T H E   U N I V E R S I T Y   O F   M I C H I G A N

Technical Report 19

Multiprogramming in a Small-Systems Environment

David L. Mills

CONCOMP: Research in Conversational Use of Computers
F.H. Westervelt, Project Director
ORA Project 07449

supported by:

ADVANCED RESEARCH PROJECTS AGENCY
DEPARTMENT OF DEFENSE
WASHINGTON, D.C.

CONTRACT NO. DA-49-083 OSA-3050
ARPA ORDER NO. 716

administered through:

OFFICE OF RESEARCH ADMINISTRATION   ANN ARBOR

May 1969

Multiprogramming in a Small-Systems Environment

ABSTRACT

This report discusses multiprogramming systems architectures suitable for use with small machines of the PDP-8 class. Techniques for task and I/O device scheduling, storage and device allocation, buffer and timer management and command language interpretation are discussed in detail. Illustrative implementation details are freely drawn from a follow-on version of RAMP, a multiprogramming system now used in several applications involving process control, message switching and terminal control.

This report was prepared using FORMAT, a computer program in MTS, the Michigan Terminal System. This program is described in: Berns, G.M., _Description of FORMAT, a Text-Processing Program,_ Comm. ACM, 12, 3 (March 1969), pp. 141-146. The text was entered to this program partly in punched-card form and partly directly from a typewriter terminal and was printed on an IBM 1403 printer equipped with a TN print train.

Multiprogramming in a Small-Systems Environment

TABLE OF CONTENTS

1. INTRODUCTION

This report describes further developments of the RAMP system, a multiprogramming system designed for use in small machines of the PDP-8 class for operation in real-time processing environments. The old RAMP systems, described in References 1-4, have been used extensively in process control, message switching and terminal control environments. The development of the new system, called here simply "new RAMP," has resulted in a greatly enhanced throughput and a much more attractive interface to which special-purpose job-program subsystems can be attached. In addition, the new system incorporates a general storage allocation technique which provides both for the dynamic loading of page-relocatable job-program subsystems and for the temporary storage of control information and I/O buffers. The concept of task, program, and real/task-time processing is given more precise definition in the new system; and the interface to device service routines is made much simpler and more powerful. Finally, I/O message transmissions are accomplished in a record-oriented rather than a byte-oriented fashion, so that high-speed I/O devices can be efficiently connected and so that program loading can proceed directly to core memory rather than through an intermediate buffer.

Although the primary implementation goal in the new RAMP system was increased throughput and flexibility as compared to the old system, the development of the new system has been different enough from the old that it is presented here _in toto;_ and familiarity with the old system is not assumed requisite in the exposition. Furthermore, although both the new and the old systems were coded for the PDP-8 family of machines, the implementation techniques described here are obviously not restricted for use on this machine, but are appropriate for use on any "small" machine requiring I/O interrupts on a character-by-character basis. In fact, it will develop that many of the construction techniques are really special adaptations of those used in larger systems.

2. OPERATING ENVIRONMENT

Although the emphasis in this report is on general systems architecture and implementation techniques, the prinicipal interest is in multiprogramming systems for relatively small systems such as those intended for process control, message switching and terminal control applications. In general the hardware architecture of the machine itself is assumed given; and no special equipment is postulated especially for its use in a multiprogramming environment. The goal of the presentation here is then a specification of attractive and useful system organizations which allow a general-purpose machine of very modest capabilities to operate efficiently in a multiprogramming environment.

Throughout the subsequent discussion a hypothetical machine will be assumed whose characteristics are typical of a number of small general-purpose machines currently on the market. Obviously, some of the characteristics assumed are influenced by certainly the most popular of these machines, the PDP-8; and indeed, the particular implementation most often referenced involves this machine. However, the implementation techniques discussed are obviously not restricted to systems involving this particular machine.

The hypothetical machine considered here has a basic core memory bank of perhaps 4K twelve- to sixteen-bit words. The supervisory system is expected to occupy perhaps a quarter to a half of this. The various job-program subsystems, designed to perform the specialized processing unique to each application, occupy the remainder of the basic memory bank and perhaps additional add-on memory banks as well. Systems now in use involve PDP-8 systems of from one to four 4K-word memory banks.

The basic machine is assumed to include a rather fast central processing unit (CPU) of a limited instruction repetoire. As will be illustrated below, a slower CPU processing speed can be exchanged for a more sophisticated I/O structure, depending of course on the application. An accumulator (AC) program counter (PC) and the various memory-bus registers are assumed integral to the CPU of course; but extended arithmetic capability (high-speed multiply/divide) and index registers are not necessarily assumed available. An indirect-addressing capability is requisite, as well as the logical operations of "and" and "one's complement" and the arithmetic operations of "addition" and "two's complement", or any equivalent set. A minimal set of conditional-skip instructions, a branch instruction and subroutine-link instruction are necessary of course. These assumptions define a machine which can be argued is the smallest useful machine larger than a Turing machine. Table 1 illustrates a comparison of a few currently-available machines in this class.

	Word Size	Mem Size	Mem Cyc	Inst Cyc	Work Regs	Page Size	Comments
Digital Equip.							
PDP-8S	12	4096	8	42	1	128	serial
PDP-8	12	4096	1.5	1.5	1	128	sup. PDP-8I
PDP-7	18	4096	1.75	3.5	1	8192	sup. PDP-9
PDP-9	18	8192	1.5	1.5	1	8192	
PDP-15	18	4096	.8	-	2	4096	1 index
Interdata							
Model 3	16	1024	2	-	16	32768	
Data General							
NOVA	16	1024	6.5	5.5	4	256	2 index
SEL							
810	16	4096	1.75	-	-		
Data Machines							
620i	16	1024	1.8	3.6	3	2048	micropgm
SDS							
Sigma 2	16	4096	-	-	8	256	

Table 1. Comparison of Small Machines

The typical application of small multiprogramming systems envisioned here includes a good deal of I/O activity; and indeed, the only significant activity performed by the system may be of this nature. Accordingly, the specification of the I/O interface of the hypothetical machine is crucial but of necessity highly tailored to each individual application. A typical specification will be assumed here; more detailed descriptions of actual systems can be found in the references at the end of this report.

The hypothetical machine is assumed to contain facilities for a program interrupt at a single priority level. That is, if the interrupt system of the machine is enabled, then a simulated subroutine branch will be taken to a fixed location in memory when an interrupt condition becomes pending. A set of instructions is assumed available to test each I/O device status and to initiate operations on these devices. The bulk of all I/O data transfers is assumed to occur on a character-by-character basis, with an interrupt occuring for each character transferred. In some equipment data transfers can be performed on a cycle-steal basis; that is, directly to and from memory and an I/O device. A richer I/O hardware architecture, possibly including circuitry to resolve the priority and identifying code of the various interruption conditions, could very easily be added to the hypothetical machine, but our main interest here is in the "barest-bones" architecture.

3. BASIC CONCEPTS AND DEFINITIONS

A _program_ is simply a collection of subroutines which operate upon a set of data. Our interest here is in a system which includes several such programs, each of which may be executed in some sense independently of the others, but only one of which is in actual control of the machine at any particular time. One of these programs, the _supervisor_, is made responsible for the coordination of the various programs in the system and the operation of the scheduling algorithm which determines which of these will next be given control of the machine. In general the supervisor is made responsible for the coordination and the allocation of all of the various system resources such as processing facilities, core storage and I/O devices.

A _task_ is a program together with I/O device assignments and storage allocations which constitute a record of its physical state. In most RAMP systems the number and configuration of the programs are usually fixed, although some of the systems can sustain dynamic loading operations. However, in all systems, the I/O device assignments and storage allocations can vary dynamically during operation, so that the existence of a task can be considered synonymous with the existence of these assignments.

As control is passed among the various tasks in the system, the state of each task is preserved in a special block of storage called the _task control block_ (TCB) (see Figure 1). Information stored in the TCB includes at least the instruction address at which the task was last suspended, and presumably is the address at which the task will be restarted when it is once again given control of the machine. Depending upon the architecture of the particular system involved, other information stored in the TCB may include certain machine registers, I/O device assignments and storage allocation assignments.

Obviously, the association of a TCB to a program and the data it operates upon is highly arbitrary and can be maintained by pointers stored in the TCB itself. Thus there is considerable justification, when describing multiprogramming systems, to subordinate the usual concepts of program and data to that of the task and its TCB, which reflects the current state of the processor as seen by the program.

As each task is given control by the supervisor, the machine state peculiar to that task is restored using the data stored in the TCB as of the last time that task was suspended, and processing continues in the usual manner.

When processing of this task is temporarily suspended for some reason, possibly pending completion of a time-dependent operation, then the state of the machine is stored in the TCB and processing continues with the next task awaiting activation. Rescheduling of the suspended task is a function of the system architecture and depends upon whether the suspension was initiated upon request by the task or as the result of a hardware interrupt.

```
+-----------------------+
|   LINK TO NEXT TCB    |
+-----------------------+
|  LINK TO CALLING TCB  |
+-----------------------+
|  INSTRUCTION ADDRESS  |
+-----------------------+
|                       |
+ DEVICE TABLE POINTERS +
|                       |
+-----------------------+
|                       |
+      PARAMETERS       +
|                       |
+-----------------------+
```

Figure 1. Task Control Block (TCB) Format

The usual RAMP task-scheduling procedure consists of arranging the TCB's of the system in a first-in-first-out queue and chaining each TCB to the next in this queue. The task-scheduling process then considers each TCB in turn so that, if the task in control of the machine at a particular moment is considered to have highest priority, then, when it is suspended, it can be considered to have lowest priority. Many other refinements are possible to this scheme, some of which are described in the next section.

Multiprogramming systems have traditionally been categorized as true multiprogramming systems where processing of a given task may continue indefinitely or until explicitly indicated by a call on the supervisor, perhaps as a request for some I/O operation, and as timesharing systems, where processing of a given task continues until explicitly indicated or until a preset time-slice interval has expired. Once the time-slice interval has expired, the task is suspended and other tasks are processed as required until the task in question is again re-activated by the supervisor, at which time it is again given its preset time slice. As can be seen here, apart from the details of task scheduling and the mechanism of timer interrupts, there is no fundamental difference between true multiprogramming systems and timesharing systems and no artificial distinctions will be drawn between them henceforth.

In summary, in these introductory sections, the gross organization of the architecture of RAMP-like multiprogramming systems should be apparent. In subsequent sections various implementation techniques will be discussed and examples presented. It should be emphasized, however, that the exposition here does not attempt to describe detailed specifications or calling sequences of the various systems, but rather to describe only the architectural characteristics and primary functional components.

3. Basic Concepts and Definitions

4. BASIC SYSTEM ARCHITECTURE

In typical multiprogramming system operation an active task runs until an interrupt occurs or until a time-dependent supervisor operation is requested by the task. In either case the supervisor is required to save the active registers and machine status in the TCB for that task and to schedule the next task to be activated. In the most general case involving timesharing operations there may be several queues maintained by the supervisor for the purpose of scheduling access to system resources such as I/O devices, bulk memory and processing time; and the algorithm for task-scheduling following an interrupt may be exceedingly complex. In particular, the next task to be executed following an I/O interrupt may not be the task in which the interrupt occured.

In most large-scale multiprogramming and timesharing systems the programs themselves are constructed so as not to modify themselves during execution. In such systems several tasks may share the same program and store the temporary information unique to each task in the TCB for that task. In a small system involving a machine with no index register this type of operation may be exceedingly awkward, since many of these machines typically store the return link for a subroutine in the subroutine code itself. Accordingly, a modified view of the typical multiprogramming system is needed in which a particular task cannot be suspended until all of its state-dependent return links and index pointers have been saved in the TCB.

In general, the philosophy inherent in the various RAMP systems has been to require the programming functions of each task to be responsible for these operations and to indicate to the supervisor precisely those points at which a task-scheduling operation can occur. As an implication of these architectural requirements, the I/O interrupt processor is required to explicitly schedule on the CPU queue those tasks which have become suspended awaiting the completion of some time-dependent operation. Only when the task in which the interrupt was taken becomes suspended as the result of an explicit supervisor operation will the scheduled task become eligible for re-activation.

This construction leads to a bimodal classification of processing as <u>real-time</u> in which only interrupts are serviced and tasks may be scheduled, and <u>task-time</u> in which normal task operations can occur. The real-time processors are assigned a reserved set of temporary storage locations and are designed to be activated by a hardware interrupt or subroutine call (with the interrupt system masked off) and to return directly to the calling task-time routine. The task-time processors share a reserved set of temporary storage locations disjoint from the real-time set, of course, and are designed to be activated by and return to the supervisor.

In a RAMP system machine control is passed among the various TCB's in the CPU queue on a first-come-first-served basis. A particular task, once given control, is said to be _active_ and will run until some time-dependent supervisor operation is requested, the completion of which presumably is predicated before execution can continue. At the time such an operation is requested, its TCB is placed on a special queue depending upon the type of operation requested. At this point the next task on the CPU queue is activated and control is passed to that task. Operation continues in this manner until some task is interrupted to signal the completion of the operation. At this time control is passed to the real-time processor responsible; and, following its completion, the interrupted task is restarted from the point of interruption. The interrupt-processing subroutine invoked in such a manner has the responsibility of removing the TCB from the special queue and _scheduling_ it on the CPU queue. The TCB is now said to be in the _pending_ state. When the pending task again becomes active, task-time processing of the I/O operation can continue.

Now an obvious refinement to this simple system model would allow certain subroutines to be shared among the various tasks. Such subroutines can contain references to parameters and temporaries residing in permanently-allocated scratch storage shared by all tasks in the system. If a task can be suspended in the sense of the preceding paragraph, then it may happen that some of these parameters and temporaries would have to be saved during the suspension interval while, presumably, other tasks may make use of them. In a RAMP system these parameters and temporaries can be saved in the TCB itself; but it is the program's responsibility to determine which to save in each case and to perform the actual copying operations. Certain system conventions and utility subroutines streamline this process, however, and in such a way as to provide an inherently recursive construction.

A RAMP routine can be classified by the way in which it can be shared by the various task invocations. At each point in the code a certain state of the invoking TCB, common temporary storage and machine registers is assumed. If this _task state_ is independent of the state of the real-time system and the common scratch storage allocated to the real-time system, then the routine is said to be _interruptable_ at that point, and presumably the hardware can be unmasked for real-time interrupts. A routine operates in the task-time state if it is interruptable at some point and in the real-time state if it is not. If the task state is independent of a prior use of the routine by this or another task, the the routine is designated _serially-reusable_ at that point. If the task state is independent over a suspension interval during which the routine can be shared by another task, then the routine is designated _re-entrable_ at that point.

A task-time routine will never alter real-time temporaries or call real-time routines without first masking off the interrupt system. If a task-time routine makes use of shared scratch storage to save information over a suspension interval, it is by necessity not re-entrable during that interval and cannot be entered by any other task. All temporary storage used by a re-entrable task must reside in the TCB for that task or in storage allocated by and belonging to that particular task invocation. Serially-reusable routines typically make use of large amounts of temporary storage which are impractical to replicate for each invocation separately. Routines that cannot obey even the serially-reusable criterion are only useful after a fresh program load, of course, and are not usually part of the multiprogramming system at all.

Although, in the current implementations, the basic RAMP supervisory system contains only re-entrable routines, a practical job program can contain a collection of both serially-reusable and re-entrable routines, depending upon frequency of use, temporary storage requirements and other such factors. A mechanism must be provided for each serially-reusable routine to avoid entry by one task when it is being used by another. Mechanisms for this are described in the references listed at the end of this report.

During normal system operation tasks are created, executed and destroyed routinely and often. In order to economize on the storage used by the various tasks, the TCB and parameter regions are dynamically allocated and released as each task is created and destroyed. A task is created by an explicit subroutine call, presumably on the part of another invoking task. The invoking task is here called the _mother_ task, and each of her invoked tasks are called her _daughters._ As implied, a mother task can create any number of daughter tasks whose operations then proceed in parallel with those of the mother task. Optionally, however, the mother task can initiate an explicit operation in such a manner that, once a daughter task has been created, the execution of the mother task is suspended and is restarted only when the daughter task has completed all processing. Such an operation can be specified for only one daughter task at a time. These operations are also possible in other systems recently described.

In summary, and before proceeding with a detailed description of the various system operations and the construction of the various queues, the configuration of a RAMP system at any time during operation can be viewed as a collection of TCB's one of which is in active control of the CPU. TCB's are scheduled on the various queues as the result of real-time interrupt processing or as a result of explicit supervisor operations. Routines can be shared between the various tasks in a limited fashion depending upon their use of common routines and scratch storage. Finally, and most importantly, a real-time interrupt operation is constrained to restart the interrupted task at the point of interruption and no task-switching operation is allowed as a result.

4.1 Task-Time Operations

In the absence of time-dependent events, execution of tasks in a RAMP system is on a first-come-first-served basis, with control being passed to each task in turn one after the other. TCB's for tasks awaiting execution are held on the CPU queue and these are described as being in the pending state. When a task-switching operation occurs, the next TCB on the CPU queue is removed from the queue and execution commences at the instruction address indicated by an entry in the TCB (see Figure 1). A task in this condition is described as being in the active state. It is possible under certain conditions when an asynchronous attention is pending that a task be in both of these states at the same time, that is pending on the CPU queue and active in execution. Once a task becomes active its execution continues until one of four events occurs:

1. The task terminates operations and returns control to either its mother task or to the system. The former behavior is possible only if the mother task in question has initiated a WAIT operation (see below) for the active daughter task. The latter behavior occurs in all other cases. In any case, the task termination procedure causes the TCB storage claimed by the active task to be released.

2. The task creates a daughter task using the INSERT operation followed by a WAIT operation. This causes the execution of the mother task to be suspended pending completion of its daughter task activity. During the waiting interval the mother task is said to be in the dormant state. It is

possible under certain circumstances that a task be in both the pending and the dormant states or in the active and dormant states at the same time. These situations can occur only when an asynchronous attention is pending.

3. The task encounters a temporary busy condition which is expected to last a long time compared to the instruction processing time but a short time compared to the CPU queue processing time. In such a situation a special BUSY or REQUE operation (see below) can be initiated, which results in the TCB being placed at the end of the CPU queue. When this TCB next becomes active due to normal task processing, the task is resumed at the current instruction address specified.

4. The task initiates an I/O operation on a device. In such a situation the TCB is placed on the appropriate _I/O queue_ (see below) associated with the device. A task in this condition is described as being in the _blocked_ state. Presumably an I/O interrupt will occur at some future time and the TCB will again be scheduled on the CPU queue.

In the latter three of these four cases, once the active task has been placed in the dormant, pending, or blocked state, the next task pending on the CPU queue is made active and processing continues in that task. In the first case, where the daughter task returns to its mother task, execution continues in the mother task without interruption due to a task-switching operation, and in particular without placing the TCB of the mother task on the CPU queue. Where no mother task is waiting for the active task in question, the active task disappears entirely and the next task pending on the CPU queue is made active.

Figure 2 illustrates a typical configuration of a RAMP system in which one task is in execution (active), some tasks are awaiting execution (pending), one is waiting for its daughter task to complete processing (dormant), and some are waiting for the completion of an I/O operation (blocked). Referring to Figure 1, note that the first word in each TCB is always used as a pointer to the next TCB on some queue or other and that the second word is always either zero or a pointer to the TCB of a mother task. The third word is used as the instruction address at which execution will re-commence when the task is again made active. With this organization note that a TCB may be claimed in only one queue at a time and that it may be waiting for only one daughter task at a time. Furthermore, no matter what the condition is that causes the task to become active, execution can begin only at the current instruction address. Finally, the only ways a task can become active are either through the normal CPU queue processing or as a result of a return from a daughter task. The fact that these two operations can be independent of each other will be important when the asynchronous attention operation is discussed below.

```
         +-----+          +-----+          +-----+
CPU Q -->|  X--+--------->|  X--+--------->|  0  |
         |  0  |          |  X--+----+     |  0  |
         |     |          |     |    |     |     |
         +-----+          +-----+    |     +-----+
         PENDING          PENDING    |     PENDING
                                     |
                                     |
         +-----+          +-----+    |     +-----+
I/O Q -->|  X--+--------->|  0  |    +---->|  0  |
         |  0  |          |  0  |          |  0  |
         |     |          |     |          |     |
         +-----+          +-----+          +-----+
         BLOCKED          BLOCKED          DORMANT


         +-----+
         |  0  |
         |  0  |
         |     |
         +-----+
         ACTIVE
```

Figure 2. RAMP System Operation

The various task-scheduling operations are implemented in the present RAMP systems as a collection of subroutines and entry points. Since these operations make heavy use of the dynamic storage allocation routines, which are constrained to operate in task-time, tasks can be created and destroyed only in task-time. However, a blocked TCB can be scheduled on the CPU queue in real-time using a subroutine designed for this purpose. In the present implementation all of the task-scheduling operations are contained on a single 128-word page of PDP-8 memory. In this implementation all TCB's must lie in a single core bank, which is declared by an assembly parameter. The operations provided are described below:

INSERT - A task-time subroutine which allocates a TCB of specified length, schedules it last on the CPU queue and presets the first three words as the CPU queue link, the return link and the initial instruction address (entry point) respectively. Both the CPU queue link and the return link are set to zero by this subroutine. If a WAIT operation is initiated following the INSERT operation, the return link is set as a pointer to the TCB which initiated the WAIT operation. Upon return from the INSERT subroutine an auto-index register is set as a pointer to the allocated TCB so that device table pointers and parameters can be easily copied. The length of the TCB requested is given upon entry to the subroutine. A special exit indicates that insufficient storage is available in which to construct a TCB of the requested length.

CHAIN - A real-time subroutine which schedules the indicated TCB on the CPU queue. The calling sequence is designed to be convenient in cases where a TCB is to be removed from an I/O queue and scheduled on the CPU queue.

RETURN - A task-time entry point which causes the TCB of the calling task to be returned to the allocatable storage pool and control to be passed to the mother task (if one exists) or to the next task on the CPU queue (if not). Return is made to the mother task without a task-switching operation, so that arguments can be returned to the mother task in common storage areas. A particular word is designated as the return code and may be used to transmit optional information to the mother task.

REQUE - A task-time entry point which causes the TCB of the calling task to be scheduled at the end of the CPU queue and await its turn next to be activated by the supervisor. This entry point is used when a system resource such as core storage temporarily cannot be acquired by a task, which by convention then retries after all pending tasks on the CPU queue have been activated.

BUSY - A task-time subroutine (no return) which causes the same action as the REQUE operation except that the instruction address in the TCB is changed to point to the next word following the BUSY subroutine call.

DELETE - A task-time entry point which simply causes the next TCB in the CPU queue to be scheduled without affecting the TCB of the calling task. This action is used after a task has been blocked on an I/O queue.

Whenever a task is made active by the supervisor a switch is set so that the task can determine whether the entry was due to CPU queue processing or due to a return from a daughter task. The distinction is important in connection with the asynchronous attention (see below). Also, by convention, certain fields of the TCB are stored in reserved locations whenever a task is activated. Most commonly (but not necessarily) these are used as device table pointers for the assigned I/O devices.

4.2 Real-Time Operations

By definition, real-time operations occur only when the system is disabled for interrupts and are not subject to task-switching operations. The construction of the real-time system is therefore completely conventional and temporary storage allocations are straightforward. Most devices attached to the various RAMP systems sustain transmission on a character-by-character basis, and generate an I/O interrupt for each character transmitted. A few devices sustain transmission on a block-transfer basis, using the PDP-8 three-cycle data break facility. Interrupts are generated by these devices following the block-transfer operation. An interrupt is taken by the CPU hardware when a device interruption condition becomes pending and the interrupt system is unmasked. The interrupt is taken as a forced subroutine jump to location zero in memory, following which each device must be interrogated in turn to identify the one requesting service. Following identification, the requesting device is serviced and the interruption condition is cleared at the device.

If (as is not possible on the unmodified PDP-8) it were possible to selectively mask off interruption conditions at the various I/O devices, then the interrupt processor could be written in several levels, each higher level able to interrupt a lower level, and so forth. If interruption conditions must be masked off from some devices but unmasked in others for the purpose of establishing such a priority ranking, then the masking hardware must be built into each device separately. In one RAMP system intended for message-switching applications, permissable device service delays have been controlled in this manner and through careful design of the peripheral equipment.

It has proved convenient in the various RAMP systems to construct the I/O interrupt system as a collection of _device service routines_, each of which services a given device operating with a given transmission protocol. A common subroutine, called the interrupt identifier is held responsible for the identification of each device interrupt, storing status information from the device, clearing the interruption condition at the device and finally calling the appropriate device service routine. These operations are driven by a set of delicately constructed interrupt control block (ICB) tables (see Figure 3), which are designed so that new devices can be added permanently or dynamically with minimum changes to the basic system. Using this hierarchy it is possible to assign transmission protocol routines rather independently to the actual devices in the system. In this manner it is possible, at least in one system, to specify any of several communication protocols for use on a single transmission device.

```
+------------------------+
|     SKIP/TEST ICT      |
+------------------------+
|   LINK TO NEXT ICB     |
+------------------------+
|    READ/CLEAR ICT      |
+------------------------+
| DEVICE TABLE POINTER   |
+------------------------+
|  DEV SERV ROUT ENTRY   |
+------------------------+
```

Figure 3. Interrupt Control Block (ICB) Format

The structure of the ICB varies among the various systems, and depends upon the peculiarities of the attached I/O equipment. In most systems involving only teletypewriter, paper tape and "simple" control-unit interfaces, the ICB can take the form of Figure 3; but, in more complex systems involving high-performance control-unit interfaces, a much more complex structure has been necessary.

Fields in the ICB specify the device table pointer (see following section) and the initial instruction address of the device service routine. Upon entry to this routine, reserved locations in common scratch areas are set as pointers to control blocks assigned to the device. Using information found in these blocks, the routine performs the service indicated, perhaps involving restarting the I/O device. Note that, using this hierarchy, the device service routine can be called from a task-time routine (after first masking off the interrupt system of course) for the purpose of starting the I/O device. This operation has been dubbed the "tweak" in the current vernacular.

4.3 Input/Output Operations

The general philosophy of I/O processing in the RAMP systems has been to perform control and line-editing processes in real-time device service routines using preallocated buffers and a set of common buffer management routines. Most devices attached to the various systems provide data transmission on a character-by-character basis using an interrupt for every character transmitted. Accordingly, input real-time routines assemble characters serially from the the input device as the interrupts occur and place them in an input buffer, performing code conversion and line-editing operations during the process. When a record-ending character is recognized, the first TCB on the input queue is scheduled on the CPU queue and the corresponding task becomes pending. This task may take the form of a transmission operation, which merely copies the record from one device to another, or a processing operation in which the record is interpreted as commands or data to the appropriate job program. While the task-time processing is going on characters may continue to arrive from the input device for the next following record. In this manner several records may be stored in an input buffer up to the capacity of the buffer.

The buffering operation required here calls for a carefully tailored set of buffer management routines (see Section 4.5) using a cyclic type of buffer construction in which the buffer is allowed to "wrap around," so to speak, so that the next character put in the buffer following the one at the highest buffer address will be at the lowest buffer address. To facilitate these operations a standard _buffer control block_ (BCB) is maintained for each cyclic buffer. The BCB for a buffer is usually allocated and preset when storage for the buffer itself is allocated.

Since most transmission operations involve storage allocation in one form or another, a requirement exists to know the size of a particular record before transmission begins. For input operations this is performed by the buffer management routines themselves such that the first word read from the buffer for each record is the length of the record itself. If the input record is to be simply transmitted unaltered to an output device, then the address of its BCB is passed to the output device service routine, which then transmits the record and disposes of the input buffer as necessary. If the output record is generated by the system itself, rather than as a direct result of an input record, then the record size is computed by the system prior to allocation of the buffer.

Each I/O device attached to the system is identified by a pointer to the device table, which contains a two-word entry consisting of a unit control block (UCB) and a device control block (DCB) pointer. These entries are initially zero when the device is unattached or not-operational and are filled in when the device is enabled for operation (see Section 6.1).

The UCB contains, among other things, the use count and enable flags (in some systems), I/O queue headers, status words and switches, BCB's and sometimes the buffers themselves. The UCB storage is allocated only when the device is enabled; otherwise no claim is made on the allocatable storage pool. The structure of the UCB depends not only upon the particular system implementation, but upon the requirements of the individual device as well. Conventionally, however, the first three (and sometimes four) words of the UCB are similarly structured as shown in Figure 4. For ease in dynamically enabling, disabling and reconfiguring I/O devices, a special set of enable tables is established for use by the system bootstrap and enable task. These tables establish for each device type (that is DCB) in the system the extent and preset values assigned to the UCB when the device is enabled.

```
+------------------------+
|     ENABLE CONTROL     |
+------------------------+
|   ATTENTION I/O QUEUE  |
+------------------------+
|     READ I/O QUEUE     |
+------------------------+
|     WRITE I/O QUEUE    |
+------------------------+
|   REC COUNT/REC SIZE   |
+------------------------+
|   TASK-TIME SWITCHES   |
+------------------------+
|   REAL-TIME SWITCHES   |
+------------------------+
|                        |
+       PARAMETERS       +
|                        |
+------------------------+
```

Figure 4. Unit Control Block (UCB)

The DCB (see Figure 5) contains, among other things, pointers to subroutines and state-transition tables used by the device service routines. The nature of the DCB information is read-only and can be shared by several devices which, nevertheless, are assigned separate UCB's. In the current implementations all UCB's must lie in a single core bank and all DCB's must lie in a single core bank. However, the device table, the UCB's and the DCB's can lie in different core banks as determined by an assembly parameter.

```
+------------------------+
|  ASYNCH TWEAK ENTRY    |
+------------------------+
|  ASYNCH BLOCK ENTRY    |
+------------------------+
|   READ TWEAK ENTRY     |
+------------------------+
|   READ BLOCK ENTRY     |
+------------------------+
|   WRITE TWEAK ENTRY    |
+------------------------+
|   WRITE BLOCK ENTRY    |
+------------------------+
|  CONFIGURATION TABLE   |
+------------------------+
|   CONFIG ROUT ENTRY    |
+------------------------+
```

Figure 5. Device Control Block (DCB) Format

The queue headers indicated in Figure 4 are used by the system as heads of I/O queues, and are assigned as the attention, input and output queues respectively. If an input operation is pending at the device then the input queue is nonempty and similarly for an output or an attention operation. An input operation is initiated by the READ task, with the location of a BCB and the length of the record given when the READ task returns to its mother task. Each input queue then is a chain through the TCB's of one or more READ tasks, each of which is in the blocked state. When a record-ending condition is recognized, the first TCB in the input queue is scheduled on the CPU queue. When the READ task is activated it establishes the BCB pointer and computes the record count. Following return to the mother task calls can be made on the system subroutine RDASC to fetch characters from the input buffer.

An output operation is initiated by the WRITE task, with the location of a BCB given as part of the WRITE task TCB. Each output queue then is a chain through the TCB of one or more WRITE tasks, each of which is in the blocked state. When a record-ending condition is recognized the first TCB in the output queue is scheduled on the CPU queue. When the WRITE task is re-activated, the buffer-release bit in the BCB is inspected. If it is set then the BCB and its buffer are returned to the allocatable storage pool.

In contrast to those tasks blocked on the input and output queues, those in the attention queue are not necessarily in the blocked state, but may be in either the active or dormant states. The TCB's in this queue are set up by an explicit subroutine call and remain in the queue until removed either when an attention signal is recognized at the device or by an explicit subroutine call. A task whose TCB is placed in the attention queue may create daughter tasks, wait for their completion, perform I/O operations and in general carry on all normal processing not involving the pending state. When an attention signal is recognized by the device, the first TCB on the attention queue is scheduled on the CPU queue and so becomes pending. The coding within the job program can be arranged so that a return from a daughter task in the active state can be differentiated from an entry initiated from the pending state, so that the attention condition can truly be recognized as an asynchronous task interrupt. It is this type of operation which causes those strange exceptional situations mentioned in Section 4 involving a task being in two states simultaneously.

Two attention queue operations are provided in the basic system:

SETATN - A task-time subroutine which links the TCB of the caller on the attention queue for the device indicated as the first device table pointer in the TCB (see Figure 1).

CLRATN - A task-time subroutine which unlinks the TCB of the caller from the attention queue for the device indicated as the first device table pointer (see Figure 1). A special exit from this subroutine indicates when the TCB is not on the attention queue, in which case it must by deduction be pending on the CPU queue.

Note that the attention queue is constructed as a first-in-last-out queue, that is, a pushdown stack. Thus the most recent SETATN call defines the TCB to be scheduled upon the recognition of an attention condition. Also note that the inherent asynchronism between these operations and the device operations require the special exit indicated in the CLRATN subroutine. If CLRATN has been called and returns via this special exit, then an attention condition is pending on the CPU queue but has not yet been taken. The coding in the various job programs becomes rather interesting in such cases.

In summary, the real-time and input/output systems are constructed of a collection of subroutines, each serving a number of devices of similar type and designed to be invoked by either the interrupt identifier or by a task-time routine. Each device attached to the system is assigned a device table entry, which contains pointers to the UCB and DCB which controls and records status for the device. Each input real-time routine assembles and edits characters serially using a common set of buffer management routines and schedules a TCB on the CPU queue upon recognition of a record-ending condition. Each output real-time routine transmits characters serially from its output buffer and schedules a TCB on the CPU queue upon recognition of the record-ending condition.

4.4 Storage Allocation

If any subsystem can be identified as contributing the most to the capabilities of the RAMP multiprogramming systems, it most certainly would be the dynamic storage allocation subsystem. In the systems described here this subsystem is used to allocate task control blocks, buffer control blocks, unit control blocks and the buffers used both by I/O devices and sometimes for program residence. The general technique is outlined here; refinements and generalizations are immediately apparent.

At any time during the operation of the system, storage is fragmented into many blocks, some belonging to a task or I/O device, and some representing allocatable free storage. Assume that the first word of each of the latter blocks contains the displacement (in words) to the next block following, that is the total size of the block. Now construct a chain through all these blocks using the second word in each block. This chain is called the _freespace thread_ and is maintained in order by increasing block size. When a block is allocated to a task or I/O operation, all words in the block are available for use. If the first word in the block (the size word) is altered, however, it must be restored before the block is returned to the freespace thread.

The operation of allocating a block to a requesting task or I/O operation then begins with a search along the freespace thread for the smallest block just large enough to satisfy the request (see Figure 6). If such a block is found, then it is unchained from the thread and split into two subblocks, the first of exactly the size requested and the second of a size equal to the actual size less the requested size. If the second subblock is of nontrivial size, then it is immediately reinserted on the thread at the proper position depending upon its size.

```
       +-----------------------+
       |                       |<-- FREESP. THREAD
       |         FREE          |
    +--|                       |
    |  +-----------------------+
    |  |                       |
    |  |       ALLOCATED       |
    |  |                       |
    |  +-----------------------+
    +->|                       |
       |         FREE          |
    +--|                       |
    |  +-----------------------+
    |  |                       |
    |  |       ALLOCATED       |
    |  |                       |
    |  +-----------------------+
    |  |                       |
    |  |       ALLOCATED       |
    |  |                       |
    |  +-----------------------+
    +->|                       |
       |         FREE          |
       |                       |
       +-----------------------+
```

Figure 6. Storage Allocation Operations

The operation of returning a block to the freespace thread proceeds in three steps. In the first the thread is scanned to find the block immediately preceding the block to *be returned, that is the block at the next lower storage* address. If one is found, then it is removed from the thread and concatenated with the block to be returned. In the second step the thread is scanned in a similar manner for the next following block, that is the one at the next higher storage address. If one is found, then it is removed from the thread and concatenated with the block to be returned. In the third step the new block, which by construction here cannot be preceeded or followed by a block on the freespace thread, is inserted on the thread at the proper position depending on its size.

This technique is designed to minimize the fragmentation of storage into many small blocks, no one of which may be large enough to satisfy a particular storage request. In the current implementation a set of routines of this type have been coded in a single 128-word page of PDP-8 memory and execute in 0.3-2.0 milliseconds per allocate/deallocate cycle, assuming randomly distributed block sizes to 128. In the current implementation blocks can be allocated on any of several threads in any memory bank, although each thread is constrained to lie entirely within one bank. Thus it is possible to maintain a number of freespace threads with some threads entirely contained in blocks allocated from other freespace threads. These routines have been modified to provide a storage-aligned allocation for purposes of program loading on page boundaries.

4.5 Buffer Management

Although device transmission in a RAMP system is specified on a record basis, most of the slower-speed devices must transmit data on a character-by-character basis. Furthermore, in the case of input keyboard devices, the input data stream must be edited on a character-by-character basis using special control characters embedded in the input data stream itself. These requirements are satisfied by a set of buffer management routines designed to present a uniform interface to the various device service routines in the system. These routines can be called directly in real-time and, after masking off the interrupt system, in task time.

Most buffers are structured as _cyclic buffers_, such that the next character placed in the buffer after that character at the highest buffer address will be at the lowest buffer address. In those cases where the buffer and its buffer control block (BCB) are reallocated before every record, this wrap around feature may not be used. Such buffers are called _linear buffers_, although for purposes of standardization they are described in the same BCB format.

At each call on these routines a single character is either placed in the buffer or removed from it or a single editing operation is performed. Upon exit from any routine it may be determined whether the buffer has overflowed, underflowed and, in some cases, whether this is first character or last character in the buffer or whether an editing operation was successful or not. Except in the case of those buffers used in editing operations, the characters processed in a buffer are not interpreted in any way, so that the full twelve-bit word is available for transmission. In editing operations the high-order (sign) bit is used to delimit an end-of-record condition. The remaining bits are not interpreted, however.

All buffer operations involve use of the buffer control block, which is either four or five words in length as shown in Figure 7. The structure of the BCB permits its storage allocation disjoint from its buffer, although the allocation is in fact constrained to lie in the same core bank. The first word of the BCB contains a flag (W) indicating whether the buffer has wrapped around, a bit (R) used to determine whether the BCB and its buffer can be released following transmission and a field containing the maximum size of the buffer. The second word is a pointer to the last character put into the buffer (put pointer) and the third word is a pointer to the last character fetched from the buffer (get pointer). The fourth word is the address of the first character in the buffer (begin pointer) and is used to reset either the put pointer or the get pointer should the buffer wrap around.

The structure of the BCB and its buffer differs slightly between those buffers that are used in editing operations and those that are not. For editing operations an extra word is inserted at the beginning of each record. This word contains the number of characters stored in the record and is computed automatically by the appropriate editing routines. The fifth word in the BCB, used only during editing operations, is a pointer to the first character in the current record. This word may be omitted from those BCB's not used in editing operations.

```
+-+-+--------------------+
|W|R| BUFFER SIZE        |
+-+-+--------------------+
|      PUT POINTER       |
+------------------------+
|      GET POINTER       |
+------------------------+
|     BEGIN POINTER      |
+------------------------+
|     RECORD POINTER     |
+------------------------+
```

Figure 7. Buffer Control Block (BCB)

5. COMMAND OPERATIONS

In almost all applications of the RAMP systems described herein some kind of operator control of the system is required. In very many of these applications a rather delicate control of system parameters by relatively untrained operating personnel is anticipated. It seems quite typical that the command language interface is the most often-modified portion once the system becomes operational. For these reasons, the operational interaction segments of the RAMP systems have received a careful development so that an easily expandable modular construction could be achieved at relatively low cost in system size and operating speeds. The <u>command language interpreter</u> (CLI) is the basic subsystem in which these operations are implemented and will be described in these sections.

The CLI is implemented as a task associated with an I/O device, called the <u>command source,</u> which is allocated when the CLI task is created. During operation, the basic system operational commands are assumed to originate via this device. From time to time commentary may be generated by the system, either as a result of commands entered via the command source or as a result of exceptional conditions recognized within the system. This commentary is produced on an I/O device, called the <u>command sink,</u> which is assigned to the CLI task on demand. In some situations, namely those involving communications store-and-forward operations, a third I/O device, called the <u>copy sink,</u> may be assigned on demand to the CLI as the destination of all non-command traffic generated by the command source. With this organization, a number of CLI tasks can be outstanding at any particular time, each one assigned to a particular command source, some of which possibly invoked by others.

A special argument is included in the task control block (TCB) assigned each CLI task. This argument is a set of parameter switches which determine, among other things, whether the default operation identified with the command source is <u>copy</u> mode or <u>command</u> mode. If the default mode is copy, then each input message originating via the command source is transmitted unaltered to the copy sink with the following exception: Each input message processed by the CLI is inspected for the selection code assigned the particular system, usually an USASCII SOH-letter sequence. If the selection code matches, then the message is processed by the CLI itself and is not propagated to the copy sink. If the default mode is command, then each message generated by the command source is processed by the CLI directly. A special CLI command is available to cause a particular message to appear at the copy sink.

A command operation is specified completely as a single record; commands are not normally continued on following records. The syntax of a command is described recursively as a keyword followed by a list of operands which may themselves be keywords with their own lists of operands. A command function is identified with each keyword name; and the arguments to this function are provided by the list of operands, each of which is represented by a value. The command function itself produces a value, which may be used as an operand.

There are two syntactic forms for these commands, the free form and the bound form, which are distinguished only in the manner of separation of the operand-field elements. In the free form the operand-field elements are separated from one another and from the keyword by one or more blanks and the operand field is terminated by an end-of-record character. In the bound form the operand-field elements are separated by special break characters, which are most often commas and equal signs, and the operand field is terminated by a blank. Missing or defaulted operands in the free form are indicated by special symbols or reserved names; missing operands in the bound form are indicated simply by the juxtaposition of two break characters. In some systems special operand-field constructions are prescribed which require syntax specifications more complex than these. Even in these cases the syntactic form assigned to the field, once the scan has terminated on that field, is still a keyword.

Now, the basic atomic element with which all commands are constructed is the keyword. A keyword can take the form of a sequence of digits representing a numerical constant, a sequence of letters designating either a self-defining constant or the name of a procedure which defines a value, or a combination of the two. The value of a numeric constant is determined directly from the sequence of digits using either binary, octal or decimal conversion algorithms. The value of a letter string is determined by a table-lookup search, as is the entry point assigned to a procedure name.

The values assigned to letter strings, the conversion radix used for digit strings and the various operand-format flags and privilege classes are stored in a tree structure called the keyword dictionary. The basic operand-field scanner is a recursive subroutine called the keyword interpreter; and its operation is as follows: At a point in an operand field scan a pointer to a keyword dictionary entry is maintained on a pushdown stack. This entry establishes what form of operand field scan is allowed (digit string, letter string, self-defining constant or procedure name), what privilege class is required to access this keyword, what conversion radix to use in the conversion of digit strings and, finally, a pointer to another dictionary entry which establishes a list of letter-string names which are valid in the operand fields of this keyword. During the operand field scan numeric strings and self-defining letter strings are converted and processed as found. If a procedure name is found, then the entire process recurses and a new keyword dictionary pointer is established at the entry corresponding to the procedure name. In this fashion the tree-structured keyword dictionary is interpreted as a function of the names of the various procedures occurring in the operand fields and the order of their occurrence.

Corresponding to each procedure name is a segment of code in the CLI itself which implements the actual function required. These segments are usually small and consist of recursive calls on the keyword interpreter, calls on other system subroutines and references to system variables. It is clear from this description that new commands can be added rather conveniently and even that dictionary entries and procedure segments can be added and deleted "on the fly" during system operations involving dynamic loading and overlay procedures. The keyword interpreter itself occupies about a page of memory in the current implementation, while dictionary entries are three or four words in length and typical procedure segments are perhaps a dozen instructions. The keyword interpreter calls upon the input formatting routine RDBCD (see Section 6.2) for all of its input text. The various procedure segments often make calls on the output formatting routines.

The task control block for the CLI itself contains several parameters which establish among other things the identity of the command source, command sink and copy sink. Also included is a word containing several switches and small fields. The TCB for the CLI task is shown in Figure 8.

The full generality described in the preceding paragraphs is not needed in all the various systems, of course; and various subsets of this general implementation are found among the several systems. In one particularly interesting variant an internal line-structured text file has been implemented in connection with a text-editor algebraic language interpreter (Reference 6). In this implementation commands can be entered into the text file via typical text-editing procedures and interpreted as a separate operation. This editing/interpreting function proved so compact and useful that it has been incorporated into other systems, notably the Data Concentrator (see References 5, 7). In this fashion it is possible to prescribe complex control procedures in a more compact form than in the basic machine language.

```
+-----------------------+
|   LINK TO NEXT TCB    |
+-----------------------+
|  LINK TO CALLING TCB  |
+-----------------------+
|      ENTRY POINT      |
+-----------------------+
|  CMD SOU DEV TBL PTR  |
+-----------------------+
|  CMD SIN DEV TBL PTR  |
+-----------------------+
|  CPY SIN DEV TBL PTR  |
+-------+---------+-----+
| FLAGS | PRIV CL |BANK |
+-------+---------+-----+
```

Figure 7. Command Language Interpreter (CLI) TCB Structure

Although not all the various RAMP systems have the same command repetoire and the same command syntax, naturally, all have certain basic minimum capabilities, including those to display and alter memory locations via the operator's console. Those systems including the time-of-day clock (see Section 6.3) have the capability to set the time-of-day cell from the operator's console in hours:minutes:seconds format. Provisions are included in all systems including more than one keyboard/teleprinter or high-speed reader/punch to specify which of these devices is to be used as the command source, command sink and copy sink in any particular operation. Some systems have a complex hierarchy of commands to enable and disable input/output devices and to prescribe their operational characteristics as terminals to the system. Finally, those that include the text-file facility mentioned just above include provisions to test various conditions and branch among the lines of the text-file itself. A few of the more interesting commands will be described below. All of these commands have been implemented in the Data Concentrator.

DISPLAY - Display in octal format selected memory locations. This command causes a special task to be created which itself generates the output text. If more locations are requested than can be printed on one line (currently eight), then this task remains in operation until all lines have been generated. In such a case the display task sets an attention interrupt (see Section 4.3) on the command source device. If an attention is received from that device, the display task is terminated at the end of the current output line.

ALTER - Alter selected memory locations using data entered in octal format.

PARAMETER - Set default parameters in the TCB of the CLI task issuing this command. Using appropriate keywords the command source, command sink and copy sink devices can be changed, as well as the privilege class and DISPLAY/ALTER core bank.

SET - Set system default parameters. Using appropriate keywords, the time-of-day clock, broadcast/signon messages and startup/shutdown flags can be manipulated.

TASK - Create a new CLI task with a TCB as specified. Using appropriate keywords the invoking CLI task may be specified to either continue or to wait for the completion of the invoked CLI task.

OFF - Offline device. If the specified device is in the available state, then place it in the offline state.

ON - Online device. If the specified device is in the offline state, then place it in the available state.

SENSE - Print on the command sink certain fields of the device tables and unit control blocks of the devices specified.

ENABLE - Enable device (see Section 6.1). Using appropriate keywords the device type and default operational attributes can be established and the invoking CLI task can be specified to either continue or to wait for the completion of the ENABLE operation.

DISABLE - Disable device enabled by the ENABLE command.

HALT - Transmit asynchronous HALT to device, placing it in the purge state (see Section 6.1).

GOOSE - Transmit asynchronous attention to device (see Section 4.3).

ECHO - Transmit message following (on the same command line) to the device specified.

BROADCAST - Transmit broadcast message to the devices specified.

EDIT - Update the internal line file at the line number specified using the message following (on the same command line).

CONTROL - Establish new operational parameters for the devices specified.

6. SPECIAL OPERATIONS

In some of the RAMP systems special supervisor operations have been implemented, some of which have general application in other systems. These operations include very general device allocation procedures, I/O utility routines, interval timer and time-of-day clock and system initialization and configuration subsystems. New systems have been synthesized by picking and choosing among these subsystems, depending on the application of the particular new system, for those subsystems which are useful. For the most part these subsystems are modular and can be added and deleted from the basic system without materially disturbing other system components.

6.1 Device Allocation

In most of the operating RAMP systems the allocation and configuration of I/O devices can be performed when the system is first initialized following either a power-on reset or initial program load (see Section 6.3). In some systems however, in particular those intended for store-and-forward operations, the allocation and configuration operations may become quite complex and may involve the generation of special probe sequences for device identification and the modification of device service routine characteristics "on the fly." In such systems several characteristics are desirable, among them the following:

1. Core storage for control blocks, I/O queues, etc., should not be allocated if the device is not in use.

2. A device may be allocated to only one particular task, although any number of other tasks may use it.

3. If a device is allocated by a task, then that task is solely responsible for its deallocation.

4. A device should be marked busy (for allocation purposes) when first allocated to a task and marked non-busy only when all time-dependent disconnect sequences have been completed.

Some of these characteristics involve rather interesting architectural requirements, which will be the topic of this section.

As discussed in Section 4.3, corresponding to each I/O device attached to the system is a two-word device table entry. One word of this entry points to a unit control block (UCB) which, presumably, is allocated only when the device is active. The other word of this entry points to a device control block (DCB), which is a fixed table of control information peculiar to the device type. Typically, a collection of devices such as teletypewriter terminals are assigned a contiguous block of device table entries such that each teletypewriter is assigned a unique UCB and a common DCB. A device will be said to be in the <u>available</u> state if its DCB entry is zero, and without respect to its UCB entry, and in the <u>reserved</u> state otherwise (see Figure 9). A device can be switched between the available and the reserved state only when the UCB entry is zero, however. A device is switched from the available to the reserved state prior to the configuration or offline operations, and is switched from the reserved to the available state following the disconnect or online operations. All data transfer and device control operations are ignored in the available state and behave as if the device were in fact nonexistent.

	DCB set	UCB alloc	ACT bit	HLT bit	USE cnt
Available	0	x	x	x	x
Reserved	1	0	x	x	x
Configuration	1	1	0	0	x
Working	1	1	1	0	0
Busy	1	1	1	0	1
Purge	1	1	0	1	1
Disconnect	1	1	0	1	0

Figure 9. Device Allocation States

Once a particular device has been switched from the available to the reserved states, then a sequence of operations may be necessary before the device can be used in data-transfer or control operations. These operations are performed by a special ENABLE task whose parameters are the device table pointer and a DCB pointer for the device type in question. An invoking task may issue a WAIT for the ENABLE task in which case a return code indicates whether or not the operation was successful. Following a successful ENABLE task completion the device may be referenced for

READ, WRITE and HALT operations (a SENSE operation is always valid in any state.). Interrupts are ignored in the reserved state.

The ENABLE task proceeds by first allocating and presetting a UCB using information found in the configuration tables. Once the UCB has been allocated and preset the device is said to be in the configuration state, in which interrupts will be serviced and CTRL operations honored. READ or WRITE operations in this state will be requeued, however. Once in the configuration state a series of quite complicated operations may be performed which are designed to identify the type of terminal, if the device is connected to a data set, or to wake up and start a mechanical device, if the device is a printer or a card reader. During these operations the UCB may be deallocated and reallocated in different formats, timeouts may be initiated and other tasks may be generated; but, in any case, if the ENABLE task terminates successfully, the UCB and DCB entries have been stored correctly in the device table and the device has been placed in the working state and is ready for use. If the operation terminates unsuccessfully, for instance due to a wrong-number call on a data set, the the ENABLE return code is set appropriately and the device is returned to the reserved state.

Now, once the ENABLE task operation has been completed, the device is allocated to the task issuing the ENABLE, although it may be used by any number of other tasks. By convention, a task which issues a number of READ or WRITE operations on the device first increments a use count in the first word of the UCB for that device. When these READ/WRITE operations have been concluded, this word is decremented. This word is also incremented if an interval timer operation is set on the device (see Section 6.3) and decremented either when the timer operation is cleared or by a timer interrupt. These actions are done so that the device will not be disconnected until all pending time-dependent operations have been terminated.

A device can be disconnected at any time by a HALT task operation. This task may be issued by any task at any time, either explicitly on the part of an operator or internally due to a hardware malfunction detection (e.g. Data set on-hook indication). A HALT operation is also implied by a DISABL task operation. Following the HALT operation the device is left in the purge state and any further operations, including interrupt service, are ignored with the exception of the DISABL task operation. The HALT operation usually causes some device-dependent activity which results in a request for disconnect. In the case of a data set the Terminal Ready control lead is dropped, causing the data set to go on-hook. In the case of a System/360 interface operation a special sense bit is set indicating to the System/360 program that the job should be signed off.

The device now remains in the purge state until the device service routine itself recognizes that disconnect has in fact been effected. This condition, recognized by the device service routines themselves, causes the device to be placed in the <u>disconnect</u> state, in which all operations except DISABL are ignored. The disconnect operation itself causes all tasks blocked on the attention, read and write queues to be indiscriminantly scheduled pending on the CPU queue. Obviously the various task returns must be coded to recognize this pathology, since I/O operations may be only partially completed. The principal result of this violence is that the task which issued the ENABLE, or one of its daughter tasks, receives an attention interrupt, which causes it to clear its current WAIT condition, usually by issuing a ATTN task on some device. Eventually, during the recovery procedure, the pathology is discovered on the dying device, which is still allocated to the task which issued the ENABLE. This task then by convention issues a DISABL task operation to disconnect the device from the system.

Note that the issuance of the DISABL is not necessarily synchronous with the trauma of the device as it progresses through the purge state to the disconnect state; and, furthermore, the decrementing of the use count as the various operations pending at the device go "down the drain" is asynchronous with both of these operations. Therefore, unless the device is already in the disconnect state and the use count is zero, then the DISABL task itself is blocked on the attention queue of the device's UCB. Following satisfaction of both of these conditions the DISABL task is scheduled pending on the CPU queue. In the final phase of DISABL processing the UCB is deallocated and the device is again placed in the reserved state. Following return to the invoking task the device can be placed in the available state and the invoking task, now assured that all I/O responsibilities have been discharged, can continue its own operations.

These rather complicated procedures are necessary in any system which allows I/O devices to be shared among several tasks, although the actual code involved is not as massive as might be suspected. The addition of these features costs perhaps a page of code in the current implementation.

<u>6.2 Input/Output Utilities</u>

In any system involving conversational interaction with an operator, a number of I/O formatting routines are always necessary. These routines provide for the character-sensitive recognition and translation functions for the input operations and the translation functions and text generation for the output operations.

In the current RAMP implementations a single input routine is provided which maps 8-bit USASCII characters into a 6-bit packed representation and determines whether the character is a digit, a letter, a special character or a control character. In some systems the assembly of letter strings as keyword names and conversion of digit strings as numeric constants is a function of the keyword interpreter as driven by the keyword dictionary. In all command input operations the parity bit is suppressed and lower-case USASCII characters are converted to their upper-case equivalents.

Conversion of 12-bit binary words into digit strings in octal or decimal radix is provided by the special subroutines PROCT and PRDEC. Leading zeros may be suppressed using PRDEC. Letter strings packed two six-bit characters per twelve-bit word binary word can be converted into 8-bit USASCII strings by means of PRBCD, which maps a single 12-bit word, and PRCCM, which maps a vector preceeded by a count of the number of 12-bit words. If the time-of-day routines (see Section 6.3) are included in the system a subroutine PRCLK is available to print the time-of-day in the hours:minutes:seconds format. In one RAMP version all output text generated by these routines is assigned the correct even-parity bit in position 8 of the USASCII character. In the others this position is forced to a one bit.

In order to print a comment or a line of text, the system architecture calls for the creation of a WRITE task control block, a buffer control block and the buffer itself. A subroutine GTDEV is provided for this purpose, which defaults the output message to the command sink. This subroutine creates the WRITE TCB, schedules the TCB on the CPU queue and sets the BCB pointer. The BCB is allocated as part of the TCB parameter region together with the buffer itself. The buffer-release bit is not set in the BCB, since the BCB and its buffer are released automatically together with the TCB when the WRITE task returns to its invoking task. The length of the buffer is specified upon the call to GTDEV, which then returns in an auto-index register a pointer to the buffer. Characters may be stored directly in the buffer via the auto-index register or may be formatted and stored automatically using the output format routines described above. A call to CRLF must be made to finish processing the BCB fields when all the text has been stored in the buffer.

6.3 Interval Timer and Time-of-Day Clock

In very many applications of the systems discussed here, a need exists to determine the time of occurrence of an event or the time difference between two events. Most commonly, only the latter facility, called here an interval timer operation, is necessary. In real-time data logging systems a time-of-day clock operation is also necessary. In the typical small machine considered for RAMP application an interval timer is easily implemented either as a special I/O device or as a special cycle within the CPU itself. In either implementation a location in core memory is caused to increment every so often, usually in the 10-100 mS range. When the count overflows an interrupt occurs which is then processed in the usual manner. The manner in which such a device can be incorporated into the RAMP system architecture to provide interval-timer and time-of-day facilities will be the topic of this section.

Due to the inherent multiprogramming nature of a RAMP system, it is common that several independent logical timers are required at any given instant. Some of these timers are necessary for task-time operations in which the task is to be blocked for a certain time interval, while others are necessary for real-time operations in device polling and control. Such an architecture has been implemented using a chain of <u>clock control blocks</u> (CCB) (see Figure 10) which correspond to the set of logical timers in operation at every instant. These CCB's are linked in a CCB chain in a particular sequence as follows: consider the intervals assigned each of n logical timers $t(1),t(2),\dots,t(n)$ and assume $t(i)\le t(i+1)$ for all $1\le i<n$. Then the CCB's corresponding to $t(1),t(2),\dots,t(n)$ are chained in that order and the countdown interval in the (i+1)th CCB is preset to contain the value $t(i+1)-t(i)$. Then each time a timer interrupt is taken, the first CCB entry in the chain is unlinked and a field in this entry is used as a branch address to a real-time processing routine. At this time the interruption condition is cleared and the countdown interval of the next CCB entry on the chain replaces the contents of the hardware timer. Processing continues in this manner until the chain becomes empty, at which time the hardware timer is allowed to cycle through its full-period interval.

A pre-constructed CCB entry requesting an interval of t basic timer steps can be entered in the CCB chain using the subroutine SETIME. This subroutine scans the current CCB chain and totals the cumulated expected countdown interval $t(i)$ at the ith entry. If j is the index of the first entry in the chain such that t(j)>t, then the new CCB is linked between the (j-1)th entry and the jth entry, a new value t-t(j-1) replaces the countdown interval of the (j-1)th entry and a new value t(j)-t replaces the countdown interval of the new entry. Obvious refinements are made if the new entry is either the first or the last in the CCB chain.

```
+------------------------+
|   LINK TO NEXT CCB     |
+------------------------+
|  COUNTDOWN INTERVAL    |
+------------------------+
|   INT ROUTINE ENTRY    |
+------------------------+
```

Figure 10. Clock Control Block (CCB) Format

An already existing entry in the CCB chain can be deleted from the chain using the subroutine CLRTIM. If j is the index of the entry to be removed, then this subroutine will unlink it from the chain and link the (j-1)th entry to the (j+1)th entry. Each entry in the chain starting with the (j+1)th then is corrected by adding to its countdown interval the value associated with the jth entry.

This rather simple and straightforward architecture is complicated by the fact that the hardware timer can increment at any time, and in particular during either SETIME and CLRTIM processing. Timer service routine snarls in the present implementation are avoided by "freezing" the timer before any change is made to the CCB chain and "thawing" it when the processing is complete. In the freeze operation the countdown interval of the first CCB entry is corrected by the elapsed time since the timer was last set; and the timer is reinitialized at zero. In the thaw operation any timer interrupt conditions, which may have become pending due to a timer increment, are serviced; and the countdown interval of the first CCB entry replaces the timer contents.

At certain points during the freeze/thaw operations when the timer is changed, a possible timer increment may be lost. This is because, in the simple machines considered here, the timer can increment after the old contents have been read out but before the new contents have been read in. Even some large machines like the System/360 Model 67 have suffered at one time or another because this could happen. In the Model 67 case the difficulty has been avoided by using a core-to-core move operation that moves the old contents out of the timer and moves the new contents into the timer in one instruction. A special interlock circuit prevents a timer increment during the execution of this particular instruction. In the small machines considered here a timer increment during this sensitive sequence is impossible to detect unless it also causes an interrupt flag to be set. This latter condition is detected and properly processed in the current implementation.

All subroutines which maintain the interval timer operation are contained in a single page of memory in the current implementation and are designed to be removable without affecting other system components (except the time-of-day clock - see below). In the current implementation all CCB's must lie in a particular memory bank defined as a system assembly parameter. It has been convenient in at least one RAMP system to implement some timer interrupt operations as a pseudo-receive or pseudo-transmit interrupt for a particular I/O device. In this system a device service routine may request a timer interrupt which, when taken, appears as a character interrupt. However, using this construction it is necessary that the timer routines appear recursive to the real-time device service routines. This is accomplished by setting a flag in the timer-freeze operation and testing this flag in the interrupt identifier after the device service routine has been called. If the device service routine ever calls a timer service routine, then the interrupt identifier itself will perform the timer-thaw operation, which may involve calling other device service routines. Special interfaces for task-time calls on the timer service routines accomplish this same function.

A real-time or time-of-day clock is easy to implement, given the interval timer operation as described above. This is done in the following manner: a 24-bit time-of-day cell is maintained in units of the basic timer step relative to midnight. Each time a hardware interrupt is taken the countdown interval indicated in the first CCB entry is added to the time-of-day cell. If the CCB chain is empty then the timer has completed one full cycle, and the corresponding interval is added to the time-of-day cell. Each time the timer is frozen a quantity equal to the countdown interval of the first CCB entry minus the current timer value is added to the time-of-day cell. In this manner the time-of-day cell is corrected each time the timer is updated.

Subroutines are provided in the current implementation which set the time-of-day cell using values for hours, minutes, seconds and hundredths of seconds as entered via the command language interpreter. A subroutine PRCLK is available to print the time-of-day in the format HH:MM:SS (HH = hours, MM = minutes, SS = seconds). All subroutines which maintain the time-of-day clock are contained in a single page in the current implementation and are so designed to be removable from the basic system without affecting other system components.

6.4 System Initialization and Configuration

It has been convenient for reasons of debugging ease and error recovery procedures to construct the RAMP systems in a self-initializing fashion. In all systems, at initial program load, selected storage blocks (including the device table) are erased, storage allocation freespace chains are initialized and the CPU queue and CCB chain headers initialized. Then a number of ENABLE tasks are generated and a task servicing the operator's console is created. All of these operations are table-driven so that changes can be easily reassembled in the system.

In the case of one of the systems intended for message store-and-forward operations, the configuration following either an initial program load or a power-on restart is controlled by an inspection of hardware conditions to determine which interfaces and line adapters are operational, following which the operational devices are marked in the available state and certain ENABLE tasks are automatically generated to start the system. This system wakes up with two System/360 interfaces, the line-adapter scan controls, the operator's console and a reserved System/360 interface unit address in the working state, following which further configuration information can be entered either via the operator's console or via the System/360 interface. In particular, the interactive behavior during the initial phase following connection of a terminal to the system can be programmed as a series of commands entered in the internal line-file (see Section 5). Using this file a cycle of inquiry/response can be elicited from the terminal user to determine special requirements prior to connection with the System/360. It is possible using this technique to connect the terminal to either or both System/360 interfaces in the case of a partitioned system where one system is running on one CPU and another on the other. In such cases the controlling line-file is loaded directly into the system via the System/360 interfaces.

In the case of the other RAMP systems, the configuration following initial program load is established simply by a prestored table. However, following a power-on reset the system is restarted at the point following the power-off interrupt. All I/O devices are reset by the power-off condition of course; but, in these simple systems, this does not result in pathological behavior other than the loss of a character or two if a device operation was pending at the same time. The power-off interrupt and power-on reset operations can be supported only if the proper hardware option is installed in the PDP-8 of course.

7. REFERENCES

1. Mills, D.L., A Random-Access Multiple-Program System for Language Laboratories, Language Laboratory, University of Michigan, April 1965.

2. Mills, D.L., A Proposal for a Computer-Supervised RAMP System, Language Laboratory, University of Michigan, May 1965.

3. Mills, D.L., RAMP: A Multiprogramming System for Real-Time Device Control, Concomp Project Memorandum 5, University of Michigan, May 1967.

4. Mills, D.L., I/O Extensions to RAMP, Concomp Project Memorandum 11, University of Michigan, October 1967.

5. Mills, D.L., The Data Concentrator, Concomp Project Technical Report 8, University of Michigan, May 1968. Also in Procedings of University of Wisconsin Engineering Institute, December 1968, pp. 1-113.

6. Mills, D.L., RAMP Architecture in a Utility Calculator System, Concomp Project Memorandum 24, May 1969.

7. Mills, D.L., Topics in Computer Communications Systems, Concomp Project Technical Report 20, May 1969. Also in Procedings of University of Michigan Engineering Summer Conference, June 1969.

8. Frantz, D.R., Brender, R.F., and Foy, J.L., Jr., LOCCSS, A Multiprogramming Monitor for the DEC PDP-7, Concomp Project Technical Report 10, University of Michigan, November 1968.

9. Brender, R.F., Frantz, D.R., Foy, J.L., Jr., and Schunior, T.W., Specialized System Software for Interacting DEC PDP-7 and IBM 1800 Computers, Concomp Project Technical Report 11, University of Michigan, December 1968.

10. Jackson, J.H., An Executive System for a DEC 339 Computer Display Terminal, Concomp Project Technical Report 15, University of Michigan, December 1968.

UNCLASSIFIED
Security Classification

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY *(Corporate author)* THE UNIVERSITY OF MICHIGAN CONCOMP PROJECT	2a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED 2b. GROUP

3. REPORT TITLE

MULTIPROGRAMMING IN A SMALL-SYSTEMS ENVIRONMENT

4. DESCRIPTIVE NOTES *(Type of report and inclusive dates)*
TECHNICAL REPORT 19

5. AUTHOR(S) *(First name, middle initial, last name)*

DAVID L. MILLS

6. REPORT DATE MAY 1969	7a. TOTAL NO. OF PAGES 43	7b. NO. OF REFS 10
8a. CONTRACT OR GRANT NO. DA-49-083 OSA 3050 b. PROJECT NO. c. d.	9a. ORIGINATOR'S REPORT NUMBER(S) TECHNICAL REPORT 19 9b. OTHER REPORT NO(S) *(Any other numbers that may be assigned this report)*	

10. DISTRIBUTION STATEMENT

Qualified requesters may obtain copies of this report from DDC.

11. SUPPLEMENTARY NOTES	12. SPONSORING MILITARY ACTIVITY Advanced Research Projects Agency

13. ABSTRACT

This report discusses multiprogramming systems architectures suitable for use with small machines of the PDP-8 class. Techniques for task and I/O device scheduling, storage and device allocation, buffer and timer management, and command language interpretation are discussed in detail. Illustrative details are freely drawn from a follow-on version of RAMP, a multiprogramming system now used in several applications involving process control, message switching, and terminal control.

DD FORM NOV 65 1473

Security Classification

UNCLASSIFIED
Security Classification

14. KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
multiprogramming						
storage allocation						
device allocation						
supervisory system						
PDP-8						

Security Classification