500 device. The next interrupt from the specified device will cause the issuing task to be run and an interrupt data element to be sent to the task.

RETURNS

act returns the following values:

IS_SUC - successful completion
IB_DEV - invalid device id
IB_DTP - invalid device type to activate

SEE ALSO

deact, invact, invdact

NAME

aesbld - alternative event status block build

SYNOPSIS

```
aesbld(eqp, datap, dev)
     struct eque *eqp;
     ESB *datap;
     DEVID *dev;
```

FUNCTION

aesbld builds an event status element from an event queue element. The device id of the device which generated the event is copied to the area pointed to by dev. The device id pointer in the esb is set equal to dev. The data buffer in the eque is copied to the data area of the esb.

RETURNS

None.

REMARKS

This function is intended for internal use by the scheduler. It is used by the rcv and rcvex functions.

SEE ALSO

esbuild

NAME

delay - load a timer with a delay value.

SYNOPSIS

```
VOID delay(device, tmg, tnt)
     DEVID *device;
     COUNT tmg, tnt;
```

FUNCTION

delay loads the timer specified by devie with the time magnitude value tmg. The value is specified in units tnt. tnt may have the value TICK, TICKS, SECOND, SECONDS, MINUTE, MINUTES, HOUR, or HOURS.

If tnt = TICK, TICKS, SECOND, or, SECONDS, tmg can be any positive value with a maximum of 15 bits. If tnt = MINUTE or MINUTES, tmg can have a maximum value of 1440. If tnt = HOUR or HOURS, tmg can have a maximum value of 24.

RETURNS

Nothing. An error exit is taken if the device specified is not a valid timer or if an invalid time parameter is specified.

NAME

eqbuild - build an event queue element.

SYNOPSIS

```
struct eque eqbuild(wbuf)
     struct iosrpl *wbuf;
```

FUNCTION

eqbuild creates an event queue element from the event data buffer pointed to by wbuf. wbuf should point to a 15 word area such as is generated by a Receive Data directive.

RETURNS

A pointer to the event queue element created is returned.

REMARKS

This function is intended for internal use by the scheduler.

NAME

getpt - get state of a point type module

SYNOPSIS

```
COUNT getpt(dev, tab)
     DEVID *dev;
     unsigned tab[];
```

FUNCTION

getpt gets the current state of a point type module, either a 1542 Point Input or a 1543 Point Output. Values are returned in the five element array, tab. The element of tab contain the following values:

offset	1542	1543
0	presnt state register	presnt state register
1	bit changed register	auto/manual state register
2	past state register	manual state register
3	0	incorrect position register
4 (low byte)	0	energized feedback
(high byte)	0	unenergized feedback

RETURNS

The function returns the following values:

IS_SUC - successful completion
IB_DTP - illegal device type specified
IB_DEV - invalid device id specified

NAME

henque

SYNOPSIS

```
VOID henque(next)
     struct eque *next;
```

FUNCTION

henque inserts the event queue element pointed to by next at the end of the event hold queue.

RETURNS

None.

REMARKS

This function is intended for internal use by the scheduler.

NAME

_nextc - asynchronous character handler

SYNOPSIS

```
_nextc(code, r0, r1, c)
     COUNT code, r0, r1, c;
```

FUNCTION

_nextc is the AST handler for unsolicited characters received from terminals attached via the attaio function. For each character received it builds an event queue element containing the character and the device id of the terminal from which it was received.

RETURNS

None.

REMARKS

_nextc is not a user callable function.

NAME

on - turn on an output

SYNOPSIS

```
COUNT on(device)
     DEVID *device;
```

FUNCTION

on turns on the output specified by device. Only Point Outputs and BCD I/O LED's are supported by this function.

RETURNS

on returns the following values:

IS_SUC - successful completion
IB_DEV - invalid device id
IB_DTP - invalid device type

SEE ALSO

off, blink, deblink, alarm, dealarm

NAME

rcv - receive next interrupt

SYNOPSIS

```
rcv(dev, buf)
     DEVID *dev;
     ESB *buf;
```

FUNCTION

This function removes the next interrupt from the queue of pending interrupts and returns it to the caller. The device id of the device which generated the interrupt is placed in the area pointed to by dev. The data generated by the interrupt is copied to the area pointed to by buf. If there are no pending interrupts, rcv stops the task until one occurs.

RETURNS

None.

NAME

rcvex - receive next interrupt or exit

SYNOPSIS

```
DEVID *rcvex(dev, buf)
     DEVID *dev;
     ESB *buf;
```

FUNCTION

This function removes the next interrupt from the queue of pending interrupts and returns it to the caller. The device id of the device which generated the interrupt is placed in the area pointed to by dev. The data generated by the interrupt is copied to the area pointed to by buf. If there are no pending interrupts, rcvex exits.

RETURNS

If it returns, rcvex returns dev.

NAME

rcvoc - receive next interrupt or continue

SYNOPSIS

```
DEVID rcvoc(dev, buf)
     DEVID *dev;
     ESB *buf;
```

FUNCTION

rcvoc removes the next interrupt from the list of pending interrupts and returns its device id in the area pointed to by dev. The data generated by the event is placed in the event status block pointed to by buf. If no events are pending, rcvoc returns without changing dev or buf.

RETURNS

rcvoc returns dev if there was an interrupt pending. Otherwise, it returns NULL.

SEE ALSO

rcv, rcvex, rcvw

NAME

rcvw - receive next interrupt from a specified device or wait

SYNOPSIS

```
DEVID *rcvw(dev, buf)
     DEVID *dev;
     ESB *buf;
```

FUNCTION

rcvw searches the list of pending interrupts and returns the first one from the device specified by dev. If there are no such interrupts, the function waits stops the task until one occurs. The data generated by the the interrupt is returned in the ESB pointed to by buf.

RETURNS

rcvw returns dev.

SEE ALSO

rcv, rcvex, rcvoc

NAME

rdast - service receive data AST's

SYNOPSIS

```
VOID rdast(code, r0, r1, ef, ps, pc, dsw)
     int code, r0, r1, ef, ps, pc, dsw;
```

FUNCTION

This function processes all receive data AST's for a task. Its only function is to unstop the task. It does not dequeue the message being held by the system.

RETURNS

None.

REMARKS

This is not a user callable function.

NAME

s15init - look up the id of an S1500 hardware device or inventory

SYNOPSIS

```
BOOL s15init(device, name)
     DEVID *device;
     char *name;
```

FUNCTION

s15init loads the device id pointed to by device with a value denoting the S1500 device or inventory accumulator named by name. name is a null terminated string with maximum length of 6 characters.

RETURNS

s15init returns YES if the lookup was successful, NO otherwise.

SEE ALSO

timinit, tskinit, ivinit

NAME

send - send data to a task

SYNOPSIS

```
COUNT send(task, buf)
     DEVID *task;
     COUNT buf[13];
```

FUNCTION

send causes a 13 word block of data from buf to be sent to the task specified by task. The receiving task is "Requested" after the Send Data.

RETURNS

The dsw from the RQST operation is returned, unless the SDAT fails, in which case the dsw from that operation is returned.

NAME

submit - submit a request to another task.

SYNOPSIS

```
COUNT submit(req, tnl, tnh)
     struct eque *next;
     int tnl, tnh;
```

FUNCTION

submit sends a request to a task and waits for a reply. Any other messages which are received during the wait are queued up in the requesting task's hold list. The name of the task to which the request is to be made is passed in rad50 in tnl and tnh.

RETURNS

The value returned by submit is the contents of the p_retc field of the request after return from the called task.

REMARKS

This is not a general purpose function. It is intended for internal use in submitting requests to IOS and INV.

NAME

timinit - load a device id with a timer identifier

SYNOPSIS

```
BOOL timinit(device, timer)
     DEVID *device;
     COUNT timer;
```

FUNCTION

timinit loads the device id pointed to by device with the timer identifier for timer. timer can have an integer value from 1 to 10.

RETURNS

timinit returns YES if the supplied timer number was valid, NO otherwise.

NAME

tskinit - load a device id with a task identifier

SYNOPSIS

```
BOOL tskinit(device, name)
     DEVID *device;
     char *name;
```

FUNCTION

tskinit loads the device id pointed to by device with the task identifier of name. name is a null terminated string with a maximum length of 6 characters.

RETURNS

tskinit always returns YES.

APPENDIX C

PCLIB FUNCTION DESCRIPTIONS

```
/* act.c - activate an S1500 interrupt */

#include <std.h>
#include <rsxdir.h>
#include <sys.h>
#include "[200,7]ios.h"

COUNT act(dev)
    DEVID *dev;                                     /* device to activate */
    {
    struct iosrpl req;                              /* IOS request element */
    extern _tname[];                                /* this task's name */

    /* build request element */

    req.r_q.p_dat1 = _tname[0];                     /* insert task name in request element */
    req.r_q.p_dat2 = _tname[1];
    cpybuf(&req.r_q.p_dev, dev, sizeof(DEVID));     /* copy device id */

    if (dev->bd_dev == INV)                         /* request to inv. task ? */
        {
        req.r_q.p_fcn = IV_ACT;
        return(submit(&req, INVTSK, 0));
        }
    else                                            /* must be for IOS */
        {
        req.r_q.p_fcn = IO_ACT;
        return(submit(&req, IOS, 0));
        }
    }
```

```
/* aesbld.c - alt. esb build (used by rcv) */

#include <std.h>
#include <sys.h>

/* build an ESB from an event queue element */

aesbld(eqp, datap, dev)
    EQUE *eqp;                              /* event queue element */
    ESB *datap;                             /* caller's ESB */
    DEVID *dev;                             /* caller's device id */
    {
    if (datap)                              /* if data pointer is there */
        {
        cpybuf(datap->evdat, eqp->evd, 26);
        datap->esbdev = dev
        }
    if (dev != NULL)                        /* if pointer for interrupting device id is there */
    cpybuf(dev, &eqp->eqdev, sizeof(DEVID));

    free(eqp, sizeof(*eqp));                /* discard event queue element */
    }
```

```
/* delay.c - load a timer with a delay value */

#include <std.h>
#include <rsxdir.h>
#include <sys.h>
#include "[200,11]dly.h"

BITS dly_act = 0;                                   /* outstanding timer mask */
LOCAL AST(dast, 1, &dlyast);                        /* delay timeout ast */

VOID delay(device, tmg, tnt)
    FAST DEVID *device;                             /* timer id */
    COUNT tmg, tnt;                                 /* time magnitude and units */
    {
    deact(device);                                  /* kill any old delays on this timer */
    if (dly_act == 0)                               /* if no delays outstanding */
        emt(MRKT, WD_EFN, WD_PER, TICKS, &dast);    /* start watchdog timer */
    dly_act |= 1 << (device->bd_b.bd_tm.bd_tmn -TMNBASE - 1);       /* mark timer active */
    return(emt(MRKT, device->bd_b.bd_tm.bd_tmn, tmg, tnt, &dast);
    }
```

```
/* dlyast - ast service routine for mark times */

dlyast(code, r0, r1, mtef)
    int code, r0, r1;
    int mtef;                                       /* event flag from Mark Time */
    {
    IMPORT COUNT _tname[];  /* this task's name */
    FAST EQUE *next;

    if (mtef != WD_EFN)                             /* if not the watchdog timer */
        {
        next = alloc(sizeof(*next), NULL);          /* get an event queue element */

        /* build a device id for the completed timer */

        fill(next, sizeof(*next), '\0');
        next->eqdev.bd_dev = DLY;                   /* insert device type */
        next->eqdev.bd_tm.bd_tmn = staff;           /* and timer number */
        henque(next);                               /* place in hold list */
        }
    else
        emt(MRKT, WD_EFN, WD_PER, TICKS, &dast);    /* restart the watchdog */
    emt(USTP, _tname[0], _tname[1]);                /* wake up the task */
    }

wdcan(dev)          /* cancel watchdog timer */
    DEVID *dev;
    {
    if ((dly_act &= ~(1 << (dev->bd_b.bd_tm.bd_tmn - TMNBASE - 1))) == 0)
    emt(CMKT, WD_EFN, 0);
    }
```

```
/* eqbuild.c - build an event queue element from a receive data buffer */

#include <std.h>
#include <rsxdir.h>
#include <sys.h>
#include "[200,7]ios.h"

struct eque *eqbuild(wbuf)
    struct iosrpl *wbuf;                                    /* data generated by an event */
    {
    FAST struct eque *next;

    next = alloc(sizeof(*next), NULL);                      /* allocate an element */
    fill(next, sizeof(*next), '\0');                        /* and initialize it */

    /* Determine the type of the interrupt */

    if (wbuf->r_name[0] == IOS && wbuf->r_name[1] == 0)     /* S1500 I/O interrupt */
        cpybuf(&next->eqdev, &wbuf->r_q.p_dev, sizeof(DEVID));

    else if (wbuf->r_name[0] == INVTSK && wbuf->r_name[1] == 0) /* inventory limit */
        cpybuf(&next->eqdev, &wbuf->r_q.p_dev, sizeof(DEVID));

    else if ((wbuf->r_name[0] == rad50("DLY", 3)) &&        /* long term delay */
         (wbuf->r_name[1] == 0))
        timinit(&next->eqdev, (*int *)&wbuf->r_q) - TMNBASE);

    else                                                    /* another task */
        {
        next->eqdev.bd_dev == TASK;                         /* build DEVID */
        next->eqdev.bd_b.bd_tsk.bd_tnl = wbuf->r_name[0];
        next->eqdev.bd_b.bd_tsk.bd_tnh = wbuf->r_name[1];
        }
    cpybuf(next->evd, &wbuf->r_q, 26);                      /* copy interrupt data */
    return(next);
    }
```

```
/* getpt.c - point module get function */

#include <std.h>
#include <rsxdir.h>
#include <sys.h>
#include "[200,7]ios.h"

COUNT getpt(dev, tab)
    DEVID *dev;
    unsigned tab[];                                    /* return area for information read */
    {
    struct iosrpl req;
    COUNT ret;

    req.r_q.p_fcn = IO_GETPT;                          /* insert function code */
    cpybuf(&req.r_q.p_dev, dev, sizeof(DEVID));        /* copy device id */
    if (((req = submit(&req, IOS, 0)) == IS_SUC) && (tab != NULL)) /* ship it */
        cpybuf(tab, &req.r_q.p_dat1, 10);              /* copy result if request
                                                        * succeeded and user
                                                        * supplied table pointer */

    return(ret);
    }
```

```
/* henque.c - containing the functions henque and hdeque */

#include <std.h>
#include <rsxdir.h>
#include <sys.h>
#include "[200,7]ios.h"

/* henque - insert an event queue element at the tail of the holds list */

VOID henque(next)
    struct eque *next;                          /* element to insert */
    {
    IMPORT struct eque *holdr, *holdi;

    emt(DSAR);                                  /* disable AST recognition while fooling with queue */
    if (holdr == NULL)                          /* if hold list is empty */
    holdr = holdi = next;
    else
    {
    holdi->elink = next;                        /* point old last element to new */
    holdi = next;
    }
    next->elink = NULL;                         /* clear link in new element */
    emt(ENAR);                                  /* reenable AST's */
    }
```

```
/* hdeque - remove an event queue element from the head of the hold list */

struct eque *hdeque()
    {
    FAST EQUE *next;

    emt(DSAR);                              /* disable AST's */
    if (holdr != NULL)                      /* if list not empty */
    {
    next = holdr;                           /* point to front element */
    holdr = holdr->elink;                   /* cut it out */
    if (holdi == next)                      /* if it was the only one */
        holdi = NULL;                       /* clean out the list */
    if (next->eqdev.bd_dev == DLY)          /* if event is a delay */
        wdcan(&next->eqdev);                /* see if watchdog timer needs cancel */
    }
    else
    next = NULL;                            /* tell caller the sad news */
    emt(ENAR);                              /* allow AST's */
    return(next);
    }
```

```
/* on.c - turn on an output */

#include <std.h>
#include <rsxdir.h>
#include <sys.h>
#include "[200,7]ios.h"

COUNT on(dev)
    DEVID *dev;                     /* device to turn on */
    {
    struct iosrpl req;

    req.r_q.p_fcn = IO_ON;          /* set function code */
    devcpy(&req.r_q.p_dev, dev);    /* insert device specification */
    return(submit(&req, IOS, 0));   /* ship to IOS */
    }
```

```
/* rcv.c - receive next interrupt regardless of enabled status */

#include <std.h>
#include <rsxdir.h>
#include <sys.h>
#include "[200,7]ios.h"

rcv(dev, buf)
    DEVID *dev;                                 /* user's place to put interrupting device id */
    ESB *buf;                                   /* buffer for interrupt data */
    {
    struct iosrpl wbuf;                         /* Receive data buffer */
    struct eque *next;

    FOREVER
        {
        while (emt(RCVD, 0, 0, &wbuf) == IS_SUC)    /* while there are messages waiting */
            {
            next = eqbuild(&wbuf);              /* enqueue them */
            henque(next);
            }
        if ((next = hdeque()) != NULL)          /* try to remove event */
            {                                   /* from hold list */
            aesbld(next, buf, dev);             /* build event status block */
            return;
            }
        else
            {
            emt(ALTP, 0, 0, 40);                /* drop priority */
            if (emt (RCST, 0, 0, &wbuf) == IS_SUC)  /* receive or stop */
                {
                emt(ALTP, 0, 0, 0);             /* did not have to stop */
                next = eqbuild(&wbuf);          /* build event queue element */
                henque(next);                   /* stick in hold list */
                }
            else                                /* we were stopped */
                emt(ALTP, 0, 0, 0);             /* and no data was received */
            }
        }
    }
```

```
/* rcvex.c - receive next interrupt regardless of enabled status or exit */

#include <std.h>
#include <rsxdir.h>
#include <sys.h>
#include "[200,7]ios.h"

DEVID *rcvex(dev, buf)
    DEVID *dev;
    ESB *buf;
    {
    struct iosrpl wbuf;
    struct eque *next;

    FOREVER
        {
        while (emt(RCVD, 0, 0, &wbuf) == IS_SUC)     /* read all pending messages */
            {
            next = eqbuild(&wbuf);                   /* build event queue elements */
            henque(next);                            /* and put in hold list */
            }

        if ((next = hdeque()) != NULL)               /* try to remove an event */
            {
            aesbld(next, buf, dev);                  /* build an event status block */
            return(dev);                             /* and give to caller */
            }
        else
            {
            emt(ALTP, 0, 0, 40);                     /* drop priority during wait */
            emt(RCVX, 0, 0, &wbuf);                  /* Receive Data or Exit */
            emt(ALTP, 0, 0, 0);
            next = eqbuild(&wbuf);
            henque(next);
            }
        }
    }
```

```
/* rcvoc.c - receive next interrupt or continue */

#include <std.h>
#include <rsxdir.h>
#include <sys.h>
#include "[200,7]ios.h"

DEVID *rcvoc(dev, buf)
    DEVID *dev;
    ESB *buf;
    {
    struct iosrpl wbuf;                         /* Receive Data buffer */
    struct eque *next;

    while (emt(RCVD, 0, 0, &wbuf) == IS_SUC)    /* read all pending messages */
        {
        next = eqbuild(&wbuf);                  /* build event queue element */
        henque(next);                           /* and enque */
        }
    if ((next = hdeque()) != NULL)          /* try to remove first event from hold */
        {
        aesbld(next, buf, dev);                 /* convert to event status block */
        return(dev);                            /* pass back to caller */
        }
    return(NULL);                           /* nothing pending, continue */
    }
```

```
/* rcvw.c - receive next interrupt from specified device or wait */

#include <std.h>
#include <rsxdir.h>
#include <sys.h>
#include "[200,7] ios.h"

DEVID *rcvw(dev, buf)
    DEVID *dev;                             /* device id of desired interrupt */
    ESB *buf;
    {
    IMPORT struct eque *holdr, *holdi;
    struct iosrpl wbuf;
    FAST struct eque *next, *q;
    FAST COUNT dsw;

    FOREVER {
        while (emt(RCVD, 0, 0, &wbuf) == IS_SUC) {          /* read outstanding messages */
            next = eqbuild(&wbuf);                          /* build event queue element */
            henque(next);                                   /* put in hold list */
        }
        emt(DSAR);

        /* search hold list for caller's DEVID */

        for (next = &holdr; (q = next->elink) != NULL; next = next->elink)
            if (cmpbuf(dev, &q->eqdev, sizeof(DEVID))) {
                next->elink = q->elink;                     /* cut it out of list */
                if (holdi = q)
                    holdi = ((next == &holdr) ? NULL : next);
                emt(ENAR);                                  /* AST'S may continue */
                return(esbuild(q, buf, dev));               /* ship interrupt to caller */
            }

        /* No such device in hold list, wait for next interrupt */
        emt(ENAR);
        emt(ALTP, 0, 0, 40);                                /* lower priority for waiting */
        dsw = emt(RCST, 0, 0, &wbuf);                       /* Receive Data or Stop */
        emt(ALTP, 0, 0, 0);                                 /* restore priority */
        if (dsw == IS_SUC) {                                /* if read was successful */
            next = eqbuild(&wbuf);                          /* enque the interrupt */
            henque(next);
        }
    }
}
```

```
/* rdast.c - service read data AST'S */

#include <std.h>
#include <rsxdir.h>
#include <sys.h>

rdast(code, r0, r1, ef, ps, pc, dsw)
    int code, r0, r1, ef, ps, pc, dsw;
    {
    IMPORT COUNT _tname[];

    emt(USTP, _tname[0], _tname[1]);
    }
```

```
/* nextc.c - process next asynchronous input character */

#include <rsxdir.h>
#include <sys.h>
#include "asynt.h"

AST(_ucast, 1, _nextc);                         /* define AST service routine */

_nextc(code, r0, r1, c)
    COUNT code, r0, r1                          /* useless system stuff */
    COUNT c;                                    /* character and lun */
    {
    IMPORT AT_ELT *_atlst;                      /* list of attached terminals */
    IMPORT UCOUNT _tname[];                     /* current task name */
    AT_ELT *q;
    EQUE *eque;
    union {                                     /* esb for event */
        AS_DAT at_esb;
        COUNT d2[13];
    } uesb;

    struct {                                    /* redefinition of arg. c */
        UTINY c_c;                              /* character received */
        UTINY c_lun;                            /* logical unit number */
    } *cp;

    cp = &c;
    for (q = _atlist; q; q = q->at_link)            /* scan list of attached terminals */
        if (q->at_ilun == cp->c_lun)                /* this is our lun ? */
            {
            c &= c & 0377;                          /* clear out lun information */
            if (c == '\001')                        /* control-A ? */
                lack(q->at_ilun);                   /* ship an alarm acknowledge */
            else
                {
                uesb.at_esb.as_term = q->at_term;   /* get terminal number */
                uesb.at_esb.as_char = c;            /* save character received */
                eque = alloc(sizeof(EQUE), NULL);   /* allocate event queue element */
                fill(&eque->eqdev, sizeof(DEVID), 0);
                eque->eqdev
                }
            }
    }
```

```
/* s15init.c - look up device id of S1500 hardware device or inventory */

#include <std.h>
#include <rsxdir.h>
#include <sys.h>
#include "[200,7]ios.h"

BOOL s15init(dev, name)
    FAST DEVID *dev;
    FAST TEXT *name;
    {
    FAST COUNT i, dsw;
    struct iosrpl buf;
    TEXT *tp;

    if (dev == NULL)                                /* must have a place to put result */
    return(NO);

    if ((name == NULL) || (lenstr(name) == 0))      /* and a name for the device */
        {
        fill(dev, sizeof(DEVID), 0);                /* fill with 0 for null name */
        return(NO);
        }

    /* copy name of device (up to 6 characters) to request.  Pad with blanks */

    for (tp = cpystr(&buf.r_q.p_dat1, name, NULL); tp < (((TEXT *)&buf.r_q.p_dat1) + 6); tp++)
        *tp = ' ';

    buf.r_q.p_fcn = 1;                              /* function = lookup by name */
    if (submit(&buf, DOTS, SLU))                    /* if lookup succeeds */
        {
        cpybuf(dev, &buf.r_q.p_dev, sizeof(DEVID)); /* copy the device id to caller */
        return(YES);
        }
    else
        {
        fill(dev, sizeof(DEVID), '\0');             /* return a 0 device id */
        return(NO);
        }
    }
```

```
/* send.c - send data to a task */

#include <std.h>
#include <rsxdir.h>
#include <sys.h>

COUNT send(t, buf)
    DEVID *t;                                   /* receiving task */
    COUNT *buf;                                 /* message */
    {
    COUNT dsw, zbuff[13];

    if (t->bd_dev == TASK)                      /* check for tast type DEVID */
        {
        if (buf != NULL)                        /* if message not null */
            dsw = emt(SDAT, t->bd_b.bd_tsk.bd_tnl, t->bd_b.bd_tsk.bd_tnh, buf, 0);
        else
            {
            fill(zbuff, 26, 0);                 /* build dummy message */
            dsw = emt(SDAT, t->bd_b.bd_tsk.bd_tnl, t->bd_b.bd_tsk.bd_tnh, zbuff, 0);
            }
        if (dsw == IS_SUC)                      /* if send succeeded, request the task */
            dsw = emt(RQST, t->bd_b.bd_tsk.bd_tnl, t->bd_b.bd_tsk.bd_tnh, 0, 0, 0, 0);
        return(dsw);
        }
    else
        return(IB_DTP);                         /* device was not a task */
    }
```

```
/* submit.c - submit request to a task and wait for reply */

#include <std.h>
#include <rsxdir.h>
#include <sys.h>
#include "[200,7]ios.h"

COUNT submit(req, lnam, hnam)
    FAST struct iosrpl *req;                                /* request element */
    int lnam, hnam;                                         /* receiving task's name */
    {
    COUNT dsw, fcn;
    FAST struct eque *next;

    fcn = req->r_q.p_fcn;                                   /* remember function code */

    if ((dsw = emt(SDAT, lnam, hnam, &req->r_q, 0)) != IS_SUC) /* send request */
        return(dsw);                                        /* return reason for failure */

    FOREVER
        {
        emt(ALTP, 0, 0, 40);                                /* drop priority while waiting */
        while ((dsw = emt(RCST, 0, 0, req)) == IS_SUC)      /* try to receive reply */
            ;
        emt(ALTP, 0, 0, 0);                                 /* restore priority */
        if (dsw == IS_SUC)                                  /* if receive succeeds */

            /* verify that this is our reply */

            if ((req->r_name[0] == lnam) && (req->r_name[1] == hnam)
                                         && (fcn == req->r_q.p_fcn))
                return(req->r_q.p_retc);                    /* return to caller with status */
            else
                {                                           /* was not reply, but some other message */
                next = eqbuild(req);                        /* build event queue elt. */
                henque(next);                               /* and shove in hold list */
                }
        }
    }
```

```
/* timinit.c - load a device id with a timer identifier */

#include <std.h>
#include <sys.h>

BOOL timinit(dev, tmr)
    DEVID *dev;
    int tmr;
    {
    fill(dev, sizeof(DEVID), 0);                    /* initialize to 0 */
    if (tmr > 0 && tmr <= 16)                       /* check valid timer number */
        {
        dev->bd_dev = DLY;                          /* device type is DLY */
        dev->bd_tm.bd_tmn = tmr + TMNBASE;          /* convert timer number to event flag */
        return(YES);
        }
    return(NO);
    }
```

```
/* tskinit.c - load a device id with a task identifier */

#include <std.h>
#include <sys.h>

BOOL tskinit(dev, name)
    DEVID *dev;
    char name[];
    {
    int l;

    l = lenstr(name);                                          /* get length of name */
    if (l > 0 && l <= 6)
        {
        dev->bd_dev = TASK;                                    /* device type is task */
        dev->bd_b.bd_tsk.bd_tnl = rad50(name, min(3, l));      /* convert 1st 3 digits */
        dev->bd_b.bd_tsk.bd_tnh = rad50(name + 3, min(3, l - 3)); /* and next 3 */
        return(YES);
        }
    else
        {
        fill(dev, sizeof(DEVID), 0);                           /* fill with 0 if bad name */
        return(NO);
        }
    }
```

VITA

Peter Volkmar Andreae was born in Chattanooga, Tennessee on June 22, 1955. He attended elementary schools in that city and was graduated from Chattanooga High School in June 1973. The following September he entered the University of Tennessee, Knoxville, and in June 1977 he received a Bachelor of Arts degree in Computer Science.

Since completing his undergraduate studies the author has been employed by Computer Concepts Corporation and Tech-Con Inc., both of which supply computer control systems to the food industry. In September of 1981 he entered the Graduate School of the University of Tennessee, Knoxville. He received the Master of Science degree with a major in Computer Science in June 1984.

The author is a member of the Association for Computing Machinery and the Computer Society of the Insitute for Electrical and Electronic Engineers.