TOPS–10 Version 7.04
Monitor Internals Course

This document is the student guide that accompanies the TOPS–10 Version 7.04 Monitor Internals Course.

Revision/Update Information: Version 1.0

Digital Equipment Corporation

June 1989

The postpaid READER'S COMMENTS form on the last page of this document requests the user's critical evaluation to assist in preparing future documentation.

This document was prepared using VAX DOCUMENT, Version 1.1

Contents

Chapter 1 Introduction to the TOPS–10 Monitor

Chapter 2 Monitor Cycle

Chapter 3 Command Processor

Chapter 4 Memory Management

Chapter 5 InterProcess Communication Facility (IPCF)

Chapter 6 The Scheduler

Chapter 7 The Swapper

Chapter 8 UUO Processing

Chapter 9 I/O Introduction and UUO-Level Routines

Chapter 10 Device Service

Chapter 11 Disk service

Chapter 12 Scanner Service

Appendix A EBOX/MBOX Accounting

Index

Figures

Tables

Chapter 1

Introduction to the TOPS–10 Monitor

The DECsystem–10 consists of hardware and software designed to allow users to run a variety of programs efficiently and conveniently. The details of the system software will be discussed after developing basic concepts and defining terms involving both the software and hardware.

The DECsystem–10 is specifically designed for interactive multiprogram operation. Normally there are several programs active and control is switched from one to another by the system executive program, or monitor. Programs that are not using the CPU can still have active input and output devices. The overlapping of I/O with the processing of several programs permits efficient use of both the CPU and the I/O devices.

1.1 User Program Addressing

The DECsystem–10's hardware features are designed to facilitate multiprogram operation. There are two basic modes of operation, executive and user. The monitor runs in executive mode with no restrictions on its operations. In user mode, a program can access memory only within areas assigned to it by the monitor. Also, certain instructions can not be executed in user mode. These include all hardware I/O instructions (except for the I/O instructions for devices 740–774, reserved for customers), the instructions to control memory access and mode of operation, and a few instructions reserved to the executive.

Each program consists of instructions, constants, and data areas, which may constitute either one or two segments (high or low) of the user's virtual address space. The hardware and microcode provide a mapping from a virtual address to a physical address.

The user's virtual address space, like physical memory itself, is divided into fixed-size pages of 512 words. Each user's page (called a virtual page) will be assigned a physical page in core. When the monitor initially assigns physical pages to a user's segment, it builds a page table and a page map in order to tell where each of the user's virtual pages resides in core.

On the DECsystem–10, the mapping of a user virtual address to a physical address is accomplished by referring to entries in the user's page map.

On an indirect memory reference, the mapping mechanism is used for each memory reference made in the effective address calculation.

Addresses 0–17 always refer to the hardware accumulators. The KL10 has 8 sets of accumulators, or fast register blocks. Three or four sets are used by the monitor, one set is available for the current user, and portions of two sets are available for the KL10 microcode. The other sets are available for real-time programs.

Also associated with each user virtual page are access bits that provide protection for the monitor and other user programs when this user is running. The monitor fills the page map and sets the access bits only for those entries that the user is allowed to access. Zero access bits in the page table cause a reference to the associated page to initiate a page failure trap or page fault to the monitor.

Before the monitor can allow the user's program to begin, it must pass the address of the user's process table to a hardware register in the MBOX called the User Base Register (UBR). Once the address of the user's process table has been passed to the UBR, the monitor is ready to start up the user's program in user mode. When the user program starts executing, the hardware, because of the user mode flag, uses the UBR to point through the User's Page Table (UPT) to the specified User's Page Map (UPM) for the mapping of the user's program's virtual addresses to physical addresses.

According to this scheme, each memory read results in two actual memory references: one to get the memory mapping data and one to get the user's mapped memory reference. To speed up the memory reference time, the last 512 (1024 with the MCA25) distinct virtual pages referenced have a copy of the associated physical page numbers and access bits stored into a special table in the MBOX, called the hardware page table (HPT). Thus, two actual memory references into main memory must be made only if the information concerning the page referenced is not in the HPT.

In a timesharing system such as the DECsystem–10, it is quite likely that several users might want to run the same program at the same time. The system can do this more efficiently by allowing users to share portions of the program. To allow sharing of code, the program's virtual address space is divided into 2 parts called segments: a pure, or reentrant, segment, and an impure segment. The reentrant segment normally consists of all the constants and instructions that do not change during the program execution. Since this part of the program does not change, a single copy in physical memory can be shared by more than one user program. That is, the same physical page numbers for the pure segment appear in more than one page map, and the pure segment pointers are duplicated to JBTSGN. All parts of the program that are subject to change must be separate for each user.

Normally, the impure segment of the program begins with virtual address 000000_8 and can go as high as 777777_8, or one section. However, that leaves no room for the pure segment. The pure segment, if there is one, usually begins with virtual address 400000_8 and extends as high as 777777_8.

On a KL10, with the 7.03 release of TOPS–10, users can have up to 32_{10} of these sections of virtual memory (controlled by the system manager).

Because the virtual addresses in the pure segment are often greater than those in the impure segment, the pure segment is called the high segment and the impure segment is called the low segment. Note that the terms high and low segment refer to the virtual addresses, not the physical addresses at which the pages of the segment are located. Any instruction can refer to a memory location in either segment. Hence, the two segments function as a single

program. To the program, the only effect of segmentation is that there may be a range of invalid (non-existent) user virtual addresses in between ranges of valid ones.

1.2 Monitor Calls

The monitor performs a number of services for user programs, including all I/O operations. The instruction codes from 040_8 through 077_8 provide the means for programs to request the monitor to perform these services. These operation codes, called Unimplemented User Operators (UUOs), have no hardware function except to give control to the monitor. When a UUO is executed, the instruction is trapped by the microcode and control is given to the monitor. A routine in the monitor then decodes the request and calls a subroutine to perform the requested operation. Each UUO appears as only one instruction in a program, but it actually functions as a subroutine call. Hence, those instructions are sometimes called "programmed operators".

1.3 Interrupts

The KL10 processor has a multiple level priority interrupt system. There are seven levels of priority: one is the highest level priority, seven is the lowest. Each I/O device is assigned to a level and can interrupt any activity running at a lower priority level. Interrupts can also be requested from within a program. Level seven is reserved for software generated interrupts. No devices are assigned to this level.

When an interrupt occurs on level n, the next instruction is taken from Executive Page Table (EPT) location $40 + 2 * n$, and is executed in executive mode. When a vectored interrupt occurs, the device or controller requesting the interrupt supplies a function word to the CPU. The contents of this function word are used to determine the address of the instruction to execute in order to service the interrupt. This allows transfer of control directly to the device service routine rather than through the fixed address of EPT $40+2*n$, as with a non-vectored interrupt. Upon completion of interrupt processing, control is restored to the interrupted program. All accumulators and processor flags must be saved and restored by the interrupt routine.

All DECsystem–10 processors have a clock that interrupts regularly according to the power line frequency. On the KL10 this is done by setting the interval timer to interrupt every 16.6ms (60 HZ) or $20ms$ (50 HZ). This interval is known as a jiffy. This clock interrupt guarantees that the monitor can always, predictably, take control back from a user program. One jiffy is, therefore, the basic unit of CPU time that the monitor allocates to a program. At each clock interrupt, the monitor reconsiders the question of which program to run.

1.4 The Monitor

The monitor provides the interface with which users and user programs interact. It controls each user job in such a way that no user needs to be concerned that there are other users on the same system. The monitor presents the appearance of a complete and independent system to each user. In addition to its control functions, the monitor provides many services to users and user programs. We might think of any function performed upon request from a user as a service and any function performed without a user request as a control function.

Requests for service can come from user terminals as monitor commands or from user programs as UUOs. For example, in response to a command that runs a program, the monitor assigns physical memory to the user job, reads an executable file into memory from some other storage medium, and adds the program to the set of programs sharing the CPU. The most frequent requests for service come from user programs via the I/O UUOs. These UUOs allow a program to access data by file name and block number without being concerned about the physical location of the data. The monitor computes physical addresses on the disk, starts I/O transfers, and handles the resulting I/O interrupts for all user programs.

Control functions are performed as necessary by the monitor, according to algorithms that attempt to give optimum overall system performance. One of the most important of these functions is dividing the available amount of CPU time among the active user programs. Jobs must be stopped when clock interrupts occur and their states must be preserved so that they may be restarted at a later time. The monitor must also decide which user jobs to keep in physical memory and which to "swap out" to disk. In addition, the monitor must decide where to put jobs in physical memory as they are swapped back in or when they change in size.

1.5 Structure of the Monitor

The monitor consists of many separate, though often architecturally related, modules. These modules are more or less independent routines that are executed according to events occurring within the system. Some of these routines operate on a regular cycle based on the clock interrupt; others are called only in response to system events such as I/O interrupts and the execution of UUOs.

The Control Routine is executed on each clock interrupt. It dispatches to the Command Processor, the Scheduler, and the context switching routine, each time it is executed. The Command Processor routine handles commands typed by users. It frequently calls the SAVE/GET routines and the Core Management module when it processes commands to set up and run various programs. The Scheduler decides which user program to run during the next sixtieth of a second, also referred to as a Jiffy. The Context Switching routine saves the computational state of the current job and restores the state of the chosen job, allowing it to run for the rest of the time slice.

The Swapper is called by the Scheduler on each clock interrupt. It transfers user programs between physical memory and disk and attempts to keep the highest priority jobs in memory.

The UUO processor (UUOCON) responds to all requests for service by user programs and specifically handles all I/O required by the user program. UUOCON is device independent and is the same in all monitors regardless of the hardware configuration for which they are built. The device dependent code required for any device is included in the device service routine for that device. Any given monitor is built for a specific hardware configuration and contains device service routines for the devices in that configuration.

The Core management modules (APRSER, CORE1, and VMSER) all handle changes in the size of user jobs and changes in their physical memory locations. These routines are called as needed by the Command Processor, the UUO processor, and the Swapper.

At any point throughout this cycle there could be an interrupt due to completion of an I/O operation or transfer. The interrupt save routine must save and restore the state of the interrupted routine. The lower priority routine normally does not need to give any consideration to the possibility of being interrupted. However, if there is an interaction between an interrupt routine and a lower priority routine, the lower priority routine must be written so that it will work properly with an interrupt on any instruction. If this is impossible, the priority interrupt hardware may need to be disabled for a few instructions when it is critical that no interrupt should occur.

1.6 The Monitor as an Event Processor

Overall, the monitor can be envisioned as a real time program that responds to events occuring within the system. The routines that operate on a regular cycle are called as a result of a periodic event, the clock interrupt. The UUO processor responds to the execution of UUOs and the Command Processor responds to a user typing a command on his terminal. Each I/O device interrupt is an event that results in the execution of a specific interrupt routine.

There is a well-defined function that the monitor performs in response to each event. However, a given event does not necessarily result in the same action all the time; the specific action taken on a given event depends on the state of the system. The system state is represented by many variables in memory and device registers and depends on the past history of the system. The monitor performs a specific predictable function in response to any specific event, depending upon the state of the system at the time the event occurs.

In summary, the monitor both controls user jobs and provides services to them. The monitor presents the appearance of a complete and independent system to each user job by switching control among the user jobs. The monitor runs and stops user programs according to the user's commands and the condition of the program. It handles all I/O operations, according to requests from user programs and attempts to allocate all system resources in such a way as to give the best overall system performance.

Chapter 2
Monitor Cycle

The heart of the timesharing illusion is the monitor cycle. Time accounting, command processing, scheduling, swapping, and context switching take place in this cycle every clock tick. This cycle allows TOPS–10 timesharing to work effectively by reallocating resources on a periodic basis. All knowledge of the monitor is built upon an understanding of this process and serves as the real starting point for the course.

2.1 The Control Routine

Every sixtieth of a second the monitor performs a cycle that, at its conclusion, gives control to a user program. In this cycle the monitor performs all functions except servicing interrupts and UUOs. The functions that are performed in-line within the control routine are discussed in this chapter. We are particularly interested in the manner in which the major routines are called and the circumstances under which the entire cycle is repeated. Functions performed as subroutine calls, such as command processing and scheduling functions, are discussed in later chapters.

2.1.1 Time Accounting

At the beginning of the cycle, CPU usage for the previous cycle is accounted for. If a user program was running, the elapsed time is added to the total time for the job in the Process Data Block (PDB). If the Null Job was running, the elapsed time is added to the total Null Job time in the CPU Data Block (CDB). If the Null Job was running, but there were jobs in one or more Run Queues that could not be run for some reason, the time is considered "lost" CPU time. If there were no jobs in the Run Queues, the time is considered idle time. Lost time is accumulated separately in the CDB, in addition to total Null time.

CPU times are accumulated in a manner that might seem peculiar, at first. Various system programs, such as SYSTAT and some user programs, look into the monitor's tables and expect these times to be in jiffies. † It is desirable to measure and bill CPU times with a finer resolution than the jiffy. The time base meter on the KL10 can measure time intervals with 1 microsecond resolution. In order to use the additional precision without having to change

† Strictly speaking, a jiffy is defined as the time for one cycle of AC line current. Hence, it means one sixtieth of a second or one fiftieth of a second depending on the country in which you are located.

all the programs that look at these time intervals, the interval is maintained in two parts. The table entries that have historically been in jiffies are still in jiffies. An additional word is used to hold the amount of excess beyond the last even jiffy. When the interval is updated, the incremental time is added to the excess. If the excess goes past one jiffy, the excess and the jiffy total are each corrected.

On systems using a time base meter, the time interval for a user job is measured from the time the program is given control until the time it is stopped again. Time spent servicing priority interrupts is included in this total but is assumed to be insignificant. The time spent performing monitor overhead functions (from stopping one user program until starting the next) is measured and accumulated separately in the CDB as overhead. This overhead time can be included into or excluded from a user's runtime as determined at MONGEN time.

KL10 processors have two additional clocks, or meters, that can be used for an even higher degree of accuracy and repeatability than the time base meter. These meters are called the EBox and MBox time accounting meters. The EBox meter counts EBox cycles during instruction execution while the MBox meter counts the number of memory references. These meters can be operated in a mode such that they are stopped during interrupt processing, thus not charging the current user for the EBox cycles used by interrupt processing. The EBox and MBox counts are scaled by the appropriate factors to be equated to CPU run time as obtained from the time base meter. Job accounting using EBox and MBox accounting is more consistent than time base meter accounting since varying instruction execution times due to memory contention and interrupt processing are excluded from the job's accounting data. However, the accuracy of the scaling factor is dependent on the nature of the memory references (read/write) and cache-hit ratio. Refer to Appendix Appendix A for more information.

In addition to CPU time, a total of CPU time is accumulated, weighted by core size for each job. Each time a job accumulates a jiffy of CPU time, its current size is added to this "Kilo Core Tick" total. Most commercial timesharing bureaus base their charges partially on this figure.

2.1.2 Time Limit

It is possible to set a CPU time limit for any job. This is especially important for the Batch Controller, but can be set by timesharing users if they so desire. If a time limit is set up, the remaining time is decremented each time the job accumulates another jiffy of CPU time. When this time limit, contained in table JBTLIM, expires, the job is stopped. A timesharing user can type a CONTINUE command, but a batch job is aborted by the Batch Controller.

2.1.3 System Time Accounting

If the current CPU is the "Policy" CPU, the monitor updates the Smithsonian Date and Time, checks if the KDPLDR (if a KS10 system) should be run, and then checks if there are any commands waiting to be processed. If there are, the monitor calls the Command Processor. The Command Processor chooses one of the waiting commands to interpret and process. After processing, the Command Processor returns to the Control Routine. The Command Processor is a major routine and is discussed in detail in Chapter Chapter 3.

2.1.4 Timing Requests

The next function in the overall cycle is processing timing requests. A timing request is a request submitted by a monitor routine for some function to be performed at a specified time in the future, such as waking a job that has gone to sleep. Each request includes the address of the routine to be called, the time interval in jiffies, and several bits of data to be passed to the routine. The requests, in the form of two word entries, are stored in the table CIPWT. On each clock tick, the monitor decrements the remaining time for each request and calls the routine for any request whose time has expired.

The monitor performs some functions only once per second and once per minute. A counter is decremented on each clock tick to indicate when another second or minute has elapsed. Each time these counters expire, the Once-a-Second or Once-a-Minute routines are called. Among other things, these routines check for Hung I/O devices and send out "Device offline" messages.

2.1.5 Calling the Scheduler

The Scheduler (see Chapter Chapter 6) is called next. The scheduler requeues any jobs that have changed state during the last cycle. Then, if core is scarce, it calls the swapper (see Chapter Chapter 7). The swapper starts swapping jobs out of memory or cleans up jobs that have finished swapping, then returns to the scheduler. The scheduler then decides what user program is to be run next. It does not give control to that program directly, but returns the job number chosen to the overall control routine that called it.

2.1.6 Context Switching

After the scheduler returns, the monitor checks to see if the job number selected to run next was the current job. If so, the monitor checks to see if the user's high segment has moved due to a LOCK UUO, updates the UPM if it has, and then goes on to the next task. If the job number is not the current job, the monitor saves the old job's ACs in the "Shadow Area" in the user's Job Data Area (.JDAT), saves the old job's PC word in the UPT (at USRPC), and, if a multi-CPU system, marks the job as runable after the next cache sweep.

Before giving control to the new user program, the monitor restores all conditions that affect the user program's execution to the state that they were in when it was last interrupted: the same conditions that were saved for the user program that ran last.

The hardware registers that are restored are the User Base Register (UBR), the user's accumulators, and the program counter and processor flags PC word. The PC word is saved immediately whenever the program is stopped. The information necessary to set up the UBR is maintained in the table JBTUPM.

2.1.7 CPU Tick Dependent Code

Before returning control to the user program, the monitor checks to see if the clock cycle was initiated because of a clock tick. If so, the monitor calls routines to handle subsystem specific functions. These routines start output on terminal lines, perform the Queued I/O processing on multi-CPU systems, DECnet functions, functions for various devices, and create any pages that needed to be allocated at clock level.

2.1.8 Returning Control to the User

When the monitor is ready to return to the user, it turns on the accounting meters, if they were disabled to exclude monitor overhead from the user's runtime, and checks for any user specific actions that need to be taken. These actions could be: starting more output if the user program is doing non-blocking I/O, generating Software interrupts (PSI), or handling and trap conditions set by the user.

After taking care of all user specific actions, the monitor returns to the user if the PC was in user mode, or restores the Exec ACs and returns to the interrupted monitor routine.

2.2 Repeating the Cycle

Once control is given to a user program, it may run until the next clock interrupt or until it "blocks" within a UUO because it needs data or a resource that is not yet available. If a clock interrupt occurs during execution of a UUO, the job is not interrupted until after completion of the UUO. After completion of the UUO, the job is stopped in the exit routine of the UUO processor. There are, therefore, three conditions under which a user program might be interrupted and three corresponding entry points to the overall monitor cycle:

1. Clock interrupt occurs while the program is running in user mode. Enter CLOCK1 at routine RSCHED.
2. Clock interrupt occurs during execution of a UUO and, then, the UUO is completed. Enter CLOCK1 at routine USCHED.
3. A UUO routine reaches a point where it can not immediately continue. A clock interrupt may or may not have occurred. Enter CLOCK1 at routine WSCHED.

2.3 Clock Interrupt

The interrupt routine that causes the monitor to begin the monitor cycle originates from the Interval timer, device TIM. This timer is assigned to a very high interrupt priority (usually Level 1, 2, or 3) because the interrupt triggers the maintenance of time of day, which must be accurate. However, there is no urgency for restarting the control cycle. Therefore, the hardware interrupt is used to drive a lower priority "software" clock interrupt. The software clock interrupt is always assigned to Channel 7 so that all I/O device interrupts can take priority over it. The software clock interrupt is also requested by certain other monitor routines in order to start a new cycle before the hardware clock interrupt has occurred. The flag .CPCKF is set by any routine that requests the Channel 7 interrupt; the flag .CPTMF is set only by the interval timer interrupt routine and indicates that an actual clock tick has occurred.

When the software clock interrupt occurs, control passes to the CK*n*INT (where *n* is the CPU number), in COMMON. The CK*n*INT routine checks to see if the flag .CPCKF is set and, if it is, a jump to the routine CLKINT in CLOCK1 is performed; otherwise, the interrupt is dismissed. If the interrupted program was in Exec mode, either UUO level or interrupt level, and rescheduling is not forced (flag .CPSCF is not set), the interrupt is immediately dismissed and the new cycle is delayed until the current monitor function is finished. The monitor stores the current PC word in the CDB and, if doing a forced reschedule, it saves the current Exec ACs in the user's Job Data Area (.JDAT), starting at location JOBDAC. If the

PC was in user mode, the monitor switches to the Exec AC block. The monitor then restarts the overall cycle again.

Before returning control to the user program, the UUO processor checks to see if a clock tick occurred while it was running, if the swapper is trying to force this job out, or if a High Priority Queue (HPQ) job has become runable. If it finds .CPTMF, FORCEF, or SCDRTF set, it passes control to the USCHED routine, which performs a similar function to CK*n*INT. USCHED sets the "user program" return address to the next address in the UUO processor, saves the necessary ACs, and passes control on to RSCHED. When the interrupted program is selected to run again, it will be restarted in the UUO processor at a point just prior to where control is restored to the user program.

2.4 Program Blocked

In some cases, a user program may reach a point where it can not immediately continue. For example, it may execute an INPUT UUO at a time when the next buffer has not yet been filled. In such cases, the monitor routine can request a new cycle be started so that another job may be selected to run. To do so, the monitor routine passes control to WSCHED. WSCHED, like USCHED, sets up the return address, saves the ACs, and passes control to RSCHED. Later, following an interrupt, the program will be rescheduled and will continue at the point where the UUO routine requested a new cycle. Some functions in the overall cycle will not be performed if the clock has not ticked.

2.5 Saving the PC word

When the software clock interrupt occurs, the first instruction executed is an XPCW. An XPCW instruction saves the PC double word for the user program at location CK*n*CHL. If the interrupted program was in user mode, the contents of CK*n*CHL are copied into .CPPC in the CDB. If the job is not chosen to run next, its PC is copied from .CPPC in the CDB to USRPC in the user's UPT. The job's PC word is preserved in USRPC until the job is chosen to run again. When the job is chosen to run again, the opposite process occurs. The contents of USRPC are copied into .CPPC during context switching, and control is given to the user program with the instruction: XJEN .CPPC. After execution of the XJEN, the PC and processor flags have the same value that they had when the program was interrupted.

When the program is stopped by software action, at either WSCHED or USCHED rather than by interrupt, the stored PC word is set up to continue the program within the routine that stopped it. Both WSCHED and USCHED are called with a PUSHJ, which leaves the PC word on the push down list. This PC word is POP'ed into .CPPC and, from that point on, is handled by the same code executed on a clock interrupt.

When control is returned to the user program, it isn't neccesary to be concerned about the manner in which the program was stopped. When the PC word is restored from the contents of USRPC, the program continues running in the correct state. If it was interrupted in user mode, it continues with the next instruction that would have been executed. If the program was stopped by a PUSHJ to WSCHED or USCHED, it continues with the next instruction after the PUSHJ. Therefore, from the point of view of a single program, the PUSHJ appears to behave like a normal subroutine call. In reality, however, the "subroutine" is the execution of an assortment of other unrelated tasks or programs.

Chapter 3
Command Processor

The command processor, COMCON, is called once a tick from the monitor cycle to process commands from a job. COMCON decides upon the appropriate function to perform, and either runs a program or handles the request itself. This chapter explains how COMCON pre-processes commands and dispatches to various routines, how error checking is done, and how users add new commands.

3.1 The Command Processor

The command processor provides a way for users to request the monitor's services. When the user types a command at a terminal, each character entered causes an interrupt to the system. The terminal interrupt routine reads in each character and stores it in an input buffer for that terminal (see Chapter 12. The interrupt routine tests each character to determine if it is a break character, which indicates the end of a character string. When a break character is received, the interrupt routine recognizes it as the terminator of a command. At that time, assuming the terminal line is at command level, a bit in the table CMDMAP is set, indicating that there is a command from that line waiting to be processed. In addition, COMCNT is incremented, indicating that there is a command pending.

On the next clock interrupt, the overall control routine checks COMCNT. If a non-zero value is found, the control routine dispatches to the command processor. The command processor calls the routine to identify a line with a command ready.

The command processor only processes a single command on each call. If several commands are presented by users during the same monitor cycle, the command processor is called on successive clock ticks until all the commands are processed. This policy ensures that the command processor does not take too much time away from any single clock tick.

3.2 Terminal Considerations

There is an input buffer and output buffer within the monitor for each terminal line. All terminal I/O is placed into and taken from these buffers. Each terminal line has a Line Data Block (LDB) containing the buffer pointers and additional information about the line.

A bit in the LDB, LDLCOM, indicates whether the line is being used for command input or for user program input. Initially, a line is at command level and any characters typed on the terminal are interpreted as a command. If the user gives a command to run a program, the line is normally switched to user level and any characters in the input buffer are available as terminal input for the program. When the program exits or is stopped, the line goes back to command level. It is possible to start a program but leave the line at command level by means of a CSTART or CCONT command. This allows the user to give certain simple commands while his program runs.

When processing commands, characters are always extracted directly from the terminal input buffer. Such considerations as file names and logical device names do not apply to the command processor. For example, assigning TTY as a logical device name for the card reader would **not** cause the command processor to take commands from the card reader.

3.3 The Dispatch Process

When the command processor identifies the line having a command, the first six characters (up to a blank, switch, or break character) are extracted from that line's input buffer as the command. Any additional characters in the input buffer may be taken as arguments by the routine that handles the specific command, but are not considered in the dispatch process.

3.3.1 Command Tables

When processing a command, the monitor looks at three tables in an attempt to find a match. The three tables are: the standard system-wide command table, COMTAB; the customer-defined system-wide table, CSTTAB; and the user-defined table (one for each job), pointed to by location .PDCMN in the process data block (PDB). COMTAB is distributed as a part of COMMON. The contents of CSTTAB are defined as a part of the MONGEN dialog. Users define the commands in the user-specific tables by using the DECLARE monitor command or the CMAND. UUO.

3.3.2 Table Format

Each of the three command tables (system, customer, and user) consists of three parallel subtables: command names, pre- and post-dispatch processing control bits, and routine addresses.

The command names table contains the SIXBIT string definition of the command. One command name is stored per word, so commands can be no longer than six characters. This table is stored beginning at COMTAB or CSTTAB for system and customer tables, and at the address pointed to by word .PDCMN in the PDB for the user table.

The pre- and post-dispatching processing control bits table has two different organizations, depending on the type of command table it is a part of. The system and customer tables, which start at UNQTAB and UNQTBC respectively, each contain one word per command entry. (Full descriptions of all of the control bits are provided in Section 3.6.) The user

control-bits table begins at the address pointed to by the left half of .PDUNQ in the PDB and consists of a string of six-bit bytes. There is one string of six-bit bytes per command, to specify the command's uniqueness.

The routine address table contains the addresses of the routines that process each command. These routines may reside in COMCON, or elsewhere in the monitor. The system table begins at DISP, the customer table begins at DISPC, and the user table begins at the address pointed to by the right half of .PDUNQ in the PDB.

3.3.3 Dispatching

The dispatch routine gets the first six characters from the terminal input buffer, converts them to SIXBIT, and performs a table look up on the command; first in the user command table, then in CSTTAB, and, if not yet found, in COMTAB. If it finds an entry that exactly matches the command given by the user, that entry identifies the command. If no entry matches exactly, a subsequent search checks if one and only one entry matches the characters the user typed (that is, if the user typed an unambiguous abbreviation of a command). If exactly one entry matches the user's command, for as many characters as he typed, that entry identifies the command. If the command given matches an entry in both the user table and CSTTAB or COMTAB, the user table entry is used. If a match is found, the address of the routine to process it is picked up from the parallel entry in the appropriate dispatch address table. Similarly, if the user's command matches entries in both the customer and system command tables, the customer table entry takes precedence. This allows customers to redefine system-wide commands without having to edit COMMON.

3.4 Long Routines

Since the command processor operates as a part of the overall monitor cycle, the time spent processing commands reduces the time spent in the next user program. Therefore, the command routine must be written to run to completion quickly. Many commands, however, require more time than the monitor can afford to take out of the overall cycle. These commands are handled by setting up a monitor routine to run in the user's time. Such a routine appears very similar to a UUO except that it does not return to the user program upon completion. Since the processing of these commands requires the use of the user's accumulators and PC, they are not accepted while the user has a program running. The SAVE and RUN commands are of this type.

Many commands that appear to be handled by the monitor actually run system programs for the user and pass the command arguments on to the program; all COMPIL-class commands are of this type. Giving such a command is equivalent to running the system program with an R command and giving the appropriate commands to the program. In the command dispatch process these commands are all equivalent to R.

3.5 Forced Commands

Certain monitor routines sometimes require that a command be executed for a given line. For example, if a user at a dataset hangs up (or the telephone circuit is broken) the job must be DETACHed, and the forced command mechanism enables such an action. A monitor routine that wants to force command execution for a specific line can, using FRCUUO, deposit a forced command index into the line's LDB and set the forced command bit, LDBCMF. The command processor does not look at the line's input buffer on a forced command, but uses the forced command index as a pointer into a table of forced commands, TTFCOM. The SIXBIT command from TTFCOM is processed exactly as a normal command. Certain control characters (CTRL/T and CTRL/C) cause similar actions to be performed.

3.6 Predispatch Bits

Several bits in UNQTAB specify conditions to check and functions to perform before dispatching to the command routine. These predispatch bits are listed and described below.

Table 3–1: Predispatch Bits

Bits	Definition
NOCORE	Specifies whether the command is legal for a job with no virtual core allocated. This bit does not cause the command to be delayed. The command is simply legal or illegal according to whether the job has any core allocated.
NOJOBN	Specifies whether the command requires a job number. If the terminal on which the command was typed is not attached to a job and this bit is not set, a job number is assigned to the terminal.
NOLOGIN	Specifies whether the job must be logged in before the command is legal. Most commands have this bit set to zero. Exceptions: LOGIN and INITIA.
NOACT	Delays a command until all I/O completes. The job is already in core because there is active I/O. CMWB is set and the job is requeued to the CMWQ. The use of the CMWB and CMWQ is two-fold: if the job is swapped out, it is swapped in, and the job is not scheduled to run.
NORUN	Specifies whether the command may be performed for a job that has a program running. All commands that result in setting up a routine to run as the user job have this bit set. Typing a command while the job has a program running results in an error message, "Please type ^C first".
INCORE	Indicates that the job must be in physical memory if it has any memory allocated. This bit does not make the command illegal for a job that has no memory allocated. If a command with this bit set is given while the job is swapped out (or being swapped) the command must be delayed, and the job is put into the Command Wait Queue and assigned a very high priority for swap in.
NXONLY	Indicates that the command is illegal for a job running an "Execute-only" program. Execute-only is a property that may be given to a executable image file to allow users to run it as a program, but not look at it. This allows unprivileged users to execute proprietary programs.

Table 3–1 (Cont.): Predispatch Bits

Bits	Definition
NBATCH	Specifies that the command is not legal from a batch program. Batch programs are treated almost identically to normal timesharing programs in most respects. However, some commands, such as DETACH, are not permitted.
NORCMP	Specifies that the command is allowed while logged out from a remote terminal.
CUSTMR	This bit is reserved for installations to use for their own purposes and has no function in the standard monitor.

3.7 Postdispatch Bits

The remaining bits in UNQTAB specify actions to be performed upon return from the command routine.

Table 3–2: Postdispatch Bits

Bits	Definition
NOINCK	Indicates that a job is not initialized as a result of this command. It is set initially in DISP for certain commands. If it is set, the accumulator in which the DISP are held in case of an error in a command which otherwise would have initialized a job.
NOCRLF, NOPER	Specifies whether a carriage return, line feed, and a period should be typed upon completion of a command. NOPER is set for any command on which the monitor is not ready to accept another command immediately after completion of the command routine. This includes all commands that set up a monitor routine to run as the user job.
TTY*xxx*	Set the job runable with different conditions. TTYRNU sets the job runable and switches the terminal to user level. Example: RUN. TTYRNC sets the job runable but leaves the terminal at command level. Both of these bits cause a job to be put into a run queue by the scheduler. TTYRNW sets the job runable, but checks to see if it was stopped in terminal I/O wait. If so, it is put back into the terminal I/O wait queue. Example: CONTIN. Not more than one of these three bits are set for any one command. None of them are set for a command that does not make the job runable.
CMWRQ	The command wait requeue bit (CMWRQ) gets the job out of the command wait queue. This bit is set in DISP for any command that might cause the job to be put into the command wait queue and does not cause the job to be requeued. The bit is cleared if the job is not requeued to command wait. Upon completion of the command routine, if this bit is set, the job is marked to be put back into its former queue.

Table 3–2 (Cont.): Postdispatch Bits

Bits	Definition
NOMESS	Command routines that output to the user's terminal do not start the terminal, but simply deposit characters in the output buffer. One of the functions performed upon return from the command routine is to start output on the line in case there are characters to be typed. The NOMESS bit suppresses this action for the commands that never type a message.
ERRFLG	Set by command routines to signify an error while processing the command.
APPFLG	Creates an alternate context ("auto push"), internally called SACFLG.
NOFLM	Do not force left margin.
PSTCST	This bit is reserved for installations to use for their own purposes and has no function in the standard monitor.

Chapter 4

Memory Management

KL10 memory is organized into pages of 512_{10} words. Pages are the basic unit of allocation of memory for both monitor and users. The memory management modules allocate memory by manipulating page maps and data structures within the monitor. Although system programmers rarely have cause to change the memory management routines or data base, a knowledge of how they work is necessary because many routines and subsystems rely on it.

4.1 Introduction

Memory management encompasses both physical pages (memory) and virtual pages (disk space used for swapping and paging). Management can mean simple bookkeeping or high level decisions about swapping and the running of programs. This chapter discusses the bookkeeping functions performed by the monitor modules CORE1 and APRSER.

Note

In TOPS-10 Version 7.04, low segments and sharable high segments are managed in much the same way. Therefore, the word "segment" is used to refer to either a low segment or a sharable high segment, unless the context clearly indicates otherwise.

The modules CORE1 and APRSER can be accessed three ways:

1. By the command decoder, COMCON, when the CORE, RUN, GET, or MERGE monitor commands are issued.
2. By the UUO handler, UUOCON, when a CORE, PAGE., RUN, GETSEG, SEGOP., or MERGE. monitor calls are issued.
3. By the swapper.

4.2 Allocation and Assignment

Memory management is accomplished by allocation and assignment. The functions allocation and assignment are defined as follows:

Allocation Management of virtual memory and swapping space.

Assignment Management of physical memory or core.

4.3 Memory Management Data Base

The heart of the memory management data base consists of the tables PAGTAB, PT2TAB, and MEMTAB. These tables allow the monitor to keep track of physical memory and consequently, consist of one word per page of physical memory, indexed by page number.

Conceptually, PAGTAB and PT2TAB are really one table consisting of sets of doubly linked lists of pages. Physically, PAGTAB contains the forward links, and PT2TAB contains the backward links, each being stored in the right halfword of their respective tables. The left halfword of each of these tables is used for various status bits and (in the case of PT2TAB) fields.

The MEMTAB table also consists of one word per page of physical memory indexed by page number. Its primary use is to correlate a physical page in memory with any disk copy that may exist for the page; thus, the right hand 22_{10} bits of this table are reserved for any such disk addresses. The remaining bits are used for status bits and fields. The exact interpretation of the contents of the fields in MEMTAB vary with the status of the page in question.

On the KL10, PAGTAB, PT2TAB, and MEMTAB reside in executive address space in the section designated by the variable MS.MEM defined in S (this is currently section 3).

As was stated above, the tables PAGTAB and PT2TAB consist of sets of doubly-linked lists of pages. The headers to these linked lists allow the monitor to access the lists according to the function or status of the physical page in memory.

A PAGTAB or PT2TAB linked list can be headed by any of the following:

Headers	Function
PAGPTR	Heads the list of all free pages.
JBTUPM	Points to a two-page chain consisting of the job's UPT and section 0 page map (also known as the job's UPM). Indexed by job number.

Headers	Function
JBT*xxx*	JBTSSA, JBTAD2, and JBTHSA. Points to the physical pages in use by the segment; indexed by job or segment number. These three labels all refer to the same table, and can be used variably, depending on whether the context is relevant to a low segment only (JBTAD2), a high segment only (JBTHSA), or either segment (JBTSSA). If a low segment, JBTAD2 points to virtual page 0 (that is, the JOBDAT page) of the corresponding job. In TOPS-10 Version 7.04, there are no further restrictions of the order of the linked list of pages for the low segment. However, the pages are often linked in ascending virtual address order if the job has not changed its memory allocation (this includes implicit or explicit allocation of section maps for a multi-section image) since the time it was last swapped in or created. For sharable high segments, the pages exist in the list in ascending virtual address order, but programs written to be monitor independent should not be dependent upon this condition. **Note:** It is frequently convenient for the monitor to handle per-process pages as parts of the user's low segment address space. Since the maximum virtual page number a user's image can possess on the KL10 is 37777_8, virtual page numbers are assigned to these pages starting at user virtual page number 40000_8. The assignment of virtual page numbers is as follows: 1. Funny pages. 2. Extended section working set bit map page (.WSBNZ). 3. Section 0 page map (addressed as .UPMAP). 4. UPT. 5. Section maps for user sections 1 through 37_8.
PAGINQ	Heads the list of pages on the monitor's "IN" paging queue.
PAGSNQ	Heads the list of pages on the monitor's "slow" "IN" paging queue.
PAGIPQ	Heads the list of pages on the monitor's "IN PROGRESS" paging queue.
PAGOUQ	Heads the list of pages on the monitor's "OUT" paging queue.
PAGTAB(0)	Heads the list of all pages owned by the monitor. Status bit MONTRB is set for these pages in PAGTAB.
.USTMU	Heads the list of pages that are affected by a job's PAGE. UUO (including on behalf of the job by MONPFH) and that exist on one of the paging queues at the time the UUO is executed. Status bit P2.TRN is set in PT2TAB for these pages.
LOKPTR	Points to pages that are reserved until a LOCK UUO completes for some job or segment. Pages on this list are free in that they are not allocated to any function; they are kept separate from the normal free list and are not allocated until the job issuing the LOCK UUO uses them. (The monitor has previously computed that these are the pages into which the issuing job is moved when they are all available. Because this operation may require swapping, it can take a significant amount of time to complete.)
BIGHOL	Number of unassigned physical memory pages, the same as the number of links in the linked list pointed to by PAGPTR.

Headers	Function
CORTAL	Number of free pages, BIGHOL, plus the number of idle and dormant high segment pages.
LOKTAL	Number of pages on the queue headed by LOKPTR.
PAGINC	Number of pages on the "IN" queue.
PAGSNC	Number of pages on the "SN" queue.
PAGIPC	Number of pages on the "IP" queue.
PAGOUC	Number of pages on the "OUT" queue.
VIRTAL	Total number of free pages in the swapping space.
JBTADR	Address and length for each segment in core for each job. The address values are 773000_8 (symbol .JDAT) for low segments, 774000_8 (symbol .VJDT) for high segments, unless the segment is locked physically contiguous.
SECTAB	An offset into the UPT which points to the extended section page maps of the user.
MS.MAP	Method by which the monitor addresses all of the user's section maps for examination and manipulation on the KL10. The user's section 0 map can be referenced either by .UPMAP or by this method on the KL10. The map pointer fetching routines are conditionally assembled to return pointers based on MS.MAP for monitors that support extended addressing, and ones base on .UPMAP for monitors that do not. MS.MAP currently resides as monitor virtual section 37_8.

4.4 Memory Management Algorithms

When a request for memory is made, four steps must be accomplished in order to assign the memory:

1. Dispatch to core handling routine.
2. Pre-processing and argument checking.
3. Allocation.
4. Assignment.

The swapper, discussed in Chapter 7, goes straight to CORGET to allocate and assign memory.

4.5 Core Allocation

When a CORE command is issued by a user, monitor control passes to the CORE routine in COMCON, then to CORE0 in the module CORE1, where pre-processing is performed. Next, the CORE1 routine is called. When done, CORE1 falls into CORE1A (in APRSER) for assignment. Through the use of the CORE command, a user can increase or decrease low segment memory allocation (subject to limit restrictions). Although the user can give an argument of zero to the CORE command, he cannot deallocate all of his core. This is because the Job Data Area (JDA), which exists for all jobs, resides in the user's page 0 address space. A job can have zero pages only when it is non-existent, swapped out, or is the victim of a JOB STOPCD or serious error (such as a swap-read error or memory-parity error). In the case of

serious errors, the routine ZAPUSR (or one of its alternate entry points) is used to deallocate the core. The CORE command can be used to expand from zero pages, but this is useful only in the case of serious errors.

4.5.1 CORE Command

The CORE command (Examples in Figure 4–1) accomplishes its pre-processing at CORE0 before going to CORE1 to allocate the core. CORE0 can also be entered from COMCON when minimal core is assigned for for the KJOB, RUN, ASSIGN, and DEASSIGN commands.

Figure 4–1: CORE Command Examples

```
CORE <returns current allocation>
      CORE 0
      CORE n
      CORE nK
      CORE nP

Example of allocation returned:

     Page number        Page status     Data on page

     0-1                RD WR EX        Private page(s)
     400-411            RD EX SH        DSKA:PIP.EXE[1,4]
     700-712            RD WR EX        Private page(s)

     Total of 27 pages
```

At CORE0, the monitor checks that the job is in core with no active I/O before going to CORE1. If the job is swapped out and the user gives a CORE command with an argument, the location IMGIN in JBTIMI is changed to reflect the amount of core desired by the user. This amount is allocated and assigned when the job is swapped in.

4.5.2 CORE UUO

The CORE monitor call allows the user to allocate and deallocate pages from a program. The restrictions are the same as for the CORE command but, because this is a UUO, it cannot be used to expand from zero pages.

After UUO setup by UUOCON, control passes to the CORUUO routine (in CORE1). CORUUO calls CHGCOR to check the arguments, and then calls the same routines as executed by the CORE command: CORE1, VIRCHK, and CORE1A. After CHGCOR is called, CORUUO calls CORBND to calculate the current core size and returns it to the user.

Whenever a CORE UUO is issued, control is passed to the routine CORUUO in CORE1. If the UUO has a non-zero argument, the job size is altered and the routine CHGCOR is called to allocate and assign core. If the UUO has a zero argument, no change is made to the user program's allocation. When returning to the user's program, the routine CORBND is called to return the amount of the user's core currently assigned.

If CHGCOR is called from CORUUO or UUOCON (to set up buffer rings), it waits for all I/O to finish so that no data is lost. Otherwise, incoming data has no place to go if the pages are deallocated (the job is decreasing in size) or if the job is swapped out (JXPN is increasing).

If the job is not locked, the CORE1 routine is called to allocate and assign core, followed by a call to UCORHI to assign high segment core, if needed. If core cannot be assigned, the job is marked as expanding (JXPN) for swap out and the scheduler is called to choose another job to run (WSCHED). The job is eventually swapped in with the correct amount of core.

Errors occur if the job is locked, the expansion request exceeds CORMAX or currently assigned user limits, or if I/O is active.

4.5.3 CORE1 Routine

The CORE1 routine tests for certain conditions before allocating core. Control reaches routine CORE1 in module CORE1 from both the CORE UUO and the CORE command. The conditions tested for are:

1. The argument is not negative.
2. The Job is not locked in core.
3. The Job is not trying to expand into high segment.
4. The Job is not trying to exceed virtual limits.
5. Allocating core to the Job would not exceed VIRTAL.
6. The Job is not trying to exceed CORMAX.

If all conditions are met successfully, the pages are allocated and assigned using VIRCHK (except as noted below) and the return is made to UUOCON or COMCON.

If the low segment is expanding from zero or if a sharable high segment is changing, control returns to CORE1 for allocation and eventually to CORE1A for assignment.

4.6 Core Assignment

After core allocation is accomplished (in routine CORE1 or VIRCHK), core can be assigned. The two routines responsible for core assignment are VIRCHK and CORE1A. VIRCHK assigns core if it requested by a CORE command or a CORE UUO, except when the core is requested for a sharable high segment or a low segment increasing from zero. In these cases, CORE1A assigns the core. CORE1A also assigns core for the swapper.

To better illustrate this division of labor, consider the following five cases:

If...	Then...
All of a job's core is deassigned.	The monitor destroys the UPT and UPM.
The job increases in size and is not going virtual.	VIRCHK simulates a PAGE. UUO by calling the routine UPAGE. in VMSER. If there is enough physical core available, UPAGE. calls the routine ADPAGS in CORE1. ADPAGS updates PAGTAB and PT2TAB. If physical core is insufficient, a dispatch is performed to routine FIXDSK where the job is marked as expanding (JXPN set) and the scheduler is called to choose another job to run (WSCHED).

If...	Then...
A job is increasing in size and is virtual or will be going virtual.	The S (software) bit is set in the corresponding page map entries (VIRCHK).
A job starts with zero core and requests an amount of core that is not available.	Control passes to routine CORGT7 and the job is marked as expanding (CORE1A).
A job starts with zero core and requests an amount of core that is available.	The monitor builds the UPT and UPM and gets the core. Once CORE1A is entered, dispatch is made to CORGT0. There, the pages are assigned in the low segment, including one for the UPT and one for the UPM. All pages are zeroed and the UPT is built. See Section 4.7 for more information on the steps necessary to complete this task. Finally, JBTADR, JBTREL, and the UPT are updated.

4.7 Creating a UPT

When a user job expands from zero core, the job's UPT must be created. Even though a page is assigned for the UPT, it cannot be accessed because all executive mapping for Executive Virtual Address (EVA) 772000_8 (.UPMP) must go through the nonexistent UPT. An alternative mapping scheme is used to access the UPT until it is completely initialized.

The UPT, besides pointing to the UPM, must point to itself so that it can be accessed. This is done through location .UPMP + 772_8. Figure 4–2, Creating a UPT, illustrates mapping for executive virtual pages 771_8 through 774_8.

Figure 4–2: Creating a UPT

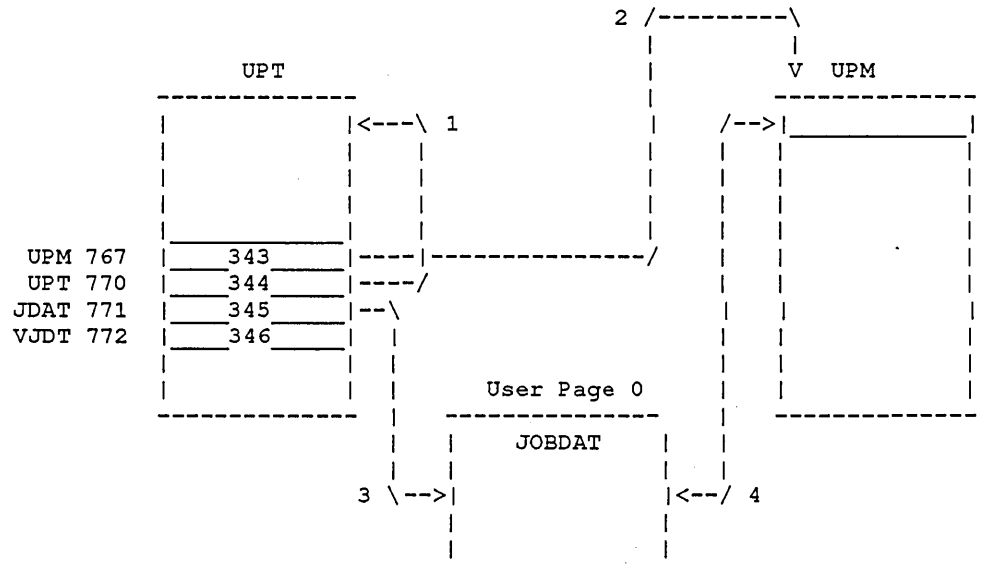


Figure 4–2 Cont'd. on next page

Figure 4–2 (Cont.): Creating a UPT

```
1.  Self Pointer (.UPMP + .UMUPT.)
2.  Current Section Map Pointer (.UPMP + .UMUPM)
3.  Job Data Area (.UPMP + .UMJDT)
4.  User's Mapping for Page 0 (Job Data Area)
```

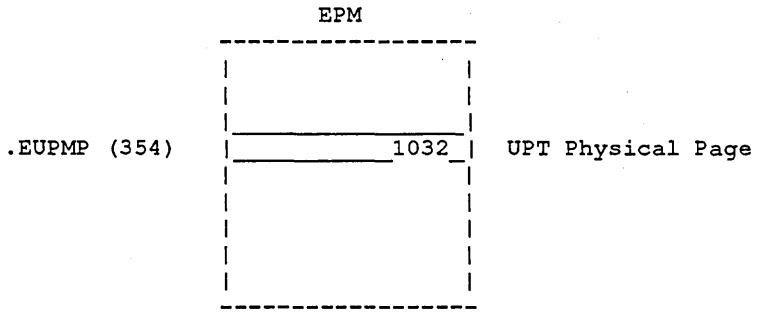
Location 773_8 of the UPT points to the first logical page of the user's program, and location 771_8 points to the UPM.

If a page is assigned to create a UPT and its 13-bit physical page number is known, the page can be accessed to create the necessary links by using the EPT and the EPM for section 0 of executive space.

The EPT always exists and is always pointed to by the EBR. The EPM for section 0, pointed to by the section map pointer in EPT 540 (SECTAB), has several spare mapping slots in the EPM. One slot at EPM offset 736_8 (.EUPMP) is allocated just for the purpose of creating a UPT.

For example, if physical page 1032_8 is allocated for the UPT, the number 1032_8 is stored in the EPT mapping location for .EUPMP (Executive virtual address 736000_8), one of the "spare" slots. This page is used as a temporary scratch pad to build UPTs. See Figure 4–3, Storing the Page in the EPT.

Figure 4–3: Storing the Page in the EPT

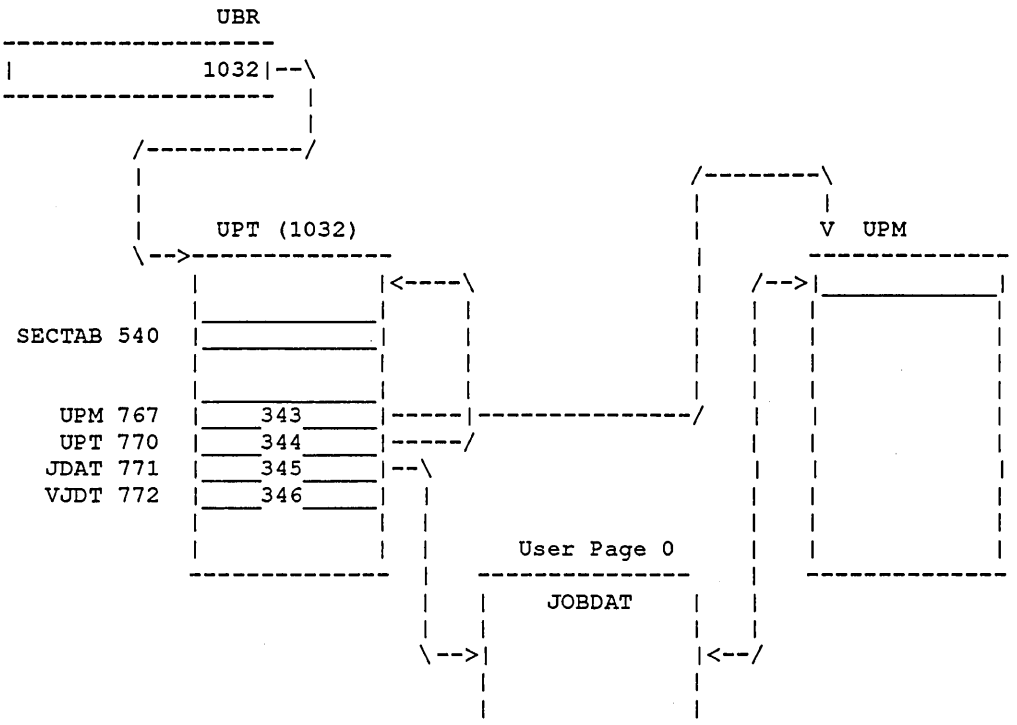


Executive virtual address 736000_8 now maps through the EPM to physical page 1032000_8, the new UPT. The UPT link to itself is accomplished by these instructions:

```
MOVEM   T1,.EUPMP+.UMUPT
TLO     T1,(PM.CSH)
MOVEM   T1,.EUPMP+.UMUUP
```

Since the UPT can now stand by itself, 1032_8 is placed in the UBR, thus making the UPT addressable at executive virtual address 772000_8. The remainder of initialization can be performed. See Figure 4–4.

Figure 4–4: Final UPT



4.8 Virtual Memory

Virtual memory was first implemented in TOPS-10 with the release of Version 6.01. The monitor module VMSER was added to handle references to logical addresses that are not in memory, as well as to handle the PAGE. UUO.

References to addresses in pages with the A bits cleared produce a page fault. The page fault condition, in most cases, traps to a routine called a Page Fault Handler (PFH). The PFH decides which pages to bring into core so that a job can continue to run. The system default PFH is resident in the monitor as of TOPS-10 Version 7.03.

Previous monitors used a default PFH located on SYS:PFH.EXE, and users can write and use their own, if so desired. In all cases, the user bears the burden of overhead for using virtual memory. For user-written page fault handlers and, in previous releases of the TOPS-10 monitor, the page fault handler resides in user space and is considered part of the user's program.

There are two sets of controls for virtual memory: one for administrators and one for the user.

Table 4–1: Administrative Controls

For each PPN:	
MPPL	Maximum Physical Page Limit. The maximum number of physical pages a user job may have in core at any one time.
MVPL	Maximum Virtual Page Limit. The maximum size that a program may reach including memory and what is stored on disk.

The user can also set limits for a specific job before or during program execution.

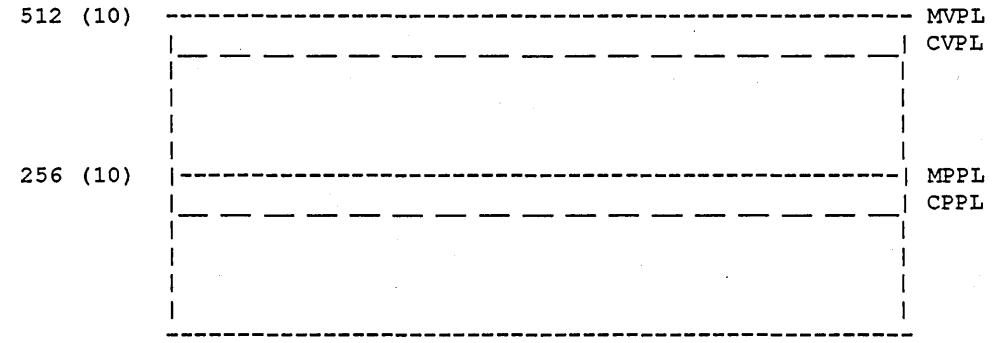
Table 4–2: User Controls

For each job:	
CPPL	Current Physical Page Limit. The current number of pages a user job may have in core at any one time.
CVPL	Current Virtual Page Limit. The current size that a program may reach including memory and what is stored on disk.

For example, if a user program's MPPL (or CPPL, if CPPL < MPPL) is 256 pages and MVPL is 512 pages, as long as the program size is less than the MPPL (256 pages), the program is not "virtual". When the program is bigger than 256 pages, the program "goes" virtual and not all pages may be in memory at the same time. For users without virtual memory, the MPPL and MVPL are identical. See Figure 4–5.

The maximum parameters are set by the administrator using REACT and are stored in the Process Data Block (PDB) during the execution of a job.

Figure 4–5: Virtual Memory



Instead of setting a CPPL, the user can set a physical guideline by issuing the monitor command SET PHYSICAL GUIDELINE *n*P. The guideline can be exceeded but the job's size is brought down to the guideline at each virtual time trap.

With regard to these limits, a job may run virtual in response to a RUN or GET command, a CORE UUO, or a PAGE. UUO.

4.8.1 Virtual Memory Data Base

When a job uses virtual memory, much more accounting is required to keep track of the pages. Because the information is unique to the job, the data base for virtual memory is kept in the UPT.

Table 4–3: Virtual Memory Data Base Elements

Element	Contents
WSBTAB	The sub-table of bits within the UPT, starting at location 440_8. This is the working set table and contains one bit per user page in section 0. If a bit is 1, the corresponding page is in core.
.WSBNZ	The working set table for sections that are greater than 0. It is for users who are running in extended sections and is mapped through the UPT at offset 770_8 (.UMWSB). If a bit is 1, the corresponding page is in core.
PM.AAB	The accessible bit in the page map entry that determines if a page is currently accessible, that is, exists in core. If it does, presumably that page's bit is on in WSBTAB.
UPM	The structure that contains one word entries for each possible page in the user's virtual address space. These entries provide the physical or disk address of the page and bits that indicate if the page is accessible, public, writable, software (that is, allocated but zero), or cacheable. Each UPM is one page long and contains entries for a single section, 512_{10} pages. If a user is using extended addressing, there is an additional UPM allocated for each section. The current section's UPM is mapped through the UPT at offset 771_8 (.UMUPM) and is found at 771000_8 (.UPMAP).

4.8.2 PAGE. UUO

The PAGE. monitor call allows a user to manipulate pages and the data contained in them. PFHs use the PAGE. UUO for all page manipulations.

The calling sequence is:

```
        MOVE    ac,[XWD fcncode,addr]
        PAGE.   ac,
          error return
        normal return
        . . .
addr:   len
        first argument
        . . .
        last argument
```

Where:	Is:
fcncode	One of the function codes described below.
addr	The address of the argument list.

Where:	Is:
len	The number of words that follow in the argument list. The value of *len* must be greater than 2 or the negative of a value greater than 2. If *len* is specified as a negative value, only one argument follows. That argument is the page number of the first page in a set, where the set contains that page plus the number of consecutive pages indicated by the value. For example, when *len* contains a negative value (such as -3), the argument (for example, page number 401), is the first of 3 consecutive pages (for this example, pages 401, 402 and 403), to be manipulated. The pages you can specify are restricted by the following attributes: • Page 0 cannot be paged out or destroyed. • If the high segment is sharable, it cannot be paged out. • If the page is a SPY page, it cannot be paged out. • If a page is locked in core, it cannot be paged out.
argument	The arguments (first through last) for the given function, usually page numbers of memory pages being manipulated.

4.8.2.1 The .PAGIO Function

The .PAGIO function swaps a page in or out. Pages swapped in are added to the working set; pages swapped out are moved to secondary storage.

Use one word in the argument list for each page to be swapped, or specify a negative list length to specify a set of consecutive pages. If you use more than one argument word, the page numbers must be in ascending order.

The argument word format is as follows:

```
XWD     flags,pageno
```

Where:	Is:
pageno	The number of the page to be swapped. In the range $0 - 511_{10}$ on a KS, or $0 - 16383_{10}$ on a KL.

Where:	Is:
flags	One or both of the following:

Bit	Symbol	Bit Set	Bit Clear
0	PA.GAF	Swap page out.	Swap page in.
1	PA.GSL	Swap to slow swapping space.	Swap to fast space.
2	PA.GDC	Suppresses error codes PAGCE%, PAGME%, PAGSC%, and PAGSM%.	

4.8.2.2 The .PAGCD Function

The .PAGCD function creates or destroys a specified page. Use one argument word for each page to be created or destroyed. If you use more than one word, the specified pages must be in ascending order.

The argument argument word format is as follows:

```
XWD     flags,pageno
```

Where:	Is:
pageno	The number of the page to be created or destroyed. This number is in the range 0-511 on a KS, or 0-16383 on a KL.
flags	One or both of the following:

Bit	Symbol	Bit Set	Bit Clear
0	PA.GAF	Destroy page.	Create page.
1	PA.GCD	Create page on disk.	Create page in working set.

4.8.2.3 The .PAGEM Function

The .PAGEM function moves or exchanges a page. The page is moved from one virtual address to another, or two pages exchange locations. A page cannot be moved to a location that is allocated to another page, and pages cannot be exchanged unless the source pages are allocated.

Use one argument word for each page to be moved or exchanged. If more than one argument word is used, the specified pages must be in ascending order.

The argument word format is as follows:

```
<flag>+<source>B17+<destination>B35
```

Where:	Is:
source	The page number of the page to be moved.
destination	The page number of the location to receive the page.
flag	The following flag can be set:

Bit	Symbol	Bit Set	Bit Clear
0	PA.GAF	Exchange page.	Move source page.

4.8.2.4 The .PAGAA Function

The .PAGAA function sets or clears the access-allowed bit for a page. The access-allowed bit can be changed for any page in the working set. If a page is accessed that has this bit off, a page fault occurs.

Use one argument word for each page whose access-allowed bit is to be changed. If you use more than one argument, the specified pages must be in ascending order.

The argument word format is as follows:

```
XWD     flags,pageno
```

Where:	Is:
pageno	The page number of the page whose bit is to be changed
flags	One or more of the following:

Bit	Symbol	Bit Set	Bit Clear
0	PA.GAF	Clear access-allowed.	Set access-allowed.
1	PA.GSA	Automatically sets access-allowed on page fault.	Dispatch to page handler on page fault.
2	PA.GDC	Ignores nonexistant page, suppresses error code PAGEM%.	

4.8.2.5 The .PAGWS Function

The .PAGWS function returns a bit map of those pages in the current working set. In the PAGE. call, the number of words that are to be returned are specified. There is one bit for each possible page: if a bit is set, the page associated with that bit is a part of the working set.

For example, Word 1 contains the bits associated with pages 0 through 35; Word 2 contains the bits associated with pages 36 through 71, and so on. The end of the bit map does not end on an integral word boundary, so the last word in the map is padded with zeroes. The bit map for another section begins on a new word.

4.8.2.6 The .PAGGA Function

The .PAGGA function returns a bit map indicating those pages that have their access-allowed bits set. This bit map has the same format as the one returned for function .PAGWS. If a bit in the map is set, the page associated with that bit is accessible. In the PAGE. monitor call, specify the number of words in the bit map that are to be returned.

4.8.2.7 The .PAGCA Function

The .PAGCA function determines the type of access allowed a given page. There is no argument block; instead, specify the function code in the left half of the *ac* (bits 0-17) and the page number in the right half of the *ac* (bits 18-35): [function,,page-number].

On a normal return, the monitor sets one or more of the following bits:

Bits	Symbol	Meaning
0	PA.GNE	Does not exist.
1	PA.GWR	Writable.
2	PA.GRD	Readable.
3	PA.GAA	Access allowed.
4	PA.GAZ	Allocated page, but zero.
5	PA.GCP	Page cannot be paged out.
6	PA.GPO	Page is paged out.
7	PA.GHI	Page is in high segment.
8	PA.GSH	Page is sharable.
9	PA.GSP	Page is SPYing (mapped onto running monitor).
10	PA.GLK	Page is locked in memory.
11	PA.GNC	Page is not cached (KL10 and KS10 only).
12	PA.GSN	Section does not exist.
13	PA.GVR	Page is virtual (SPY page).

Bits	Symbol	Meaning
14	PA.GIN	Page is in an indirect section, that is, a section mapped onto another section.
15		Reserved for use by DIGITAL.
16-20	PA.GSC	Section is independent, that is, a section that another section is mapped onto. PA.GIN must be set for PA.GSC to be set.
21		Reserved for use by DIGITAL.
22-35	PA.GPN	Page number of the SPY page that the specified user page is SPYing on.

4.8.2.8 The .PAGCH Function

The .PAGCH function changes the pages in a high segment, or creates a high segment from a contiguous collection of pages.

The argument block format is as follows:

```
addr:   EXP     number-of-words-that_follow
        EXP     number-of-pages-to-be-remapped
        EXP     starting-page-number
        EXP     destination-page-number
```

Where:	Is:
addr+3	Is an optional word of the argument block. If not specified, page 400 is assumed. This function waits for all I/O to stop before creating the high segment.
destination-page-number	On a normal return, the specified pages are REMAPped into the high segment, begining at this location.
starting-page-number **and** *number-of-pages-to-be-remapped*	The error return is taken if all of the pages specified by *starting-page-number* and *number-of-pages-to-be-remapped* do not exist, or if a page included in the list already exists in the program's address space. If the number of pages specified is negative, those pages are remapped from the low segment to the high segment, and appended to the existing high segment. A sharable high segment cannot be created or affected with this function code. If only one argument is given, the number of pages specified is deleted from the end of the high segment.

4.8.2.9 The .PAGCB Function

The .PAGCB function sets or clears the cache bit for the page. This flag is automatically set if PA.GAF is on. This function sets or clears the cache bit on a per-page basis (KL10 and KS10 only).

The argument word format is as follows:

Bit	Symbol	Bit Set	Bit Clear
0	PA.GAF	Cache bit is set in the corresponding entry in the job's page map.	Cache bit is cleared.
1	PA.GDC	Ignores nonexistent page, suppressing error code PAGME%.	
2-26		Reserved for Digital.	
27-35		The page number.	

If there is more than one argument word in the argument block, the page numbers specified in those words must be in ascending numeric order. The error return is taken if any of the following are true:

- The function or call is not implemented.
- A high segment page is specified in the argument list.
- The argument list is not set up properly.
- The job is not locked in core.

4.8.2.10 The .PAGSP Function

The .PAGSP function allows a program to map an arbitrary set of pages from memory or from the monitor's virtual address space into the program's address space. Use one argument word for each page to be mapped. If more than one argument word is used, pages must be specified in ascending order. This function requires that the calling job have PEEK privileges on all of core.

Theargument word is formatted as follows:

```
<flags>+<source>B17+<destination>B35
```

Where:	Is:
flags	One or more of the following:

Bit	Symbol	Bit Set	Bit Clear
0	PA.GAF	Remove page from user's addressing space.	Add monitor page to user's addressing space at specified page number.
1	PA.GDC	On a create, this bit overlays an already existing page.	On a delete, nonexistent page is ignored, and error code PAGME% is suppressed.

source	The page number of source page. If UU.PHY is set in the PAGE. monitor call itself, *source* is a physical page in memory. If UU.PHY is not set, *source* is a monitor virtual address mapped through the executive page map.
destination	The page number of the page to be mapped into the program's address space.

4.8.2.11 The .PAGSC Function

The .PAGSC function creates or destroys a specified section. Use one argument word for each section to be created or destroyed. For more than one word, the sections or arguments must be specified in ascending order.

The argument word format is as follows:

```
XWD <flag>+<source>B17+<destination>B35
```

Where:	Is:
flag	One of the following:

Bit	Symbol	Bit Set	Bit Clear
0	PA.GSF	Delete the section if it exists.	Create the section if it does not.
1	PA.GMS	Map the sections specified in PA.GSS and PA.GDS together.	
2	PA.GDC	Empty any existing section.	Ignore nonexistant section.

source	The section number of the source section (bits 4-17, PA.GSS).
destination	The section number of the destination section (bits 22-35, PA.GDS).

4.8.2.12 The .PAGBM Function

The .PAGBM function returns a bit map that indicates whether specified page accessibility attributes belong to a certain page. If the bit in the map is set on, the page has the specified attributes.

Each argument word is of the form:

```
addr:   EXP     count
        EXP     attribute-settings
        EXP     care-mask
        EXP     starting-page-number
```

Where:	Is:
count	The number of arguments.
attribute-settings	The word indicating the desired state of the given attribute. The page accessibility attribute bits are the same as those given for .PAGCA.
care-mask	The word specifying which bits of the *attribute-settings* word should be examined. Note that PA.GSC, the independent section bit, is checked only when PA.GIN is turned on in both .PAGCA and in the care mask in .PAGBM. Likewise, PA.GPN, the SPY page number, is checked only when PA.GSP is on in .PAGCA and the care mask in .PAGBM.
starting-page-no	The word specifying the page number of the page that is mapped to bit 0 of the mask.

4.8.2.13 The .PAGAL Function

The .PAGAL function determines the type of access allowed for a given page.

The argument block format is as follows:

```
EXP   count
EXP   starting-page
```

Where:	Is:
count	The number of arguments.
starting-page	The starting page of the area in which information is to be returned. The bits returned are the same as for .PAGCA.

On a normal return, the specified function has been performed; the AC is unchanged. On an error return one of the following error codes:

Table 4–4: PAGE. UUO Error Codes

Code	Symbol	Meaning
0	PAGUF%	Function not implemented.
1	PAGIA%	Illegal argument.
2	PAGIP%	Illegal page number.
3	PAGCE%	Page should not exist, but does.
4	PAGME%	Page should exist, but does not.
5	PAGMI%	Page should be in core, but is not.
6	PAGCI%	Page should not be in core, but is.
7	PAGSH%	Page is in sharable high segment.
10	PAGIO%	Paging I/O error.
11	PAGNS%	No swapping space available.
12	PAGLE%	Core limit exceeded.
13	PAGIL%	Function illegal if page locked.
14	PAGNX%	Cannot allocate zero page with virtual limit zero.
15	PAGNP%	Not enough privileges.
16	PAGSC%	Section should not exist, but does.
17	PAGSM%	Section should exist, but does not.
20	PAGIS%	Illegal section.

4.8.3 Internal Use of the PAGE. UUO

The monitor can perform certain PAGE. UUO functions on behalf of the user. Some of these functions are executed as part of a monitor job (RUN UUO and related functions) and are therefore executed as normal UUOs at UUO level, using the shadow ACs to pass arguments. However, some of the PAGE. UUO functions must be executed at clock level (VIRCHK, for example). Those functions (.PAGIO, .PAGCD, and certain functions of .PAGRM) are specially coded to be executable at clock level.

4.8.4 Page Fault Handlers

A Page Fault Handler (PFH) manages memory for individual jobs when a job exceeds its current physical page limit. The PFH tries to keep the number of pages in core below the limit and close to the guideline. To do this, pages must be paged in and out. The PFH decides what specific pages to swap in and out. A list of pages must be maintained according to some algorithm. In addition, the PFH can create pages (allocated but zero).

Until a program exceeds the CPPL, a PFH is not needed. But, when that limit is reached, the monitor looks at .JBPFH in the user's job data area. If that location is non-zero, the contents are treated as the address of the PFH and dispatch is made there. If zero, the monitor uses the internal PFH contained within the monitor.

Control is sent to the PFH for some types of page failures, a time trap or a potential page failure (user address check in UUO processing).

4.8.4.1 Page Failure

A page failure (or page fault) occurs for the following reasons:

1. Proprietary Violation. A concealed page is referenced.
2. Page refill failure. Hardware problem.
3. Address failure. Address break (KL10 only).
4. Illegal Indirect address.
5. Page table parity error. Hardware problem with pager.
6. Illegal Address. Section number greater than 37_8.
7. AR/ARX parity error.
8. Reference to a location whose UPM entry has a pointer type of zero.

Only the last case is handled by the PFH; the rest are handled by the monitor.

KL10 page faults trap using locations 500 through 503 of the current user's UPT. When the trap occurs, the page fail word is stored in location 500 of the UPT, and the processor flags and the current PC Word are stored in locations 500 and 502 with a new PC Word taken from location 503.

The new PC in the page fail word is the address of the SEILM routine in APRSER. SEILM sorts out the type of condition and dispatches to the correct location. If the normal condition is met (a page needing to be read in from disk, or just turning on the A bit) control passes to the routine USRFLT in VMSER.

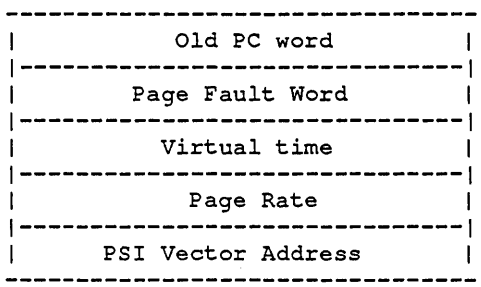
If the page fault is for a user program that is already virtual, the routine USRFLT does not return to SEILM.

USRFLT calls USRFLZ to do the work in processing the page fault. USRFLZ checks to see if the user specified his own page fault handler. If not, the Monitor Page Fault Handler (MONPFH) is called to take care of the fault. Otherwise, the monitor checks to see if the reference that caused the fault is for an extended section or if the current PC word is in an extended section. If so, the message "?Extended page fail, cannot be passed to user PFH" is printed on the user's terminal and the program is stopped.

If the pointer type specified in the UPM entry for the page is zero, the page fault is called an access fault. The monitor checks the software accessible bit (PM.AAB) to see if the page is in the working set. If this bit is set, the monitor changes the pointer type in the UPM and returns control to the user program, provided that the user's page fault handler specified the PA.GSA bit on the last function call to set page access (.PAGAA).

In the other cases, the accessible bit not set (the page is not in the working set), PA.GSA is not set, or a UUO Address check occured, the PFH argument block is filled in and control is transferred to the user PFH. The format of the PFH argument block is shown in Figure 4–6.

Figure 4–6: Page Fault Handler Argument Block



4.8.4.2 Potential Page Failure

When a monitor call is issued and a test must be made to see if the user's UUO argument block is in core, UUOCON calls the UUOCHK routine in VMSER to check if the argument block itself is accessable and to verify that any pages pointed to by the argument block are accessable.

UUOCHK calls GETWRD in DATMAN to access the user argument block, then it calls the XRNGE routine in VMSER to access all of the pages specified by the argument block. If access to any of these locations fails, UUOCHK dispatches to UUOFLT. UUOFLT releases any interlocks or resources that are being held, flags that the fault occurred in UUO code, and checks if the UUO was issued from user mode. If the UUO was issued from user mode, USRFL1 is called to run the PFH to get the page into core. If USRFL1 fails or the fault occured from exec mode, a UUO address error (check) occurs.

4.8.4.3 Virtual Time Trap

The purpose of a virtual time trap is to allow virtual jobs to page without thrashing.

For jobs that have gone virtual and have set a physical guideline rather than a limit, the monitor enforces a virtual time trap on a periodic basis. This is based on the job's runtime and is usually every half second.

By using a guideline, the user job is allowed to increase in size over the guideline (CPPL) without having to page out. At the time trap, the monitor initiates actions leading to the job's paging out either enough pages to get back to the guideline (if using SYS:PFH.EXE), or all pages that haven't been accessed since the last time fault. In addition, all accessible bits (A bits) are turned off. Thereafter, when the job runs, every reference to a different page causes a page fault. If the page is in the job's working set (WSBTAB bit set), the A bit is simply turned back on and control is returned to the user.

Thus, the A bit reflects usage history during the time trap interval. Selection of pages for page out, at the time trap, begins with unaccessed pages, pages whose A bit has not been turned on during the preceding interval.

The time trap counter is typically initialized or reset to .5 second. This counter, maintained in the PDB, is decremented every jiffy of runtime. When the counter reaches zero, control passes to TIMFLT in VMSER and then on to either the Monitor's Page Fault Handler (MONPFH) or the user's Page Fault Handler (PFH).

4.8.4.4 Page Fault Handler Conditions

The following is a list of conditions that a PFH must be able to handle:

1. Page not in memory. Examine the physical limit. If the physical limit has been reached, a page must be paged out before a new page is paged in. If the limit has not been reached, just page the new page in.
2. Page allocated but zero. Examine the core limit. If the limit has not been reached, create a new page. Otherwise, reduce the working set to allow room for the new page and create it.
3. A bit off, but page in memory. Turn the bit on. The PFH can have the monitor do this automatically by setting the PA.GSA bit on the call that turns off access allowed.
4. Time trap. The PFH can use this to periodically trim the working set.

4.8.5 System Page Fault Handler (SYS:PFH.EXE)

In monitors previous to Version 7.03, the system page fault handler (SYS:PFH.EXE) used a second chance first-in first-out (FIFO) algorithim to maintain a user program's working set. This means that the first page paged in is the first page to be paged out, unless there are other pages that have not been accessed.

The four access conditions are handled in the following manner:

1. Page not in memory. Examine the physical limit. If the physical limit has been reached, the first page in the FIFO list is paged out and the new page is paged in. If the limit has not been reached, the page is paged in.
2. Page allocated but zero. Examine the core limit. If the limit has not been reached, a page is created and stored on the FIFO list. If the limit has been reached, the first entry in the FIFO is paged out and the new page created.
3. A bit off, but page in memory. Turn the bit on.
4. Time trap. Examine physical guidelines. If the guideline has not been exceeded, the virtual time trap counter is reset to its initial value. If the guideline is exceeded, enough pages with the A bit off and the W bit on are paged out to bring the number of pages down to the guideline, the FIFO list is rebuilt, and the time trap counter is reset. Only pages that are in the low segment and are writable are paged out.

4.8.6 Monitor Page Fault Handler (MONPFH)

Unlike the system page fault handler, the monitor page fault handler (MONPFH) sequentially faults pages out in a ring fashion. That is, it starts at the top of the working set and pages out sucessive pages until the user program is under the limit. The guideline is not used; if a user specifies a guideline, MONPFH uses the maximum physical limit (MPPL). MONPFH remembers the last page that it paged out from the working set and restarts from there.

The four access conditions are handled by MONPFH in the following manner:

1. Page not in memory. The physical limit is examined and if reached, the page following the last page that was paged out is removed from the working set, then the requested page is paged in.
2. Page allocated but zero. The physical limit is checked, paging out a page if necessary, and a new page is created.
3. A bit off, but page in memory. The access allowed bit is turned on and the count of not accessable pages is decremented.
4. Time trap. When this happens, MONPFH pages out all pages that are in the working set but have their access allowed bits turned off last trap time. All pages that are in core have their access allowed bits turned off and the count of pages that are not accessable is adjusted to reflect this.

MONPFH does not page out the user's software interrupt vector, (PSI), JOBINT block, high segment, and DDT breakpoint pages. MONPFH also breaks down the large dump mode I/O lists into pieces small enough to keep the user's program within memory limits. MONPFH is also used by the monitor to migrate pages off of units that are being removed from the swapping space.

4.9 Paging Queues

In order to minimize paging I/O, the monitor does not immediately write pages paged out with the .PAGIO function of the PAGE. UUO. Instead, the pages are placed on a queue. Subsequently, the entire paging queue is written out using one I/O operation.

The advantages of using paging queues are as follows:

1. Monitor overhead is decreased, because the monitor is more efficient at a fewer large I/O operations as opposed to many smaller I/O operations (assuming the same number of pages are transfered).
2. If the job faults for the page before it is necessary for the monitor to write the queue, the overhead of the write operation is saved.
3. Even if the monitor writes the paging queue, the pages may not be immediately recycled, and the page does not have to be reread from the disk.

4.10 Internal Paging Queues

The monitor maintains four internal paging queues:

Queue	Function
IN	The queue on which pages are initially placed when removed from the job's working set by the PAGE. UUO. The page number exists in the page map entry for the job, but the hardware access bits for the entry are cleared and a software bit is set, indicating that the page is on the queue. The owner of the page is written in the MT.JOB field of MEMTAB for the physical page, and the virtual page number to which the page belongs is written in the corresponding P2.VPN field of PT2TAB.
SN	The queue reserved for those pages that are destined for slow swapping space (refer to the PA.GSL bit of the .PAGIO PAGE. UUO function). IPCF pages are also placed on this queue initially, in lieu of being paged out: in this case, the low order 15 bits of the address of the IPCF block that owns the page are written into P2.VPN, and the high order 3 bits are written in MT.JOB. MT.IPC is set to indicate an IPCF page. Conceptually, the IN and SN queues are identical, with exception of the swapping space designation.
IP	The in-progress queue is the queue to which the IN or SN queue pages are placed while they are being written to disk. When pages are moved from the IN or SN queues to the IP queue, swap space is allocated and the disk addresses written to the appropriate page maps or IPCF blocks using the information previously stored in MEMTAB and PT2TAB. The disk address is also written into MEMTAB.
OUT	The OUT queue receives the pages from the IP queue once the I/O completes successfully. The pages may now be recycled to the free list on a FIFO basis.

4.11 Sharable High Segments

In TOPS-10 Version 7.04, the flexibility of high segment assignment has been expanded to allow a user's core image to contain more than one sharable segment. In order to accomodate this feature, certain aspects of the monitor's handling of high segments were significantly altered:

1. High segments are now handled much more like low segments insofar as swapping and memory allocation are concerned. In particular, sharable high segments must have a map allocated to them that is pointed to through the use of the SPT and shared pointers in the individual user's core image maps.
2. The JBTSGN table was expanded to effectively allow more than one entry per job.

4.11.1 High Segment Map

The high segment map is the sharable high segment equivalent of a user section map. As such, it contains either page map pointers or disk addresses in the same form as a user page map. However, there are a number of significant differences between user section maps and high segment maps:

1. Whereas user maps are always allocated as full pages from the free page list, high segment maps are allocated as blocks of memory from monitor virtual address space in

the MS.MEM section, and are allocated like any other non-zero section free core, which is actually allocated in section 0 for non-extended monitors.

There are a number of restrictions that must be enforced in order for this approach to work. In particular, due to a combination of hardware restrictions and restrictions due to the overall method by which TOPS-10 uses the SPT, high segment page maps are not allowed to cross page boundaries and must be contiguous. The code that allocates and extends existing high segment maps is smart enough to create a map that meets these specifications. This restriction means that a single high segment can never be more that a 512_{10} pages and thus never larger than a single section.

2. High segment maps are never swapped.
3. High segment maps contain a data header that contains items of interest related to the high segment itself (and not any particular job using the high segment). These items are described as part of the definition of the .M2*xxx* symbols in S, by which they are referenced.
4. The high segment map is virtually addressable by the monitor as a normal part of the monitor's address space, because it is allocated from monitor free core. The physical page number of the map area of the high segment is pointed to by the high segment portion of the JBTUPM table. The monitor virtual address of the map (which contains the offset from the physical page number in the JBTUPM table) is pointed to by the JBTVAD table. This offset also must reside in the map pointers in each user's page map.
5. High segments are not allowed to be virtual. This means that the order of linked pages pointed to as belonging to the high segment is that of increasing virtual address.

4.11.2 Per-Job Database

The pointer to the per-job data base for high segments is JBTSGN. JBTSGN does not contain a high segment number, but the address of a block in section 0 free core. The free-core block (.HB*xxx* symbols defined in S) gives the segment number and type of segment information previously stored in JBTSGN entries (in the same format as previous monitors) in the .HBSGN entry.

There is one high segment data block for each high segment (including non-sharable segments) a job owns. The data blocks are linked together as a linked list pointed to by the job's JBTSGN entry. This list is zero-terminated. Because the high segment data block resides in section 0 of the monitor's address space, the high order 18_{10} bits of the JBTSGN entry for the job are used to contain bits that are an inclusive OR of certain high segment status bits (present in the .HBSGN word of the high segment block) for the job. This OR function allows certain data to be checked quickly for the job without chaining through all of its high segment data blocks.

4.12 Alternate Contexts

The implementation of alternate contexts under TOPS-10 Version 7.03 provides facilities to save and restore the state of a job. The alternate context features allow a running program to be halted and saved in order to perform some other task which would normally destroy the core image. The saved context can be restored and the original program continued. Temporary or permanent alternate contexts may be created through the use of monitor commands or monitor calls. The alternate context features is available to any program through the use of a monitor call.

By default, the alternate context service module (CTXSER) is always loaded. The loading of CTXSER is accomplished by setting the MONGEN parameter M.CTX to 1. If M.CTX is set to zero, CTXSER is not loaded. Not loading CTXSER has the following affect:

1. Monitor and user defined commands marked to perform an auto-save always destroys the user's core image.
2. The CONTEXT, PUSH, and POP commands yield the error message "?Job contexts not supported".
3. The CTX. UUO always takes the non-skip return, leaving the AC unchanged.
4. All UUOs that allow the inclusion of an optional context number in their arguments fail if the context number is non-zero.

4.12.1 Saving and Restoring the State of a Job

Alternate contexts are implemented by memorizing a job's current state and some terminal parameters. The job's core image is then swapped out to disk, an implicit RESET UUO is performed, and all user core is released. Once on disk, the context is considered idle. At this point, a user is free to run programs and use any monitor commands normally available.

When restoring the state of a job, an implicit RESET UUO is performed, all user core is released, and the saved parameters in the context block are restored. The next time the job wants to swap in, it has its original core image in-core again. All data residing in the per-process area (funny space) is saved and restored when switching contexts. No attempt is made to selectively save and restore portions of this data. Selected job tables, portions of the job's terminal DDB, and words within the PDB must be preserved over changes in contexts. Other items, while not absolutely necessary, do provide a friendlier interface and are desirable.

The following is a list of all items saved and restored. Those marked with an asterisk indicate data that is absolutely necessary to preserve the integrity of the monitor and job; all other items are optional.

1. Program run from physical SYS bit: JB.LSY from JBTLIM *
2. Monitor mode bit: LDLCOM from LDBDCH *
3. SAVCTX word in the PDB: right half of .PDSCX *
4. Break mask words LDBBKM and LDBBKB
5. PSI data base address JBTPIA
6. All IPCF-related words in the PDB

7. Enqueue block chain address: .PDEQJ
8. Selected words in the TTY DDB *
9. Job status word: JBTSTS *
10. Swapped out disk address: JBTSWP *
11. Swapped in image size: JBTIMI *
12. Swapped out image size: JBTIMO *
13. High segment number: JBTSGN *
14. Per-process (funny space) page count: JBTPDB *
15. Swapped out checksum: JBTCHK *
16. Program name: JBTNAM *
17. User PC: JBTPC
18. I/O wait DDB: JBTDDB
19. Program run data: .PDNAM, .PDSTR, .PDDIR, .PDSFD
20. Time of last reset: .PDSTM
21. Address to user defined commands: .PDCMN *
22. Address to UNQTAB for user defined commands: .PDUNQ *
23. Address of DECnet Session Control job block: .PDSJB

Additional items can easily be added to the list. The routines to save and restore job and terminal information are completely table driven. Provisions exist for calling subroutines to perform the save and restores where the complexity or amount of data to be moved requires special attention.

4.12.2 System Services

The ENQ/DEQ, IPCF, and PSI facilities have been modified to allow them to work with idle contexts. The concept of a Job/Context handle (JCH) was created to uniquely identify a job and one of its contexts. JCH storage requires 18 bits. A JCH with a zero context component is always translated into the JCH for the job's current context. The 18-bit UUO argument blocks for ENQ/DEQ, IPCF, and PSI are used for JCH storage. Thus, a program may target the UUO at a particular context for a job or leave the context number zero and the current context is the default. Idle contexts can hold ENQ/DEQ locks, have IPCF message queues maintained, and PSI interrupts posted.

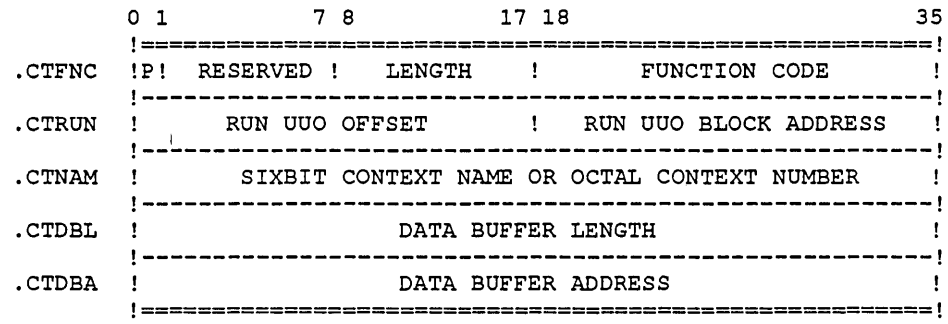
4.12.3 Program Interface

All context manipulation is done through the CTX. UUO (CALLI 215). The UUO argument block contains storage for a UUO function code, flags, RUN UUO block address, context name, and a data buffer length and block address. The argument block is never written by the monitor; it resides in a write protected page or a literal. All data is returned to the calling program in the UUO AC or the data buffer, if present. The UUO functions perform a superset of those available at monitor command level.

The calling sequence for using the CTX. UUO is:

```
XMOVEI  AC, addr
CTX.    AC,
 <error return>
<normal return>
```

The format of the CTX. UUO argument block is:



Words	Meaning

.CTFNC This word is used for UUO flags, block length, and function code. Currently, only one flag is defined.

Bit	Symbol	Meaning
0	CT.PHY	The physical-run bit. Function .CTSVR (save current context and run program) uses this bit to search for the physical device given in the RUN UUO block. Optionally, UU.PHY may be used with the same results.
1-8		Reserved for use by DIGITAL.
9-17	CT.LEN	Holds the length of the argument block.
18-35	CT.FNC	The function code. Negative functions are reserved for customers.

.CTRUN This is the RUN UUO word. It contains the run offset and block address that is normally put into the RUN UUO AC.

Bit	Symbol	Meaning
0-17	CT.OFS	The mask for the run offset.
18-35	CT.ADR	The mask for the address of the RUN UUO block.

.CTNAM This word is used to specify a context name when creating new contexts, namely functions .CTSVH, .CTSVR, and .CTSVT. For functions that perform some task on or for a particular context, this word contains either a context name or number.

Words	Meaning
.CTDBL	This word contains the data buffer length in words. The maximum value is 510_{10} words.
.CTDBA	This word contains the full 30-bit address of the data buffer. The IFIW bit is respected and causes a section local address reference relative to the section in which the CTX. UUO is executed.

4.12.4 CTX. UUO Functions

The CTX. UUO provides functions for creating, destroying, switching, and naming contexts, processing the data buffer, and manipulating quotas.

Function	What it does:
.CTSVH	Saves the current context and halts the job. This provides "PUSH" capabilities. The new context is inferior.
.CTSVR	Saves the current context and runs a program. This is equivalent to the auto-save and restore facilities available at monitor command level.
.CTSVT	Saves the current context and creates a new top level context. This function creates an adjacent context, not associated with the current context or PUSH chain, if one exists.
.CTSVS	Saves the current context and switches to another, already existing context.
.CTSVD	Saves the current context, deletes the target context, and switches back to the original one. In order to delete a context, a has to have that context as its current one. Hence the need to save the original.
.CTRDB	Reads the data buffer. This is used by an inferior context to read data passed to it by its superior. This is not a destructive read and can be repeated.
.CTWDB	Writes the data buffer. This is used by an inferior context to return data to its superior. Once data is written using this function, any old data originally set by the superior is lost. An inferior may write the data buffer repeatedly, each write function overwriting any existing data.
.CTRQT	Reads a job's context and saved-page quotas. This function requires the use of the data buffer to pass and return arguments. Word 1 (.CTJOB) specifies the target job number. Words 2 and 3 (.CTCTQ and .CTPGQ) return the context quota and saved-page quota respectively.
.CTSQT	Sets a job's context and saved-page quotas. This function uses the data buffer as function .CTRQT.
.CTNCX	Names a context.
.CTLIS	Lists context data for a job.

On any return from the CTX. UUO, the AC contains information pertaining to the status of the data buffer and errors. Regardless of the success or failure of the UUO, if a data buffer is used, CT.DAT (1B0) will indicate the existence of data returned in the buffer. CT.DBT (1B1) is returned if the buffer is truncated. CT.ETX (1B2) flags UUO error text in the buffer. The CT.RDL (777B27) field is a count of the number of words returned in the buffer. For UUO function .CTSVR, if the implied RUN UUO fails, CT.RUN (1B3) indicates the returned error code is a RUN UUO error, not a CTX. UUO error. RUN errors never return error text in the data buffer. Finally, the CT.ERR (777B35) field is the CTX. or RUN UUO error code. This is always returned on UUO errors even when the data buffer contains error text.

Table 4–5: CTX. UUO Error Codes

Code	Symbol	Meaning
0	CXIFC%	Illegal function code
1	CXACR%	Address check reading arguments
2	CXACS%	Address check storing answers
3	CXNEA%	Not enough arguments
4	CXNLI%	Not logged in
5	CXLOK%	Locked in core
6	CXDET%	Detached
7	CXSCE%	System context quota exceeded
10	CXSPE%	System page quota exceeded
11	CXJCE%	Job context quota exceeded
12	CXJPE%	Job page quota exceeded
13	CXNCS%	Not enough core to save context
14	CXNCD%	Not enough core to return data block
15	CXICN%	Illegal context number
16	CXNSC%	No superior context
17	CXNPV%	No privileges to set quotas
20	CXIJN%	Illegal job number
21	CXCSI%	Cannot switch to an intermediate context
22	CXCDI%	Cannot delete an intermediate context
23	CXCDC%	Cannot delete the current context
24	CXCNP%	Context not privileged
25	CXNDA%	No data block available

4.12.5 Command Interface

The three commands CONTEXT, PUSH, and POP context creation, deletion, display, naming, and switching. Contexts can also be changed when certain programs are run by using the auto-save and restore facilities.

4.12.5.1 CONTEXT Command

The CONTEXT command allows a user to display and manipulate contexts within his or her job. It requires that the job is halted and at monitor level. It always preserves the current core image. Most options of the CONTEXT command take at least a context name or number. A context name is a string of 1 to 6 alphanumeric characters; the first character must be alphabetic. A context number is a decimal number assigned by the monitor and associated with a given context. A period (.) may be substituted in place of any context name or number to represent the current context for a job.

4.12.5.2 List Options

A user may list the status of a single context or the status for all contexts. The simplest form of the command is CONTEXT with no arguments. This produces a display showing the status of all contexts.

```
.CONTEXT

Contexts used/quota = 2/4, pages used/quota = 33/1000
    Context      Superior     Prog     Idle time
  TOPLVL   1                 DDT      3.53
*          2   TOPLVL   1
```

In the above example, the user used 2 out of 4 contexts allowed for the job and 33 out of 1000 saved-pages. Context #1 is named TOPLVL and was running the DDT program when it was saved. TOPLVL(1) has one inferior context, context #2, which is unnamed. Currently, no program is running. TOPLVL(1) had been idle for 3.53 seconds before the CONTEXT command was typed. An asterisk indicates the current context. In this case, it is context #2. An individual context may be listed using the command CONTEXT *n*/LIST. *n* is a valid context name or number. The following examples show the use of the /LIST option:

```
.CONTEXT TOPLVL/LIST

    Context      Superior     Prog     Idle time
  TOPLVL   1                 DDT      3.53

.CONTEXT 2/LIST
    Context      Superior     Prog     Idle time
*          2   TOPLVL   1
```

4.12.5.3 Create Option

The CONTEXT command can be used to create an adjacent context chain. Context blocks in the new chain are not associated with the current PUSH chain in any way. This facility allows a user to create multiple contexts so that programs that take lots of time to initialize (such as MS) may be started, halted, and "kept". Keeping an idle context with an initialized program allows a user to quickly switch to that context and continue it, thus avoiding the expensive overhead of restarting the program. The syntax to create an adjacent context is:

```
.CONTEXT =              ;CREATE AN ADJACENT CONTEXT
.R MS                   ;RUN A PROGRAM
MS>                     ;PROGRAM TYPEOUT
MS>EXIT                 ;EXIT PROGRAM
.CONTEXT                ;LIST ALL CONTEXTS

Contexts used/quota = 2/4, pages used/quota = 33/1000
    Context      Superior     Prog     Idle time
  TOPLVL   1                 DDT      3.53
*          2                 MS
```

Optionally, preceeding the equal sign with a name associates that name with the newly created context.

4.12.5.4 Delete Option

Any single context or a context at the end of a PUSH chain can be deleted from any other context. The command syntax is CONTEXT *n*/KILL, where *n* is a valid context name or number.

```
.CONTEXT                ;LIST ALL CONTEXTS

Contexts used/quota = 2/4, pages used/quota = 33/1000
    Context      Superior     Prog     Idle time
  TOPLVL   1                 DDT      3.53
*          2

.CONTEXT TOPLVL/KILL    ;DELETE TOPLVL(1)
.CONTEXT                ;LIST ALL CONTEXTS

Contexts used/quota = 1/4, pages used/quota = 0/1000
    Context      Superior     Prog     Idle time
*          2
```

4.12.5.5 Naming Option

The CONTEXT command can be used to associate an alphanumeric name with any context owned by the job. The command format is CONTEXT *new=old*, where *new* is the new name to assign to context *old*. As with other CONTEXT command options, *old* can be any valid context name or number.

```
.CONTEXT                ;LIST ALL CONTEXTS

Contexts used/quota = 2/4, pages used/quota = 33/1000
    Context      Superior     Prog     Idle time
  TOPLVL   1                 DDT      3.53
*          2

.CONTEXT FOO=2          ;ASSIGN "FOO" TO CONTEXT 2
.CONTEXT                ;LIST ALL CONTEXTS

Contexts used/quota = 2/4, pages used/quota = 33/1000
    Context      Superior     Prog     Idle time
  TOPLVL   1                 DDT      4.12
* FOO      2
```

Additionally, a context can be named when it is created via the CONTEXT command. See the section on the create option.

4.12.5.6 Switch Option

Once two or more adjacent contexts are established, the user can switch from one context to the other using the CONTEXT *n* command. *n* can be any valid context name or number. Here, as in the delete option, the same restrictions apply: a user can switch between the current context and any single adjacent context or one at the end of a PUSH chain.

```
.CONTEXT                ;LIST ALL CONTEXTS

Contexts used/quota = 2/4, pages used/quota = 102/1000
    Context      Superior     Prog     Idle time
* TOPLVL   1                 DDT
  MS       2                 MS       10:02

.CONTEXT MS             ;SWITCH TO MS(2)
.CONTEXT                ;LIST ALL CONTEXTS

Contexts used/quota = 2/4, pages used/quota = 33/1000
    Context      Superior     Prog     Idle time
  TOPLVL   1                 DDT      3.53
* MS       2                 MS
```

4.12.5.7 PUSH Command

This command allows a user to create an inferior context. The following sequence of commands demonstrate the PUSH command:

```
.R DDT                  ;RUN A PROGRAM
DDT                     ;PROGRAM TYPEOUT
^Z                      ;EXIT PROGRAM
.PUSH                   ;CREATE AN INFERIOR CONTEXT
.CONTEXT                ;LIST CONTEXTS

Contexts used/quota = 2/4, pages used/quota = 33/1000
    Context      Superior     Prog     Idle time
  TOPLVL   1                 DDT      3.53
*          2   TOPLVL   1
```

The display shows context #2 is inferior to context TOPLVL(1). The action of the PUSH command caused an additional context to be charged to the user. In the case of DDT, 33 additional pages have been counted against the user's saved-page quota. The context with the DDT core image had been idle for 3.53 seconds before the CONTEXT command was typed. The functionality available in the PUSH command is identical to that provided by the .CTSVH function to the CTX. UUO.

4.12.5.8 POP Command

This command provides the only means to return to a superior context. Its purpose is to undo what the PUSH command did. That is, destroy itself and restore its superior context. The following sequence of commands demonstrates the use of the POP command.

```
.POP                    ;RETURN TO SUPERIOR CONTEXT
.CONTEXT                ;LIST CONTEXTS

Contexts used/quota = 1/4, pages used/quota = 0/1000
    Context      Superior     Prog     Idle time
* TOPLVL   1                 DDT
```

```
.CONTINUE                ;RETURN TO THE SAVED PROGRAM
<>                       ;PROGRAM TYPEOUT
```

POPping a context causes a free context to be returned to the number available to the user. In the DDT program, 33 pages is no longer counted against the total saved-pages allowed to the user.

4.12.5.9 Auto-Save and Restore

Normally, when a command is given that runs a program, the user's core image is destroyed. The system administrator has the option of marking certain commands as auto-save commands. When the user types a command that runs one of these programs, the monitor automatically saves the user's current context, and runs the specified program. Whenever the job attempts to return to monitor level due to EXIT UUO, CTRL/C, HALT, or fatal error in job, the monitor performs an automatic restore of the saved context for the user. The net result is the user's core image is still intact. Additionally, user defined commands can be flagged as auto-save commands. This sequence of auto-save, running a program, and auto-restore is identical to the user typing PUSH, RUN, and POP commands, with one exception: a user's context and saved-page quotas are never checked when an auto-save is in progress. This is done to allow the system administrator to set a user's context quota to 1 and still allow the use of normal monitor commands that will invoke the auto-save and restore sequence. System context and saved-page quotas are always checked under all circumstances.

4.12.5.10 SET WATCH Command

An additional keyword has been added to the SET WATCH command. The CONTEXTS keyword is now accepted by the SET WATCH command. When this feature is enabled, context information is displayed each time the current context changes. For example:

```
.CONTEXT                 ;LIST CONTEXTS

Contexts used/quota = 1/4, pages used/quota = 33/1000
    Context      Superior     Prog    Idle time
* TOPLVL   1                 DDT

.SET WATCH CONTEXTS      ;ENABLE CONTEXT WATCH
.R DDT                   ;RUN A PROGRAM
DDT                      ;PROGRAM TYPEOUT
^Z                       ;EXIT PROGRAM
.PUSH                    ;CREATE AN INFERIOR CONTEXT
[Context 2]              ;WATCH CONTEXT TYPEOUT
.POP                     ;RETURN TO SUPERIOR CONTEXT
[Context TOPLVL(1)]      ;WATCH CONTEXT TYPEOUT
```

4.12.6 Conservation of System Resources

Any time JOBKL is called to kill a job and the job has contexts, the contexts are deleted, freeing up monitor free core and swapping space. This is accomplished by CTXSER ordering the job's entire list of context blocks into one PUSH chain and positioning the current context to the end of the chain. A POP command is simulated that propagates back to the highest superior context. Then, the final context block is deleted and the job is allowed to logout.

4.12.7 Administration

The standard monitor installation automatically makes alternate contexts available to a customer. The system administrator should refer to the *TOPS-10 Software Installation Guide* if non-standard default context and saved-page quotas are desired. Also, swapping space may need to be increased depending on the amount of existing swapping space and the expected increased load created by saving contexts. The *TOPS-10 Software Installation Guide* provides guidelines for the allocation of swapping space.

Using alternate contexts increases the swapping load on a system. The amount of increase can vary greatly depending on the size of the jobs being saved and restored. For systems using the recommended amount of swapping space, it is strongly suggested that the amount of swapping space be increased to compensate for the additional core images being saved.

Chapter 5

InterProcess Communication Facility (IPCF)

5.1 Introduction

IPCF allows user and system programs and the monitor to communicate with one another. In IPCF, communication takes place between processes. These processes can be an entire program, or just a portion of a program. A single program can also have one or more processes. For example, the QUASAR program within GALAXY has five different processes that each send and receive IPCF messages.

The monitor routines to support IPCF reside in the IPCSER module.

5.1.1 Communicating with packets

Processes that use IPCF communicate with each other by sending and receiving packets. These packets can be of two types: short-form and page-mode. A short-form packet consists of header information and a small number of words of data (up to 12_8 in the standard monitor). A long-form packet consists of the same header information as a short-form packet, plus a full 512-word page of data.

Processes residing within user-mode programs use the IPCFS., IPCFR., IPCFQ., and IPCFM. UUOs to send and receive packets. Monitor-resident processes call the equivalent UUO service routines directly.

5.2 Identifying Processes

A process that wants to send a message using IPCF must identify who the intended receiver is. This can be done by using the receiver's:

1. Job number or Job-Context Handle (JCH)
2. Process IDentification number (PID)
3. Symbolic process name

5.2.1 Job Numbers and Job-Context Handles (JCH)

A process that wants to send a packet can simply provide the destination process' job number as the destination address. This, however, has three problems. First, it precludes sending packets to certain system processes. Second, if the destination job's program has multiple processes, they cannot be addressed directly. Third, if a job has multiple contexts that are enabled for IPCF reception, then a packet addressed to a job number could go to any of the job's contexts.

A slightly better alternative is to use the destination's job number or JCH as a packet address. This ensures that the packet is placed in the appropriate program's IPCF receive queue instead of being sent to an arbitrary context. This approach, while better, still shares the first two problems that accompany the job number approach.

5.2.2 Process IDentification Numbers (PIDs)

Instead of using job numbers, a sending process can specify the PID number of the destination. PIDs are monotonically increasing numbers that are assigned to a process by [SYSTEM]INFO (a system process that will be discussed later). PIDs are not reused until the system is reloaded. One job can have any number of PIDs assigned, up to the maximum PID quota for the account (the default is 2). Unlike job numbers, PIDs allow packets to be sent to any process on the system.

5.2.3 Symbolic Process Names

When a process requests a PID from [SYSTEM]INFO, it can optionally provide [SYSTEM]INFO with a symbolic name for the process. This name is placed in a table that is managed by [SYSTEM]INFO. If one process wants to send a message to another process, but only knows the destination process' name and not its PID, the sending process can ask [SYSTEM]INFO for the PID associated with that name.

5.3 System Processes and Known PIDs

In a standard TOPS-10 system running with GALAXY, several system-wide IPCF processes exist. Some of these processes are required for IPCF to function, while others are used to communicate with the monitor, other parts of GALAXY, or directly with users. The monitor maintains a table of these special PIDs for easy access, at .GTSID in COMMON (GETTAB table 126). The following table contains the PIDs assigned to the following symbolic names, in order. If a process resides in a timesharing job, the program name is also listed.

IPCF process	Function	Resident in
[SYSTEM]IPCC	System IPCF controller	
[SYSTEM]INFO	System IPCF information manager	QUASAR
[SYSTEM]QUASAR	Central manager of GALAXY	QUASAR
[SYSTEM]MDA	Mountable Device Allocator	QUASAR

IPCF process	Function	Resident in
[SYSTEM]TAPE LABELLER	Magtape labelling process	PULSAR
[SYSTEM]FILE DAEMON	File access arbiter	FILDAE
[SYSTEM]TAPE AVR	Automatic Volume Recognizer for magtapes	QUASAR
[SYSTEM]ACCOUNTING DAEMON	Accounting records collector	ACTDAE
[SYSTEM]OPERATOR	Operator interface program	ORION
[SYSTEM]ERRLOG	System error logger	(not used)
[SYSTEM]DISK AVR	Automatic Volume Recognizer for disks	QUASAR
[SYSTEM]TGHA	MOS memory error analysis and reconfiguration	TGHA
[SYSTEM]DECNET CONTROLLER	DECnet's Network Management Layer process	NML
[SYSTEM]GOPHER	Performs miscellaneous IPCF duties	
[SYSTEM]CATALOG DAEMON	Manages system disk and labelled tape catalog	CATLOG
[SYSTEM]MAILER	Sends and receives mail messages	MX

Three of these processes are parts of IPCF itself:

- [SYSTEM]IPCC, called an "exec pseudo-processes" because it resides in the monitor.
- [SYSTEM]GOPHER, another "exec pseudo-process."
- [SYSTEM]INFO, is a user-mode process that resides in the GALAXY component QUASAR.

5.3.1 [SYSTEM]IPCC

[SYSTEM]IPCC is the IPCF Controller process. It can perform many functions for both privileged and unprivileged user jobs. It can create and destroy PIDs, or transfer PIDs between user jobs. Since it is a resident part of the monitor, it can be called by other parts of the monitor to send messages to other processes (mostly those residing in various GALAXY components) to inform them of changes in the system (such as people logging out, or spooled files being closed, or MDA-controlled devices being placed on-line).

[SYSTEM]IPCC can send messages for two reasons:

- To respond to requests from user-mode processes. These can involve creating a PID, providing information, or checking IPCF quotas. [SYSTEM]IPCC's response contains the result from executing the user's request.

- To generate unsolicited messages. If something happens in the system that is of interest to one of the system processes (such as a disk unit coming on line, important to [SYSTEM]DISK AVR, or a job logging out, important to QUASAR), then the monitor tells [SYSTEM]IPCC to send a message to the system process that needs to know.

[SYSTEM]IPCC is a "talk only" process. That is, when it sends a message, it does not expect a response from the recipient. This is because any messages it sends are either in response to a user request or are on behalf of another part of the monitor that wanted to spread the word about a system event.

5.3.2 [SYSTEM]INFO

[SYSTEM]INFO is the information manager for IPCF. Like [SYSTEM]IPCC, [SYSTEM]INFO can create and destroy PIDs. However, PIDs created by [SYSTEM]INFO can have a symbolic name associated with them. [SYSTEM]INFO manages the system-wide table of jobs, PIDs, and symbolic PID names. User programs can send requests to [SYSTEM]INFO to create or delete named PIDs, or to find out the PID associated with a particular symbolic name. When a job logs out, [SYSTEM]IPCC automatically sends [SYSTEM]INFO a message telling it to destroy any PIDs and symbolic names owned by that job. The [SYSTEM]INFO process resides in a user-mode program. Prior to the 7.03 monitor, [SYSTEM]INFO lived in the SYSINF program. From 7.03 onward, [SYSTEM]INFO resides in QUASAR, the GALAXY queue manager.

5.3.3 [SYSTEM]GOPHER

[SYSTEM]GOPHER is the go-between (hence, the name gopher) between the monitor and various system processes. Like [SYSTEM]IPCC, [SYSTEM]GOPHER resides within the monitor. Also, like [SYSTEM]IPCC, it allows the monitor to generate and send IPCF messages to system processes. However, unlike [SYSTEM]IPCC, it waits for a reply from the destination process before continuing. This allows the monitor to send messages as a byproduct of UUO execution, since it provides a mechanism to block the job until a reply is received.

Any part of the monitor can use [SYSTEM]GOPHER to communicate with another system process. User-mode use [SYSTEM]GOPHER whenever they execute certain UUO functions:

- LOOKUPs or ENTERs on File Daemon protected files
- QUEUE. monitor calls
- IPCFM. calls for [SYSTEM]INFO functions

Each of these is described below.

5.3.3.1 LOOKUPs and ENTERs on File Daemon Protected Files

The file system is a major user of [SYSTEM]GOPHER. When a user issues a LOOKUP or ENTER UUO for a file that has a 4 or higher in the owner's protection code field, and the file (or the file's directory) is protected against that user, the file system calls [SYSTEM]GOPHER, and tells it to send a message to the File Daemon (FILDAE), asking it if the user can access the file. Because this exchange takes a relatively long time to complete, [SYSTEM]GOPHER places the user's job in the event wait queue for IPCF traffic (EW for EW.IPC), freeing the Scheduler to run another job. When [SYSTEM]GOPHER receives its

response, it removes the job from the event wait state and returns to the caller with the response.

5.3.3.2 QUEUE. UUOs

A similar chain of events occurs when a user executes a QUEUE. UUO. The monitor calls [SYSTEM]GOPHER, and tells it to send the details of the request in a message to one of the system processes (usually a GALAXY component). Again, [SYSTEM]GOPHER blocks the requesting job until it gets a response from the component or a no-response timer expires. In the former case, it returns to the calling routine with the response. In the latter case, it returns to the caller with an error code indicating that the component didn't answer the request.

5.3.3.3 IPCFM. Calls to [SYSTEM]INFO

The IPCFM. monitor call greatly simplifies the coding needed for a user-mode process to interact with both [SYSTEM]IPCC and [SYSTEM]INFO by replacing a two-UUO IPCFS.–IPCFR. message exchange. [SYSTEM]IPCC functions are handled directly, since the message is sent directly to [SYSTEM]IPCC. [SYSTEM]INFO functions, however, cannot be handled in the same manner. When an IPCFM. request for [SYSTEM]INFO is received, IPCSER places the requesting process' job into the event wait for IPCF state, and sends a [SYSTEM]GOPHER request to [SYSTEM]INFO to perform the requested function. When [SYSTEM]INFO's response is received, it is returned to the job. The job is removed from the event wait queue and allowed to continue.

5.4 Message-Sending Mechanics

IPCF messages are sent in the form of packets from one process to another. Each packet consists of two parts:

- A Packet Header Block (PHB), six words in length.
- A Packet Message Block (PMB), containing the actual message.

The PHB describes the characteristics of the communication (for example, the sender and receiver) and points to the PMB, where the actual message is stored.

The PMB can be of two types: short-form, or long-form. Each is described below.

5.4.1 Short-Form Messages

A short-form message contains a standard PHB and a PMB of a few words, up to a maximum of 10. This limit is specified by the monitor parameter M.PKTL, and is stored in the GETTAB table .GTIPC, item 0 (%IPCML).

When a user sends a short-form packet, the packet contents are copied from the user's buffer into monitor freecore, where it is held until received. When the intended receiver executes the IPCFR. UUO to read the packet, the monitor copies the packet into the user's buffer.

5.4.2 Long-Form (Page-Mode) Messages

A page-mode message consists of a standard PHB, which points to a PMB of one memory page (512 words) in length. The PMB in a page-mode message starts on a page boundary, and is always one page long. †

When a user sends a page-mode message, the monitor removes the page from the user's address space and marks it as an IPCF page. The page is also marked as eligible for page-out. If system memory is scarce, the pager writes a copy of the page to the swapping area and recovers the memory for other uses. When the receiving process executes the IPCFR. call to get the page, the monitor maps the page into the receiver's address space. If the page was paged out before being received, it is not paged in now. The process' first reference to the page causes a page fault that, in turn, causes the monitor to bring the page into memory.

5.4.3 Performance Considerations

When running a monitor with the M.PKTL parameter set to its default value of 10 words, delivery of a short-form IPCF packet is quicker than a page-mode packet. The CPU time required to copy a handful of words twice (first from the sender's PMB to monitor freecore, then from freecore to the receiver's PMB) is significantly shorter than the time needed to manipulate the page map for page-mode packets.

Short-form packets of larger lengths can be sent by changing the value of M.PKTL during the MONGEN dialogue. The amount of CPU time saved by using short-form packets decreases as the packet size increases. If M.PKTL is set to values above 150 words, short-form packets take longer to execute than page-mode packets. In addition, since short-form packets are stored in monitor freecore while in transit, larger packets take up more room. On a system with much IPCF traffic, system freecore fills up rapidly if messages are allowed to back up. This can cause the system to come to a grinding halt. On the other hand, since the PMB in a page-mode message is an entire page (and is eligible for page-out) it is far less likely that system congestion will occur due to backed-up messages. ‡ Therefore, page-mode messages are much more efficient than short-form messages when there are a lot of data to move between processes.

5.5 IPCF Data Structures

IPCF maintains a set of data structures that indicate the state of all processes, packets, and message queues. These data structures fall into the following categories:

- System-wide data
- Job-specific data
- Context-specific data
- Packet-specific data

Each of these is described below.

---

† Technically, a page-mode packet's PMB can have two potential sizes: zero words and 512 words.

‡ It is less likely but not impossible. The PHB for page-mode messages still exists in monitor freecore while in transit. Even though it is only six words in length, enough of them could pile up to cause problems if traffic is very heavy and is allowed to back up.

5.5.1 System-Wide Common Data

The system-wide data structures all reside in COMMON and contain information that is important to IPCF. These include:

Data Structure	Contents
PIDTAB	The master table of all PIDs that exist on the system.
IPCTAB	The GETTABable table of IPCF quotas and statistics, such as packet, word, and page counts. It also contains copies of the PIDs of the monitor processes [SYSTEM]IPCC and [SYSTEM]GOPHER.
.GTSID	The GETTABable table of special system PIDs.
.GTQFC	The QUEUE. UUO function table. The QUEUE. UUO causes [SYSTEM]GOPHER to send packets to system processes to carry out various functions. Customers can add new functions (with negative function numbers) to the top of the table.

5.5.2 Job-specific data

In addition to the system-wide data, the monitor maintains a set of data for each job in the system. This data resides in the Process Data Block (PDB), the repository for job-specific information. The PDB words that are used by IPCF, and their contents, are as follows:

PDB word	Contents
PDIPC	The left half points to the first (oldest) packet in the job's IPCF receive queue. The right half contains the number of outstanding packets, both sent and to be received.
.PDIPA	The left half contains the number of packets sent since login, while the right half contains the number received since login.
.PDIPQ	IPCF flags and quotas.
.PDIPL	IPCF queue interlock word.
.PDPID	PID number for a PID-specific receive.
.PDIPI	PID number of the current job's [SYSTEM]INFO process.
.PDIPN	The left half contains the pointer to the last packet in the receive queue. The right half is zero.
.PDQSN	The left half contains the sequence number of the last packet sent to the File Daemon by [SYSTEM]GOPHER in the current job's behalf. The right half contains the same information for packets sent to a system process as the result of a QUEUE. UUO.

PDB word	Contents
.PDEPA	Can contain three different kinds of data during a [SYSTEM]GOPHER message exchange with a system process: 1. After [SYSTEM]GOPHER has sent the message, and while it waits for a response, .PDEPA contains the PID of the destination process. [SYSTEM]GOPHER places the initiating job in the "event wait for IPCF" queue. 2. When the destination process sends its response, .PDEPA contains the exec address of the packet. 3. If the destination does not respond, or if the packet is turned around because of an error, .PDEPA contains an error code.

The set of PDB words above is collectively referred to as the IPCF Process Control Block (PCB). The two IPCF processes that reside in the monitor ([SYSTEM]IPCC and [SYSTEM]GOPHER) have their own private executive PCBs. These blocks are located at IPCADR and GFRADR, respectively. Their format is identical to the user-mode PCB, except for an extra word at the end. Locations within each exec PCB have names of the form .EP*xxx*, where each *xxx* is the same as those in the list above. The extra word in the exec PCB, .EPADR, contains the address to call (using a PUSHJ) when an IPCF packet arrives for the process.

The .EP*xxx* and .PD*xxx* IPCF symbols have one additional difference. While the .PD*xxx* symbols are numbered relative to the start of the PDB block, the .EP*xxx* symbols are numbered relative to the beginning of the IPCF PCB. Thus, while the symbol .PDIPC (the offset of the first word of the IPCF PCB) has a nonzero value, its counterpart, .EPIPC, does have a zero offset. IPCSER uses the .EP*xxx* offsets exclusively when accessing PCB locations. Accumulator W, which contains the address of the current PCB, is used as an index register. If the PCB is for an exec pseudo-process ([SYSTEM]IPCC or [SYSTEM]GOPHER), then W points to the exec PCB for one of those processes. If the PCB is for the current context of a user job, then W points to the PCB words within the PDB. If the current PCB is for an inactive context, then W points to the IPCF words in the saved context block for that context.

5.5.3 Context-specific data

When a job uses multiple contexts, the PDB words that contain the IPCF state for each saved context are placed in a saved-context block. This allows each context to have its own set of PIDs and streams of packets. The PDB words are stored in the saved-context block in the same order that they occur within the PDB itself.

When a packet is to be delivered, the routine VALPID is called within IPCSER. VALPID takes as input the job number, JCH, or PID of the destination process. If the input is valid, VALPID sets up accumulator J to point to the correct JCH of the destination, and W to point to the first word of the IPCF data for the target process.

5.5.4 Packet-specific data

The monitor maintains a block of data about every packet that is active on the system (that is, packets sent but not yet received or discarded). This information resides in a packet descriptor block, which is stored in monitor freecore. The packet descriptor block contains the following words:

Word	Contents
.IPCFL†	Link and flags word. The left half contains a pointer to the next packet. The right half contains various flags about the type and status of the packet.
.IPCFS†	Sender's PID number.
.IPCFR†	Receiver's PID number.
.IPCFP†	The left half contains the length of the PMB.
.IPCFI	Information about the location of the packet, whether it's in core or has been paged out to disk, and where it is located.
.IPCFU†	PPN of sending process. Filled in by monitor during a send.
.IPCFC†	Sender's capability word. The status of the bits indicate whether sender has JACCT., is logged-in, is execute-only, can poke the monitor, or has IPCF privileges. It also contains the job-context number of sender. This word is also filled in during a send.
.IPCFD	First word of data in a short-form packet.

†These words are found in the packet header block for any of the IPCF UUOs. Some of the words and fields have different meanings depending on whether it is in the UUO packet header block or in the monitor's packet descriptor block.

Chapter 6
The Scheduler

The scheduler selects the next job to run. This chapter explains how jobs are placed in various queues and how they are selected to run, based on scheduling parameters such as: priority of the job, interactiveness of the user, and current state of the job.

6.1 Introduction

The function of the scheduler is to choose the next job to run. In order to do this, the scheduler must ensure that all jobs are properly queued according to their current state. Specifically, the scheduler performs all requeuing, selects a job to run next, controls the allocation of sharable resources, and governs the priority of jobs to be in physical memory. The overall philosophy of scheduling in the DECsystem–10 and the specific procedures by which this philosophy is implemented is explained in this chapter.

6.2 Queues

Scheduling in the DECsystem–10 is based on the use of queues and wait state codes. The queue in which a job waits, and its position within the queue, determine the job's relative priority for the use of the CPU. The wait state code can be used to indicate that a job is waiting for a particular resource and is not able to use the CPU until that resource is available. Jobs that are most likely to be able to use the CPU wait in the processor queues, called PQ1 and PQ2. All jobs in PQ1 are given high response but for a very short time. Jobs enter at the back of PQ1 when they start to run or when they come out of any long term wait such as terminal I/O wait, command wait, or DAEMON wait. Certain jobs that have been sleeping or hibernating are also be queued into PQ1 if they were sleeping for more than one second. Jobs enter PQ2 from PQ1 when they have exceeded the amount of time that they are allowed in PQ1. In most systems, there are also high priority processor queues for particularly important jobs and real-time applications. Users must have special privileges to get their jobs into these queues.

Wait state codes are maintained for each job in the JBTSTS table and are used to define short- and long-term states. Short-term states apply to waits that are predictably less than one second. These are:

- Wait satisfied
- Sharable resource wait
- I/O wait
- Short sleep (less than one second)

There are no special queues associated with short-term states. Jobs in short-term wait states maintain their position in a run queue but have their wait state code altered.

Long-term states refer to waits of longer than one second. These are listed below:

- Command wait
- Jobs waiting for service by DAEMON
- Terminal I/O wait
- Sleep wait
- Event wait

These wait states, and two others, have individual queues. The two other states having individual queues are: stopped (job does not want to run) and null (no job occupying a particular job slot). Jobs going into long-term states enter at the rear of the respective queue.

The scheduler does the following:

1. Manages In-Core Protect Time (See Section Section 6.4)
2. Deals with the current job if necessary
3. Requeues other jobs needing requeuing
4. Calls the swapper to make a swapping decision, if necessary
5. Chooses the next job to run
6. Allocates sharable resources

Physical memory is a sharable resource for which there is no one queue or wait state. Jobs in any processor queue may either be in an in-core queue or an out-core queue. Maintaining separate in-core and out-core queues for each processor queue reduces overhead in queue scanning for job selection. The scheduler uses the in-core queues for job selection. The swapper uses the out-core queue for swap in job selection and the in-core queues for swap out job selection. The swapper attempts to keep jobs in core that are most likely to do productive work (make use of system resources). However, swapping depends very much on the sizes of jobs, as well as the queue in which they are located.

6.2.1 Queue Transfers

All transfers of jobs from one queue to another are performed by the routine QXFER in SCHED1. Transfers are made to the beginning or end of a specified destination queue. The destination queue can be specified in one of three ways:

- On a fixed transfer, a queue number is given directly.
- On a link transfer, the destination queue is specified as a function of the job's current queue.
- On a job size transfer, the destination queue is determined by the size of the job.

The routine requesting a queue transfer can also request that the job's quantum run time be reset. This is done when the job is being requeued into a run queue.

When a queue transfer is requested, the calling program specifies the job number, its current queue number (on link transfers), and the address of a transfer table. The transfer table specifies how the job should be requeued. A transfer table consists of two words, in the following format:

```
+------------------+------------------+
|       place      |     function     |
+------------------+------------------+
|      quantum     |    destination   |
+------------------+------------------+
```

place is negative (1B0) for a transfer to the end of a queue and zero for a transfer to the beginning of a queue. TOPS-10 does not permit requeuing to the beginning of a queue (RBQ stopcode).

function has one of two values, QFIX or QLNKZ, corresponding to the manner in which the destination queue is to be determined. These are actually labels for the entry points of the routines within QXFER that determine the destination queue. However, they can be thought of as codes specifying the type of transfer.

On fixed destination transfers, *quantum* and *destination* are the actual values of the new quantum run time and destination queue number. A negative value of *quantum* indicates that the quantum run time is not to be changed. A positive value of *quantum* provides the address of a word containing a new Quantum Run Time (QRT) value. *destination* is always negative. It is used as an index to JBTCQ.

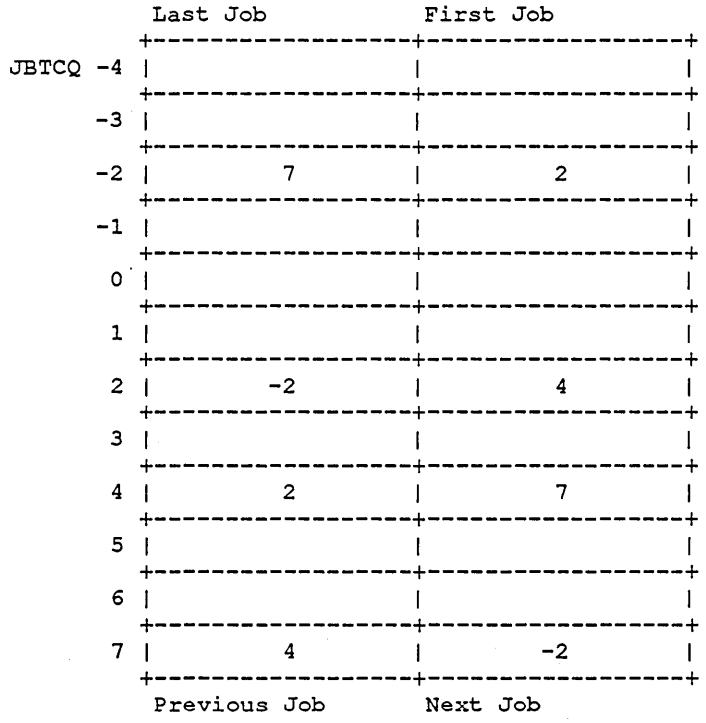
On link and job size transfers, *quantum* and *destination* are addresses of tables that are used to determine the destination queue and quantum run time.

The actions taken by the queue transfer routine on a fixed destination transfer are listed below:

- The calling routine sets up the job number in AC J and the address of a transfer table in AC U, and does a PUSHJ to QXFER.
- At QXFER, AC P4 is loaded from the second word of the transfer table. Then, there is a jump to the address in the right half of the first word. On a fixed destination transfer, the jump is to QFIX.

- At QFIX, if the job is being requeued to a run queue and successfully requested a High Priority Queue (HPQ), the priority level is obtained from table JBTRTD. The corresponding HPQ number and quantum run time are obtained from tables QTTAB and QQSTAB.
- The monitor checks to see if the job is currently in PQ2. If so, it is deleted from the core and no-core subqueues and removed from the just swapped-in list (JBTJIL).
- At QFIXB, the monitor checks to see if the job is in PQ2. If not, the job is removed from the output scan list (JBTOLS). If the job is in PQ2, a check is made to see if it is going back into PQ2. If not, it is removed from the output scan list.
- If the job is going into PQ1, the in-core protect time is reset. If the job is going into PQ2 and if its in-core protect time has expired, the job is placed on the output scan list and, based on job class, is entered into a parallel subqueue. If the in-core protect time has not expired, the job is cycled to the end of the just swapped-in list.
- The job is removed from its current queue. This is done by giving its "following job" entry to its preceding job, and its "preceding job" entry to its following job. A sample section of the current job requeue table is shown in Figure 6–1. Note that this procedure works correctly when the job is first or last in its queue, or is the only job in its queue.

Figure 6–1: Deletion of Job 4 from Its Queue



- The job is inserted into the destination queue. The job could be inserted following either the first link (the queue header) or the last link, depending on the value of the PLACE entry in the transfer table. In fact, the transfer is always to the back of the queue.

AC J points to the entry that is to be inserted. AC T2 is loaded from P4 and initially points to the queue header, which is correct if the insertion is to be at the beginning of the queue. If the insertion is to be at the end of the queue, AC T1 is backed up one entry, to point at the last entry. This is done with the instruction:

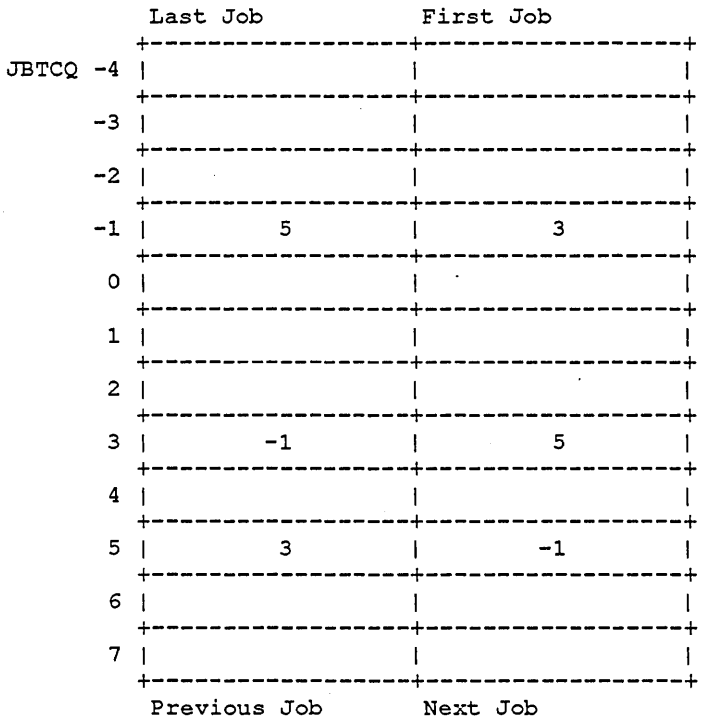
```
HLRE     T1, JBTCQ(T2)          ;Puts "last job"
                                ;Number from queue header into T1
```

AC T1 is thus loaded with the index of the entry at the end of the queue, after which the insertion is made. This is the value in the RH of the entry to which AC T1 now points.

The new linkages are set up with four easy instructions:

```
HRLM     J,JBTCQ (T2)           ;"End of Queue" entry
HRRM     J,JBTCQ (T1)           ;New "preceding" entry gets (J) as
                                ; following entry
HRL      T2, T1                 ;Put old "last job" number in LH of T2
MOVEM    T2,JBTCQ (J)           ;Create new job entry:
                                ; previous job number,, - queue number
```

Figure 6–2: Insert Job 1 at End of Queue 1

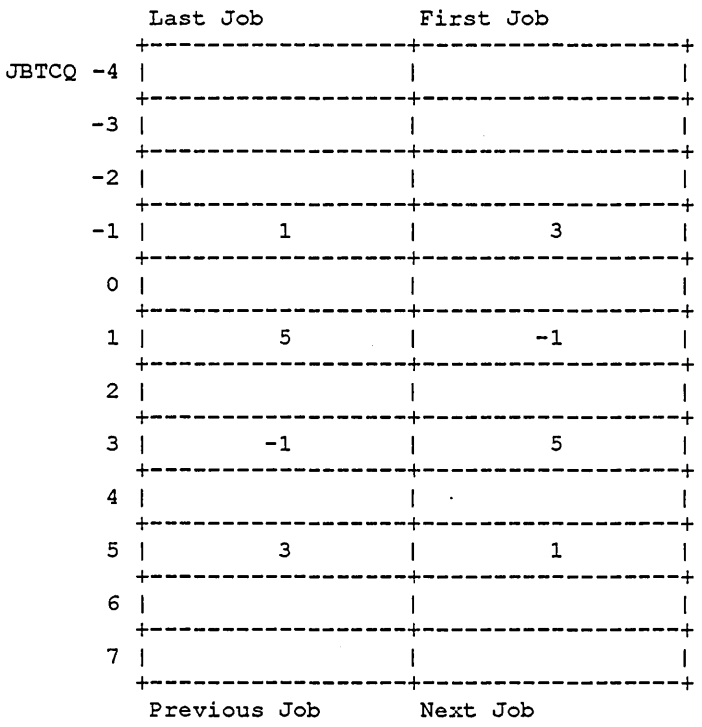


- P4 initially contains -1, the index of the queue into which the insertion is to be made. This is copied into AC T2.
- Since the insertion is to be at the **end** of the queue, AC T1 is loaded from the LH of the entry to which T2 (RH) points. The ACs are now set up (all values are relative to JBTCQ) as follows:
  - J points to the entry to be inserted (+1)

- T1 points to the entry after which it will be inserted (+5)
- T2 points to the queue in which the entry will be inserted (−1) To insert the new job:
- J is placed into the LH of JBTCQ (T2) and is placed into the RH of JBTCQ (T1).
- T2 RH is placed into the RH of JBTCQ (J).
- T1 RH is placed into the LH of JBTCQ (J).

The final result of the queue transfer is shown in Figure 6–3.

Figure 6–3: Final Result



- After the job is inserted into its new queue, if QUANT < 0, there is a jump to the routine exit.
- Otherwise, the value pointed to by QUANT is inserted into the Process Data Block for Job J (the job's quantum run time is reset).

The QFIX transfer is used for all queue transfers except for requeuing done as a result of quantum runtime expiration; in this case, the QLNKZ function is used. All requeuing is always done to the back of the destination queue.

On the QLNKZ transfers, the destination queue and, possibly, quantum run time must be determined. This is done by a simple table lookup. The job's current queue number, from JBTST2, is used to index the table to which DESTINATION points. The destination queue number is picked up from the corresponding entry in the table. If QUANT is given is also a table address, and the new quantum run time is picked up from a parallel entry in that

table. Once the destination queue number and quantum run time are determined, QLNKZ continues through the same procedure executed for QFIX.

6.3 Mechanics of Requeuing

Jobs are requeued according to events. Each time a job is to be requeued, a specific transfer table is used. Transfer tables are not set up or modified dynamically. Rather, for each event, the requeuing algorithm produces the address of a specific transfer table. This is done by means of several data structures and a number of checks for special cases.

The most general mechanism for requeuing jobs uses the Wait State Code (WSC) in the job's JBTSTS entry. On the next clock tick, the scheduler is called. The scheduler picks up the job's WSC and uses it as a pointer into QBITS. QBITS contains a dispatch address as well as the transfer table address, if there is one. The dispatch routine is executed and, if necessary, calls routine QXFER to perform the actual queue transfer. The WSC indicates the event, and the QBITS entry specifies the response. QBITS is used in putting jobs into I/O wait and sharable resource wait, removing jobs from I/O wait, and in requeuing jobs into a run queue after they have been stopped.

There are a number of special conditions that the scheduler checks for individually that call for specific transfer tables. For example, if the job's RUN bit is not set, the job is put into the stop queue, regardless of its WSC. If the job's command wait bit is set, it is put into the command wait queue. The use of the special bits allows the WSC to indicate a previous event. Since the WSC is unchanged, the job can be put back into its previous queue when returning from the stop queue or command wait queue (CTRL/C followed by CONTINUE).

Another bit used for this purpose is the JDC bit. The JDC bit is used to put the job into a queue to wait for a function to be performed by DAEMON. (DAEMON is a system program that runs as a user job and performs various functions for the monitor and other user jobs. It is, in effect, a non-resident portion of the monitor.)

6.4 CPU Scheduling

The scheduling of CPU time is based primarily on the order indicated by the scheduler scan table SSCAN. SSCAN specifies that the following in-core jobs be scanned in the order in which they are listed:

1. jobs in HPQs
2. jobs in PQ1
3. jobs in PQ2, by subclasses
4. background batch jobs

If no runable jobs are found, the null job is run. There are three additional factors influencing the normal selection of jobs. These are:

- Quantum Run Time (QRT). Quantum run time is an amount of run time assigned to a job when it enters a run queue. It is used to limit the amount of time a job maintains the same position in the processor queue and therefore provides for a fairness consideration in CPU scheduling.

- In-Core Protect Time (ICPT). In-core protect time provides a mechanism to prevent the swapper from immediately swapping out a job that was just swapped in. When the ICPT expires, the job is requeued to the back of PQ2.

  With QRT, ICPT ensures that a job gets its fair share of the CPU when it is in core. A job's ICPT value is decremented if it is in the EWQ or SLPQ, or was scanned to run but rejected (for example, because it was waiting for a sharable resource). Upon expiration of ICPT, a job is requeued to the back of PQ2 (or HPQ).

- The subclass quota is a percentage of a scheduling interval that is allocated to a subclass. During each scheduling interval, the scheduler considers the highest class. If no runable jobs are found in this class, the next class in the ordering is scanned.

6.5 Queue Scanning

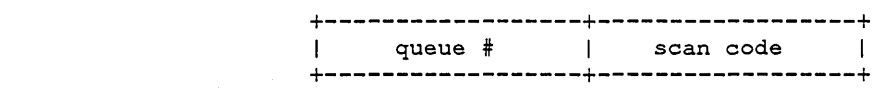
The following sections describe queue scanning and queue transfers. Queue scanning and transfer are not essential parts of the scheduler philosophy, however, they illustrate the table-driven nature of the scheduler. An understanding of these sections will make it easier to understand the detailed flow charts contained within the supplement.

6.5.1 Queue Scanning

The routine QSCAN is used to scan through one or more of the job queues. To use QSCAN, the caller must supply a scan table. A scan table specifies which queues to scan and the direction in which each is to be scanned.

A scan table consists of an arbitrary number of words, as shown in Figure 6–4.

Figure 6–4: Queue Scan Table Entry



queue # is the negative queue number, and is always written as a symbol (for example, −PQ2). The *scan code* is the label of the routine that performs a specific scanning operation. The following are some of the meanings:

Table 6–1: Scan Codes

Code	Function
QFOR	Scan the entire queue forward.
QFOR1	Look at the first entry only.
QBAK	Scan the entire queue backward.
QBAK1	Scan backward, but omit the first entry in the queue.
SQFOR	Scan subqueues forward according to SSSCAN (or SSSCN1).

A zero word terminates the table.

To use QSCAN, a routine puts the scan table address into accumulator U, and does a JSP to QSCAN. QSCAN gives a skip return, with the first job number in accumulator J. QSCAN also supplies an address to which the caller may return to continue scanning for another job number. Each time the caller returns to that address (with a JRST), QSCAN returns to the second word beyond the original call, supplying the next job number. QSCAN automatically steps from entry to entry in the scan table, and gives a non-skip return when the table is exhausted.

QSCAN is used by the scheduler in choosing the job to run next. It is also used by the swapper in selecting jobs for swap in and out.

6.5.2 Sharable Resources

There are a number of sharable resource wait states. A sharable resource is some part of the system, either software or hardware, that can be used by only one job at a time but is shared among different jobs over relatively short periods of time. For example, a DECtape controller is a sharable resource. The code and data relevant to it must be shared by all jobs doing I/O on the units it controls. Only one of these jobs may have I/O in progress at any given time. A line printer is not a sharable resource. It is given to a single job and that job has its exclusive use until the job chooses to give it up.

Access to sharable resources is controlled by table REQTAB. REQTAB has one entry for each sharable resource. Each entry is referenced by its own label, which is of the form *xx*REQ, where *xx* represents a two letter mnemonic for the resource. Each REQTAB entry is initialized to -1. Code that uses a sharable resource begins with an instruction that increments and tests the appropriate entry. If the value is greater than zero, the job for which the code was being executed must have its wait state changed to the short-term state associated with the sharable resource. The job becomes unblocked when it is being considered to run by the scheduler and the resource is available. For some resources, such as executive virtual memory, the REQTAB entry is initialized to *-n*, where *-n* is the number of executive virtual memory slots.

Another table, AVALTB, contains entries parallel to the REQTAB entries. The AVALTB entries are used as flags to the scheduler that the corresponding resources have become available while someone is waiting for them. The flag in AVALTB is set to a non-zero value at the end of the code that uses the sharable resource. It is set, however, only if there is a job waiting for the resource (that is, only if REQTAB is positive).

Sharable resource management is accomplished by two routines in CLOCK1: SRWAIT for resource allocation, and SRFREE for resource deallocation. These routines are called from various modules within the monitor that use the resources. Specifically, a PUSHJ P, *xx*WAIT, where *xx* is the two-letter mnemonic for the resource, results in a subsequent PUSHJ P, SRWAIT. The code within CLOCK1 for resource allocation does the following:

```
SRWAIT: (housekeeping)
        AOSG    xx'REQ              ;Is resource available?
                                    ; (actually addressed as REQTAB - K,
                                    ; indexed by WSC)
        JRST    SRAVAL              ;Yes, go use it
                                    ;No, block job
        (if necessary, return EVM)
        (if necessary, start partial cycle via entry to WSCHD1)
```

```
SRAVAL: (housekeeping)              ;Have resource
        (return to calling module)
```

The code within CLOCK1 for resource deallocation does the following:

```
SRFREE: SOSL    xx'REQ              ;Decrement request count
        SETOM   xx'AVAL             ;Mark as available, if needed
        (return to calling module)
```

Note

REQTAB entries are incremented and tested with a single instruction. If the resource is given up at interrupt level, there is no need to worry about the interrupt occurring between incrementing and testing.

In the case of magtape usage, a job is placed in the long-term event wait state and associated event wait queue if the magtape controller is not available.

The basic purpose of the sharable resource mechanism is to interlock a section of code so that only one job at a time executes any part of it. Since rescheduling is not allowed when the clock ticks during a UUO, there is no need to worry that another job will start through a monitor routine before a previous job finishes it. The only case in which this is possible occurs when a job might go into I/O wait within a routine.

Requirements for sharable resources may be nested. A job that owns one resource may have to queue for another. This may lead to a deadlock situation, that is, a situation where two jobs each wait for the resource owned by the other. Neither job can run, and neither can release the resource it owns. This problem is overcome by adherance to the following programming convention: any time there are nested requirements for a sharable resource, they must be nested in the same order.

Chapter 7

The Swapper

The swapper allows TOPS–10 timesharing to run efficiently by moving jobs of variable size in and out of physical memory. The primary purpose of the swapper is to maintain an equitable collection of runable jobs in memory from which the scheduler chooses one to run. The process includes swap selection, disk I/O, swapping area maintenance, high segment management and paging requests.

7.1 Introduction

Swapping allows the timesharing system to present the appearance of having more physical memory than it actually has. Assuming the presence of more jobs than can fit in memory, the core images of some subset of user jobs are written to disk and read back into memory as required. Ideally, no user can tell if his job is swapped in or out. Whether this artifice succeeds depends on many factors. These include: the number of users, their interaction rate, their memory requirements, the speed of the swapping device, and the amount of physical memory. This chapter presents the overall philosophy of swapping on the DECsystem–10 and the procedures by which this philosophy is implemented. The procedures are general enough to allow a great deal of flexibility in policy.

7.2 Swapping Philosophy

Under light loading, all active jobs reside in physical memory. However, if the requirement for physical memory exceeds that which is available, a job may be swapped out when the program stops or requires a response from the user. Each time the user requests another function, there is a short delay while the core image is swapped in. After the core image is swapped in, the program runs without being swapped out again until another user response is needed. Since the time to swap in a job is normally very short compared to human response times, the user does not usually notice a delay. Under these conditions, swapping can be quite successful.

Under heavier loading, the system is unable to keep all active jobs in physical memory. In this mode of operation, swapping gives the system the appearance of a larger, but correspondingly slower, system. The swapping algorithm becomes much more complex, because it must account for the possibility of jobs in memory entering a deadlock over a resource owned by a job that is swapped out. System overhead may spiral due to the (remote) possibility of swapping a job many times for a single interaction.

The effective size of physical memory is reduced because swapping an active job does not contribute to overall system efficiency. It is not available to run, and cannot have any I/O in progress. Moreover, the physical memory it occupies is not available until the swapping transfer is complete. The total system throughput would probably be greatest if runable jobs were never swapped out. However, the total throughput consideration must be balanced against the value of dividing the system resources fairly among all the users. All things being equal, it is preferable that the system appears twice as slow to all users than four times as slow to half the users. (Although the other half probably wouldn't object.)

In the way that swapping is implemented on the DECsystem–10, there is no direct distinction between the two modes of swapping discussed above. Swapping is based on the job queues, and the queue transfers are set up to give the desired swapping characteristics. There are three basic levels of priority, and normally, jobs at each level are given complete priority over jobs at the next level. Some exceptions are made to this policy in the interest of fairness. (Real time jobs are locked in core, exempt from swapping consideration).

Highest priority is given to jobs which must be in core in order for a command to be processed. Users expect instantaneous response to commands, but are normally more tolerant of delay when a program must be run. Actually, the class of commands that require a job to be in core is so small that the effect of this top priority level on the overall swapping behavior of the system is probably insignificant.

Next priority is given to the class of jobs that are, in some way, considered interactive. This class includes jobs doing I/O, all jobs that own sharable resources or are waiting for them, and all jobs that have had a recent user response. It is assumed that these jobs are the ones that make the most use of system resources, or whose users are less tolerant of delay. Under normal operating conditions, the jobs in at least these two top levels are kept in core.

The lowest priority for core memory is given to CPU-bound jobs. If a job is CPU-bound, it is doing a considerable amount of processing, and the user does not expect immediate response. These jobs do not contribute to the total system I/O throughput. Because the jobs are not using system resources other than the CPU and core, there is no advantage to having many of these jobs in core at a time.

The actual implementation is somewhat more complicated than that discussed in the preceding paragraphs. It is important, however, to understand the overall philosophy on which the implementation is based.

7.3 Mechanics of Swapping

Two scan tables, ISCAN and OSCAN, specify the relative priorities for jobs being in core as a function of the queue positions. The entries in these tables are fixed in a specific monitor, although they may be easily changed to implement a change in swapping policy. In effect, ISCAN is a listing of the out-of-core chain of queues needing swap-in. OSCAN is a listing of the in-core chain of queues containing jobs that might be swapped out. In addition to ISCAN and OSCAN, all the tables that control the requeuing of jobs are also important to the swapping process. The contents of ISCAN and OSCAN, as well as the requeuing tables, can be obtained from module COMMON.

The STOP Queue and SLEEP Queue are the first to be swapped out and are not considered for swapping in. After HPQs, the Command Wait Queue has top priority for being swapped in, followed by PQ1. In general, queues that are in ISCAN are listed in the opposite order to OSCAN. All actual queues (short-term wait queues have no members) are in OSCAN. ISCAN specifies only those jobs that are in Command Wait or Processor queues.

After a job is swapped in, there is an interval during which it is protected from being swapped out. This interval, called the In-Core Protect Time (ICPT), specifies an amount of time before the job becomes eligible to be swapped out. That time, in jiffies, is contained in the Process Data Block for each job. When a job is swapped in, the ICPT is computed according to the basic formula (the actual formula includes some scaling factors to account for the swapping units actually being used):

$$PROT0 + (PROT * Size\ in\ K)$$

where $PROT0$ and $PROT$ are constants computed during system initialization according to the speed of the swapping device. The purpose of these values is to make the ICPT interval dependent on the time required to swap the job. Typical values are 3 seconds for $PROT0$ and 0 seconds $PROT$. This results in a default ICPT of 3 seconds applied to all jobs, regardless of size. $PROT$ may be modified using the SCDSET program to make the ICPT a function of size.

Several other items are important to the swapper. The tables JBTIMO and JBTIMI specify the size of each swapped out core image and the memory requirement for swap in; these may not be the same. The table PAGTAB specifies which pages of core are free and which are in use. CORTAL tells how many pages are available either as free pages or pages occupied by dormant or idle segments. † BIGHOL tells the number of pages available in core. The table JBTADR gives the location of the Job Data Area and the size for each segment that has any physical core assignment. A job's JBTADR entry is still set up while it is being swapped in either direction. The physical core assignment must be made before beginning to swap a job in and cannot be canceled until the job is completely swapped out.

† An idle segment is a sharable high segment whose low segments are all swapped out. A dormant segment is a sharable high segment which no job is using.

7.4 The Swapper Cycle

The swapper operates on an overall cycle that is repeated as often as possible. This cycle typically requires many jiffies to complete. Hence, the swapper must operate as an asynchronous process under the control of the monitor, rather than as a simple closed subroutine. Every clock tick, if there is memory contention, the scheduler dispatches to the swapper's single entry point in SCHED1. The swapper proceeds through as much of its cycle as it can, and then returns to the scheduler. A number of flags are set up to allow the swapper to "remember" actions completed on earlier calls. Whenever the swapper reaches a point at which it cannot immediately continue, it exits, and attempts to continue on the next clock tick. Although the entire swapper cycle normally requires many clock ticks, it is best, initially, to look at the cycle as a continuous process.

A flow chart of the entire swapper cycle appears in the supplement. The first step of the cycle is to choose a job to swap in. The job number is stored in FIT, and remembered there. All the following actions are directed toward the goal of getting this job into the core. If there is no job to swap in, the swapper checks for jobs that must be swapped out in order to expand (JXPN bit set). If there is such a job, it is swapped out.

Once a job is chosen for swap in, all further actions have the objective of creating enough room for the size specified by its in-core image size in JBTIMI. The first step is to ensure that the total amount of available core (CORTAL) exceeds the amount required. If CORTAL is less than the amount required, another job must be forcibly swapped out. If CORTAL is greater than or equal to the amount of core required, the required assignment is made without forcing any other job to be swapped out.

If there are enough free pages, core is assigned. If not, dormant and idle segments are deleted until the necessary pages are obtained. The core assignment routine is called to assign physical core to the job chosen to be swapped in. Then an I/O request for disk service is set up to read in the core image. When the transfer is complete, the necessary housekeeping is performed and the swapper cycle begins again.

7.5 Choosing the Job to Swap Out

The relative priorities of jobs for being swapped out are specified by the scan table OSCAN. When a job must be swapped out, the swapper scans the in-core queues, according to OSCAN, looking for jobs eligible to be swapped. A job is rejected for any of the following reasons: its in-core protect time has not expired, it is locked in core, or it has I/O in progress. A tally of the amount of core that has been checked is kept. When this tally reaches the amount needed, scanning stops and the first available job found is chosen for as the one to swap. Note that if jobs were swapped strictly according to priority, all of the jobs examined would have to be swapped.

For example, if 20 pages are needed and the jobs in the order specified by OSCAN are set up as follows:

Job #	Memory Size
14	4P
20	Locked in core
6	12P
12	8P
10	24P

Scanning would stop at Job 12, and Job 14 would be chosen to be swapped out. If the queues were unchanged after Job 14 was swapped, Job 6 would be chosen on the next scan.

7.6 Swapping I/O

All I/O for the swapper is performed by the Disk Service Routine according to requests set up by the swapper. The following discussion of Swapping I/O assumes that the Disk Service Routine is a "black box routine" that writes specified physical disk blocks from specified core areas. The swapper sets up and submits a request for transfers. The Disk Interrupt routine clears a flag, SQREQ, each time a transfer is completed.

Swapping space is reserved on a per unit basis when the disk is initialized. This space is marked in use as far as the rest of the system is concerned. The swapper maintains a Storage Allocation Table (SAT) for each unit on which it has space. It uses one bit in a SAT to represent 1K or 1024 words of disk space. A SAT bit is set when a block is used for swapping out a job and cleared when the job is swapped in.

Swapping space may be reserved on any or all disk-like units or disk packs. Since the actual I/O is handled by the disk service, the swapper logic is independent of the type of unit being used for swapping. Each unit having swapping space is assigned a class for swapping at the time the space is reserved. The class indicates the priority of the device for swapping and (normally) assigns the fastest device to the lowest numbered class. When the swapper needs to find space to write out a segment, it starts with the lowest class:

1. The swapper scans through its SAT for each unit in the lowest class, and if it finds a large enough hole, it uses the corresponding area for the transfer.
2. If it cannot find a large enough single hole, it searches for several holes that, together, provide the required amount of space.
3. If it finds enough space, it writes the core image as several fragments within that class. If there is not enough space altogether within a class, it tries the next class.
4. If no single class has enough space, it fragments the image, across classes.

The purpose of multiple classes is to distinguish between devices of widely varying speed (for example, factors of 10). Better performance is achieved, under 7.02 and 7.03, with a single class of swapping units judiciously placed on the available channels.

A sharable segment is normally left on the swapping device if there is a copy in core, even if the segment is dormant. This prevents having to write the same segment out to the swapping device at a later time. If there is not enough free swapping space, one of the unnecessary segments is deleted from disk and rewriten later.

7.7 Examples

The following examples show how jobs are swapped under some highly simplified loading conditions. Specific values are used for CPU time requirements, but average or "expected" values are used for delays. The average delay time more closely approximates the effect of the delay when actual CPU times vary randomly about the specified average value. The additional complications that can result from the introduction of user I/O, competition for sharable resources, and any transient conditions, are not considered, even though it is quite possible that in actual operation these might have significant, or even dominant, effects.

7.7.1 Small, Interactive Jobs

A small, single disk pack system runs 20 jobs. The available user core is 200 pages. Each job runs in 20 pages and requires an average of one-tenth second (100ms) of CPU time for each interaction. Each job does only a small amount of I/O to disk and TTY. Hence, swapping and CPU time are the primary considerations in computing expected performance.

Normally, the swapper chooses a job to swap in on the next clock tick after a user inquiry. Thus, there is an average 8ms (1/2 clock tick) delay between the user action and the swapper's initial actions. Given the following assumptions:

1. The core is filled with jobs that are available for swap out (inactive jobs that have no I/O in progress), and
2. The swapper can immediately submit its output request to the Disk Service,

A job is swapped out to make room for the next job swapped in. Given the following assumptions:

1. The disk access arms are randomly positioned, and
2. The swapping space is in the center of the pack,

The seek takes an average of 28ms (407 cylinders).

The transfer is initiated immediately upon completion of the seek. There is an 8ms (1/2 rotation) average latency, or rotational delay time, followed by a 56ms transfer (5.6 microseconds per word $*$ 10K [20 pages]). The swapper is not called again until the next clock tick, giving an average 8ms delay. On the next call, the swapper starts the input transfer, and it is assumed that this transfer takes 28ms seek time.

There is another 8ms latency, 56ms transfer, and 8ms delay until housekeeping for swapping in the active job. The job then runs for 6 consecutive clock cycles (100ms). Under ideal conditions (user requests are regularly spaced, with one arriving every 1/2 second) each user job is completed within 308ms. This appears to be an instantaneous response. There is then a 192ms (500-308) period while the system is idle and waiting for the next user request.

If the same sequence of actions is assumed for swapping each job, and if all 20 users requested action at the same time, the last request finished requires 5.7 seconds to complete.

Operating as described above, the system repeats a cycle that takes an average of 500ms to complete. Of this 500ms, 100ms of CPU time is taken for the user job. Of the Null job time, 208ms is counted as lost time and 192ms as idle time. Hence, the system shows 20% user program CPU, plus 42% lost, plus 38% idle.

How many such jobs can the system support? At saturation, the running of one job is completely overlapped by the swapping of another. Assuming the same sequence of actions for swapping, a maximum of one inquiry every 209ms, or 48 users, is possible. (Note, however, that any disk I/O done by user programs increases swapping time, and decreases the maximum request rate accordingly.)

7.7.2 Large, CPU Bound Jobs

Ten users start compute bound jobs that require 80 pages each and 18 seconds CPU time. The system has 100 pages of available user core. Hence, one job is in core while the other nine are swapped out, so the jobs are processed and swapped on a round robin basis. It is assumed that all jobs circulate entirely within PQ2 and the ICPT for each job is greater than 6 seconds.

The scheduler chooses the job in core to run. Since this job is the only job in core, it is the only runable job. Hence, it runs in the PQ2 quantum run time, 6 jiffies, and is requeued to the back of PQ2. During this time the swapper picked a swapped out PQ2 job to swap in but found no eligible jobs, (jobs with expired ICPT), to swap out. Since the ICPT is assured to be greater than 6 seconds, this NOFIT condition persists until the swapper becomes frustrated. During this time the job in core is run continuously, regardless of position within PQ2. After the Frustration timer has gone off, the job in core is eligible for swap out. Then, the original job chosen for swap in is brought in.

In this example, an RP06 disk is used for swapping. There is an average 27ms seek, 9ms latency and 224ms transfer time. There is an average 8ms delay until the next clock tick. The save sequence occurs on the input transfer. Hence, changing jobs requires a total of 536ms for swapping, both out and in.

The system reports a cycle in which a job gets 6 seconds CPU time and then 536ms are spent swapping. Each job runs for one such cycle then is swapped out for nine such cycles.

Therefore, each job runs just once, accumulating 6 seconds CPU time every 65.36 seconds elapsed time. Three such passes are required to accumulate 18 seconds CPU time for each job, with all jobs completing after 196.08 seconds. During this time the system has lost .536 seconds out of 6.536 seconds elapsed, or about 8.2% lost time.

7.8 Swapper Data Base

SPRCNT
Contains the number of jobs that have been selected for swapping.

SWPCNT
Contains the number of jobs that finished data transmission and are waiting for final cleanup at the scheduler level.

SQREQ
Contains the number of data transmissions awaiting the swapper. This is the number of fragments plus the number of page I/O requests.

PAGTAB

A table containing one word per page of physical memory. Whereas it once had many uses, it is now used only as a memory management tool. It contains a linked list of pages for every segment currently in core, not necessarily in the same order as they are in the segment address space (if the job has "gone virtual" and has only a working-set in core). It also contains a linked list of pages not in use.

MEMTAB

Also one word per page of core. It is used during swap or page requests in conjunction with the UPT to keep track of the location of pages end up in the swapping area and which page to transmit next. On the KL10, MEMTAB is in Section 3. The first three bits of a MEMTAB entry are flags:

Bit	Symbol	Meaning
0	MT.LEF	Last entry in fragment chain
1	MT.GPB	Return swapping space when I/O done in IP queue
2	MT.IPC	IPCF page,,address of packet + .ICPFI in MEMTAB

The format of bits 3 through 35 of the MEMTAB table entries differs for the status of the page. For a page that is being transmitted to or from disk, the entry contains the disk address in bits 15-35.

The MEMTAB entry for a page in a paging queue contains the job number in Bits 5-14 (MT.JOB). For an IPCF page, when the page is in the IP queue, the high-order 3 bits of MT.JOB contain the IPCF header address (remaining 15 bits of address of IPCF header are stored in PT2TAB).

For a page in one of the IN paging queues, the remainder of the word is:

Bit	Symbol	Meaning
24-35	MT.VPN	Section-relative virtual page number (page is a job page)
22-26	MT.VSN	Section number
18-35	MT.IPA	Address of IPCF header block for this IPCF page

JBTSWP

Contains information when swapping segments. Each word in the JBTSWP table is formatted depending on the setting of Bit 0. If Bit 0 is set, a core address is in bits 18-35. If Bit 15 is clear, bits 15-17 contain the index to the unit number, and bits 22-35 contain the first logical K of swapping space allocated for the segment on the unit.

Bit	Symbol	Meaning
0	FRGSEG	1 if low or high segment is fragmented on the swapping device
15-35		Disk address, if Bit 0 is off. Core address in Bits 18-35 of fragment table if Bit 0 is set.
15-17	JBYSUN	Index to unit number in SWPTAB
22-35	JBYLKN	First logical K on the unit

After allocation, and before the swap is queued, if the segment is a low segment, LH of JBTSWP becomes the swapping pointer for the UPT.

Fragment table entries are put into four-word chunks of monitor free core and have the same format as JBTSWP entries, terminated by a zero word. If there is not enough room in the current four word entry, there is a fragment pointer (bit 0 on) to the next four word block. The bits in the fragment entry are:

Bit	Meaning
0-17	Number of pages in fragment
20-22	Unit index into SWPTAB
23-35	Logical page within unit where fragment starts

There are 3 parallel tables used by the swapper to keep track of the requests currently under its control (swapping and paging). They are SWPLST, SW2LST, and SW3LST, and each is SLECNT long. Giving more than SLECNT jobs to the swapper results in a STOPCD.

JBTIMI

Contains the number of pages to allocate for non-zero section page maps when swapping the job in, number of pages which are currently allocated to non-zero section page maps, and the number of physical pages in the user portions of the job.

JBTIMO

Contains the number of physical pages in a swapped-out job (that is, the number of pages on disk).

SWPLST

Keeps track of the progress of the swapping or paging I/O for the job it is assigned to. SWPLST is used in conjunction with MEMTAB. Its format is:

Bit	Symbol	Meaning
0	SL.FRG	Fragmented entry
1	SL.DIO	Direction of I/O (1 = out)
2	SL.SIO	Swapping/paging (1 = swapping)
3	SL.IOP	I/O in progress
4	SL.IOD	I/O done (this swap list entry is done)
5	SL.IPC	On if an IPCF page
6	SL.DFM	Don't find me (used to keep FNDSLE from finding this entry)
11	SL.CHK	Swapping checksum error
12	SL.ERR	I/O error
13	SL.CHN	Channel error

For a contiguous entry, bits 14-26 contain the starting pysical page number (used as an index into MEMTAB), and bits 27-35 contain the number of pages.

For a fragmented entry, bits 18-35 contain the address of the fragment table.

SW2LST
Saves the original SWPLST entry during the swap because it is needed for cleanup, but the SWPLST entry is modified while I/O is in progress.

SW3LST
Contains the job number, if a low segment is being swapped; contains the job number, high segment number, if a high segment is being swapped.

7.9 Swapping High Segments

High segments are never chosen directly to be swapped in or out; job numbers (low segment numbers) only are considered when looking for jobs to swap in or out. High segments are swapped as appropriate for the low segments with which they are associated.

A high segment is swapped out along with the last low segment using it. This means that non-sharable high segments are always swapped at the same time as their low segments. A sharable high segment is never swapped if there are any jobs still in core using it. Also, sharable high segments are normally written out to the swapping device only once. If there is a copy of a write protected high segment on the swapping device, we do not have to write it again. Hence, commonly used sharable high segments are on the swapping device all day. If the last job using a given high segment is swapped out, the high segment becomes idle. As an idle segment, it is subject to being deleted from core memory if the the space it occupies is needed. However, a copy is kept on the swapping device before deleting the core image. A high segment is swapped in whenever a job using it is swapped in and there is not a copy of it in core.

Essentially, the same code is used to swap high segments as is used to swap low segments. The routines that swap jobs in and out look at the numbers in FIT and FORCE as segment numbers and swap the specified segment in or out. A job is always choosen to swap in. The low segments are swapped in first. If the job has a high segment to be swapped in also, the high segment number is put into FIT and the routine is repeated.

When a job is choosen to swap out and, if it has a high segment to write out, the low segment number is stored and the high segment is swapped out first. After the high segment has been swapped out, its corresponding low segment is put into FORCE and swapped out. The reason for swapping the high segment first is because a segment cannot be written out until all its I/O stops. A sharable high segment cannot have I/O in progress, and therefore can always be written out immediately. Hence, the high segment can be swapped out while waiting for for I/O to stop in the low segment. Funny-space pages are swapped in or out with the low segment.

7.10 Complications and High Segments

A number of complications can arise in swapping jobs with sharable high segments. Several cases are identified and discussed below.

1. Low Segment in Core, High Segment Swapped Out

This does not usually happen because the high segment is not normally removed from core until the last job is swapped out; it is then brought back along with the first low segment that uses it. However, it is possible for a job to do a RUN for a swapped out high segment. This could occur while the low segment is set up in core and the high segment is swapped out. In this case, the low segment is marked as swapped even though it is in core. When the low segment is choosen to swap in, and seen to be already in core, and proceed to swap in the high segment. This problem can also occur on a GETSEG UUO.

2. Zero Length Core Images .

   A segment can exist in that it has a number and is recognized as a job or high segment but have no core allocated. This is quite common when a job or high segment initially expands from zero to a non-zero size. If the core management routine cannot make the requested assignment in core, it marks the job to be swapped out and sets the in core image size to the size requested. The swapper eventually chooses the job to swap out. Upon finding the the segment to be swapped out is of zero length, it bypasses the output process and simply marks the segment as swapped out. This gives a zero length segment on the swapping device. When the job is chosen to be swapped in, the swapper finds enough free pages of the size specified by the in core image size and assigns it to the segment being swapped in. The assignment routine sets the entire area to zeros. The swapper's input routine detects that the swapped out image size is zero and bypasses the process of reading in the core image. The segment is then marked as swapped in and is available for use.

3. Idle Segment Not on Swapping device

   This does not normally occur because the high segment is written on the swapping device along with the last low segment using it. However, it is possible that the last low segment will not be swapped out. The could happen if the user runs another program or if the program does a GETSEG UUO , detaching from the high segment and leaving it without attached low segments in core. When an idle segment is chosen to be deleted, it is checked for this condition and, if necessary, the segment is forcably written out before deleting the core image.

Chapter 8
UUO Processing

UUOCON is the monitor module that executes Unimplemented User Operations (UUOs) for the user. It performs three functions: UUO pre-processing, dispatch to the correct service routine, and exit. This chapter explains how each operation functions and how to add new UUOs.

In order to introduce the topic of crash analysis, this chapter explains how illegal UUOs are handled.

8.1 Description

The function of UUOCON is to service those unimplemented or illegal operation codes that are trapped by the processor microcode to locations 424 (.USMUO), 425 (.USMUP), and 426 (.USMUE) of the current job's UPT. These are opcodes 000_8 through 077_8 and, in user mode, the device I/O instructions (7*xx*) outside the range of 740_8 through 774_8 (reserved for customers), the HALT instruction (JRST 4,), and the JEN instruction (JRST 10,). Any unassigned operation code causes the microcode to trap.

For the purpose of this discussion the operations of UUOCON can be divided into three sections:

1. Operator-independent pre-processing and dispatch
2. Operator service (operator-dependent algorithms)
3. Exit routines

8.2 Operator pre-processing and dispatch

Pre-processing includes the following:

- Switching to the exec AC block (block 0).
- Setting up a push down list in the job's UPT (for use by the monitor during UUO execution).
- Saving the return PC on the stack.

- Loading accumulators with information to be used by the operator service routines.
- Dispatching to the proper service routine.

8.2.1 Special registers

An important pre-processing function is to place information about the UUO-issuing job into certain accumulators and index registers before dispatching. The tabler below lists these registers and their contents.

Registers	Contents
P	A pushdown pointer to a list in the User's Process Table 133_8 words long. One of the first items placed in this list (JOBPDO) is the user's return; that is, a copy of the PC word in location 425 of the UPT (.USMUP).
R	A copy of the contents of JBTADR: job size,,first page of user virtual address space. The first page of the user's virtual address space, which contains the Job Data Area, is accessed through executive virtual page 773_8. Unless a job is locked in core, R contains (as does JBTADR) the job size in the LH and 773000_8 in the RH.
M	A copy of the programmed operator so that operator service can refer to the effective address (E) indirectly through AC M.
P1	A copy of the AC field of M. P1 could hold the number of a user I/O channel, as in the case of input and output operators.
F†	A copy of the USRJDA (protected .JDAT) entry for this software channel. This register contains 0 if the channel is unassigned. If the channel is in use, the left half of this word had status bits indicating what UUOs have been performed for the device; the right half contains the address of the Device Data Block (DDB).
S†	A copy of the DEVIOS status word for the device on this channel.
T4†	A copy of the DEVSER word for the device on this channel. The left half of the word contains the address of the next DDB in a chain of all such blocks; the right half contains the address of the device dispatch table. Each type of device has a unique device dispatch table providing pointers to the UUO-level code for various I/O functions on that type of device.
W	Contains the address of the job's Process Data Block (PDB).

†These registers are pertinent only to Input/Output UUOs, and are loaded when an AC field (P1) corresponds to an assigned I/O channel.

8.2.2 Functional description

The following table lists pertinent routines and contained in the operator-independent pre-processing and dispatch section of UUOCON:

Label	Function
MUUO (COM-MON)	After switching to the EXEC AC block from user mode, a flag bit of the trapped PC word is tested to detect whether the call is from the monitor (as in a GET command) or from the user. If from the monitor (that is, the processor had been in executive mode), certain ACs have been set up and a portion of the UUOCON coding can be skipped; control goes to UUOSY1. If the call was from user mode, the contents of R, J, and P are established. If the job doing the UUO is the null job and the UUO is a WAKE UUO, the Scheduler is called. This occurs only on multi-processor systems when one processor stops running a runable job while another processor is running the null job. This event forces the CPU running the null job to stop and select the runable job to run.
UUOSY1	This routine in UUOCON loads register M with the UUO itself and J with the job number. If the UUO has an opcode of 0, control is transferred to UUOERR in ERRCON. The return PC word is taken from offsets 424, 425, and 426 in the UPT and placed on the stack to ensure that it is not overwritten in the event a UUO is executed by the monitor itself. The opcode is checked for a value greater than 100 (illegal at this point). If the value is legal, accumulator P1 is set up. If there is a device on this channel, F, S, and T4 are set up. If no device has been assigned to this channel coincident with this UUO's AC address, the routine NOCHAN is entered. Otherwise, if this UUO is indeed an I/O operator of opcode 72 or greater (long dispatch I/O UUO), then routine DISP1 is entered. DISP0 is entered directly for non-I/O UUOs or I/O UUOs between codes 55 and 71 if the channel is found to be assigned.
DISP0	This code obtains an address from the half-word dispatch table using the opcode as an index. Prior to dispatch, routine UUOCHK in module VMSER is called to verify that the UUO arguments are in core; if not, control is transferred to the PFH, which pages in the page or pages containing the UUO arguments. This approach is taken to prevent a page fault from occurring as a result of a memory reference made by the monitor. If the UUO is from user mode, the service routine is dispatched to by a PUSHJ, which puts the address of the user exit routine on the list as it jumps. If the UUO is from the monitor, then the desired address is already on the list and is left undisturbed when dispatching to the service routine.
NOCHAN	This routine calls DISP0 if the UUO is from the monitor or if it was from the user and is not an I/O operator. If the UUO is a CLOSE or RELEASE operator, the successful return exit is called. Otherwise, the routine IOIERR is entered to type the message "I/O to Unassigned channel . . . " and stop the job.
DISP1	This routine fakes a successful return to the user if the UUO was a long dispatch UUO and the device service routine does not have a long dispatch table. (This is an important concept in making user programs device independent; for example, it enables a LOOKUP to a physical paper tape reader tape to be successful.) If the device service routine is capable of performing long UUOs, the dispatch routine DISP0 is called.

8.3 Operator service

Operator service routines perform the algorithm designed for the particular UUO, allowing the user to:

- Receive information about the system
- Alter the operation of the system concerning his job

- Communicate with the input/output devices

A few specific examples are included in this chapter module to demonstrate the information flow between the three sections of UUOCON and the user's job. Input/output UUOs are described in the chapter on Input/Output Service, Chapter 9

Communication between the user's program and the monitor takes place in the following manner: Information is passed to the monitor through the user AC block or through argument lists somewhere in the user's address space. These lists, as well as the actual arguments themselves, may be in a page of address space that is in core or paged out.

The primary method of referencing UUO arguments, given a user virtual address, is by the use of the Previous Context Execute instruction (PXCT). However, before making memory references to a user virtual address, two conditions must be verified: first, that the address is a valid virtual address for that user's address space and second, that the page containing that address is in core. There are also some locations in the Job Data Area that need to be protected, and some references to user ACs that must be prevented.

There are three address checking routines in UUOCON which are called from many UUO service routines:

Label	Function
UADCK1	Takes successful return if the address being checked is an AC, otherwise falls into UADRCK.
UADRCK	Called only from UUO level. However, because the address being checked may be referenced from interrupt level sometime in the future, AC references are illegal. References to locations in the protected part of JOBDAT and to pages that do not exist are also rejected. If the reference is to a paged out page, the job's page fault handler is invoked to get the page in core before proceeding. If the reference is to the high segment, the error return is taken. If an illegal address is encountered in either UADCK1 or UADRCK, the job is stopped and the message "Address check . . . " is typed on the user's terminal.
IADRCK	This routine is called from interrupt level primarily for I/O buffer address verification. References to ACs, the protected part of JOBDAT, non-existent pages, and pages not in core are all illegal.

Once the user virtual address has been verified, the monitor makes the memory reference through the use of the PXCT instruction. In some instances the PXCT instruction appears in-line with the code that called the routine to verify the addresses. The EXCTUX, EXCTXU, and EXCTUU macros are used to generate the appropriate PXCT instruction. In other instances, calls are made to other routines to complete the memory reference. Consider the following four cases:

1. Fetch the contents of the EA of the UUO into T1.

   The routine GETWDU in DATMAN is called. After some rechecking of the user virtual address, the code generated by the following macro is executed:

```
EXCTUX  <MOVE T1,@M>
```

   The interpretation and translation of the contents of M as a user virtual address is done strictly by the hardware due to the execution of a PXCT instruction in executive mode.

2. Store the contents of T1 into the EA of the UUO.

   The routine PUTWDU in module DATMAN executes the code generated by the following macro:

```
EXCTXU  <MOVEM S, @M>
```

   where S had previously been loaded from T1.

3. Get an argument from the AC referenced by the UUO itself.

   Routine GETTAC in DATMAN extracts the AC number through use of the PUUOAC byte pointer and executes the GETWDU routine.

4. Store the contents of T1 into the AC referenced by the UUO.

   Routine STOTAC in DATMAN uses routine PUTWDU to accomplish the desired results. There is an alternate entry point, STOTCI, that accomplishes the same result as STOTAC but takes a skip return.

In returning to the user, it is possible to skip one or more instructions that followed the UUO or to give a skip or non-skip return to signify the success or failure of the operation. The UUOCON exit routine is designed to pass on to the user either a skip or non-skip return. If, at the level equal to that following the dispatch, a POPJ P, is used to exit, the user receives a non-skip return. If the sequence:

```
AOS     (P)
POPJ    P,
```

is used, a skip return occurs. This can be used to bypass one instruction following the UUO (a system routine, CPOPJ1 performs this action if called by a JRST CPOPJ1). If it is necessary to bump up the user's return by more than one, the routine must add the correct quantity to the correct entry on the pushdown list. (Recall that the pre-processor dispatch was not a PUSHJ if the original UUO was issued by the monitor). If, for example, two instuctions are to be skipped in return to a user-mode call, this sequence can be used:

```
AOS     -1(P)
JRST    CPOPJ1
```

To give the same return to a call from the monitor:

```
AOS     (P)
JRST    CPOPJ1
```

As an example, all operators that do not deal with some phase of input/output are invoked through the use of the CALLI UUO. For example:

```
CALLI   ac, 27          ;OP-CODE = 47

    or

RUNTIME ac,             ;EA =  27, serving as a code
                        ; or function within Op-code 47.
```

The referenced AC is loaded (by the caller) with a job number before the CALLI, and the CALLI returns with total running time (in milliseconds) of that job in the same AC.

The pre-processor routine of UUOCON sets up the standard accumulators and, using the UUO opcode (CALLI - 047), dispatches through UUOTAB to UCALLI. UCALLI picks up the UUO effective address and uses it as an index into UCLJMP to find the dispatch address for the specific CALLI, RUNTIM. This argument is used to effect another dispatch to the routine JOBTIM, which gets the appropriate run time (for the job specified in the caller's AC) and stores it in the user accumulator.

When entered, the JOBTIM routine checks the contents of T1 for a valid job number and uses it as an argument to the FNDPDB routine to find the Process Data Block (PDB) for the job. The desired time is extracted from the PDB, converted to milliseconds, and placed into T1. A JRST STOTAC causes this result to be stored in the user's accumulator, now addressed by M, and returns to the UUOCON exit routine.

8.4 Exit routines

The exit routines (normal or error) perform the setup necessary to return to the calling program or, in the case of errors, produce error messages and appropriately alter the status of the job. One important function of the normal exit routine is to check if the clock went off while the UUO was being executed before returning to the calling program. A software interlock between the Scheduler and UUOCON allows a UUO (which is, after all, one instruction) to run to completion before the current job is stopped. The normal UUO exit routine calls the Scheduler if the clock ticked during the UUO processing.

8.4.1 Error exits

Hard error exits, which do not allow a return to the user, occur when a UUO opcode is illegal or an address supplied by the user is illegal. An unimplemented UUO in the range 40_8 through 77_8, or a UUO of 0 all stop the job with the error bit on (cannot continue) and print "Illegal UUO at . . . ". An illegal opcode (such as a DATAI in user mode) causes the job to stop with the error bit set and the message "Illegal instruction at . . . " to be printed. In user mode, the HALT instruction is treated like an illegal MUUO. It causes the job to stop (without actually executing the instruction), types "HALT at . . . ", but does not set the error bit. Thus, the CONTINUE command does function after a user mode HALT. (This can be useful for debugging.)

When an illegal address is detected by a non-I/O UUO, the UUOERR routine is called to print the message noted above ("Illegal UUO at . . . ") and puts the job into an error stop. When a UUO is associated with a particular device, ADRERR may be called. ADRERR prints "Address check for device . . . " and results in an error stop condition.

8.4.2 Normal exits

If the original UUO is issued by the monitor, the pre-processor dispatch is by a JRST rather than by a PUSHJ. The service routine's last POPJ bypasses the user exit routine and goes directly back to the monitor code following the call.

If the UUO is from the user, the service routine's terminating POPJ returns to location USRXT1-1 (no-skip return) or a JRST CPOPJ1 returns to USRXT1, which passes a skip return to the user by adding 1 to the address on the pushdown list.

Label	Function
USRXIT	This routine checks to see if the user has typed a CTRL/C or if the clock has ticked (software interlock), or if the system wants to stop this job (to swap it, for instance). If none of these conditions exists, the user's accumulators are restored and control is returned to his program. Otherwise, the Scheduler is called (USCHED) to take appropriate action. If the user's job continues in the future, control comes back here to restore the user's accumulators and continue the job.

8.5 Adding a programmed operator

There are two ways to add a new UUO function to the monitor. One is to use a previously unused opcode (42 through 46 for customers, 52 through 54 for DIGITAL). The other is to add an additional CALLI. Adding customer defined CALLIs with negative arguments is the preferred technique.

8.5.1 Adding a new operator

1. Edit the new code into the source file for UUOCON. If it is desired to make this routine a conditional feature, it may be enclosed in conditional assembly brackets preceded by a symbol like the feature test switches currently in use.
2. Edit the CALLI UUO dispatch table macro definition, CNAMES, to include the name of the UUO, dispatch address, and legality bits. For instance, to add a new CALLI called UDUMP, change the dummy entry for CALLI -2 in the CNAMES definition. Conditional assembly can be used to set up the dispatch table entry if conditional assembly is used with the routine itself.

   For example, add this code to UUOCON:

   ```
   IFN FTDMPU,<
   UDUMP:  (subroutine)
   >
   ```

   Then add the appropriate entry to the CNAMES macro:

   ```
   IFN FTDMPU,<
   X UDUMP, UDUMP
   >
   IFE FTDMPU,<
   X CPOPJ, CPOPJ##
   >
   ```

   In this example, the routine assembles, and the address of UDUMP is added to the dispatch table if the feature switch FTDMPU is non-zero.
3. In preparation for assembling the new UUOCON, include the correct feature test switch in the F.MAC (software features) source file by running MONGEN.
4. Assemble F.MAC first, then UUOCON.MAC.
5. Use MAKLIB to replace the old version of UUOCON with the new one in the library file to be used in building your system. (See the MAKLIB User's Guide for details).

6. Build a new monitor, using this new library file, according to the procedures in the Software Installation Guide.

The new monitor call is available to all users at your site, as long as this version of the monitor is used. Of course, this new call can be patched into the monitor tables and code in executable form. That, however, presents complications leading to exhaustion of patching space and difficulty of documentation.

Chapter 9

I/O Introduction and UUO-Level Routines

User programs perform I/O using a select group of UUOs such as OPEN, INIT, IN, OUT, CLOSE, and FILOP. Processing each UUO includes two phases: device-independent processing and device-dependent processing. This chapter discusses the first of these two phases, explaining what each UUO does at the device-independent level.

9.1 Introduction

Input-output handling by the DECsystem–10 monitor is based on the objectives of device independence and modularity of code. Any user program should be able to use any device capable of meeting its requirements. The user should not have to specify the device until run time and should be able to specify different devices for different runs as conditions require.

Modularity of code plays an important role in meeting this objective. Code that is device independent is separated from various modules of device-specific routines. Modularity also makes it convenient to tailor a monitor for any specific configuration from a single set of source files. The systematic manner in which the I/O modules are organized makes it possible for an installation to add code to handle a special device without changing the existing code. The new code can take full advantage of all device-independent routines in the standard system. User programming for the special device can follow the same device-independent principles and protocols that apply to standard devices.

This module discusses the following topics:

- Hardware principles that apply to I/O processing.
- Organization of the I/O-processing code.
- Functions performed by various modules.
- Device-independent functions performed within the UUO processor.
- Device-service routines, which perform the device-dependent functions at UUO level
- Macros that generate configuration-dependent code.

- Timing problems that must be considered by device-service routines and techniques for solving these problems.

9.2 Hardware Principles

All I/O transfers are done by DATAO/BLKO or DATAI/BLKI instructions. Each instruction addresses a specific device (or controller) by a device code in bits 3–10. Upon execution of the instruction, a single word is transferred between core memory and a register of the device. Execution of the instruction requires only a few microseconds, after which the CPU continues to execute the program. The device, however, is not ready to accept another instruction for a relatively long time. When it is ready, the device requests a priority interrupt.

There are two different types of I/O devices: I/O-bus devices and data-channel devices. I/O-bus devices require the I/O bus for each word transferred. Data-channel devices require the I/O bus only for initiating the transfer.

I/O-bus devices cause an interrupt for each word (or character) to be transferred for slower devices. An entire interrupt routine is executed on each interrupt. This interrupt routine checks: (1) for reaching the end of the buffer and (2) the device status for error conditions. Also, it normally requests the next transfer. The faster devices, DECtape and non-DMA magnetic tape (TM10A), also cause an interrupt for each word, but the interrupt results in execution of only one instruction, a BLKO or BLKI.

At the beginning of a block, the BLKx instruction is set up at the interrupt location, and a pointer-counter word is set up. On each interrupt, the transfer is performed, the pointer counter is incremented and tested, the interrupt is dismissed, and control is returned to the interrupted routine. If the counter expires, the interrupt remains in effect, and the next instruction after the BLKx is executed. This instruction calls an interrupt routine that does the necessary "housekeeping" and sets up the next block transfer.

The BLKx devices are assigned two priority-interrupt levels. One of these, which is normally a very high-priority level, is used for the BLKx instruction on normal data interrupts. A lower level is used for error interrupts. The lower-priority channel is called the *flag channel.*

The TM03 magnetic tape controller, and all disk controllers, use a data channel to access memory directly without interrupting the CPU or using the I/O bus. A single instruction initiates the transfer, and the controller requests an interrupt when the entire block is finished. Although these devices have very high data rates, their interrupts are infrequent. Disk controllers of the RH20 type usually automatically interrupt on level 3, using *ac* block 3. Actual disk transfers are discussed in Chapter 11.

9.3 Organization of I/O Routines

All device-independent routines are contained in UUOCON. These include outer-level routines to handle all I/O UUOs. The UUO decoder dispatches to these routines, with various "global" *ac*'s set up, and the I/O routines return to the UUO decoder for final housekeeping functions before it returns to the user program. Also included in UUOCON are subroutines to perform various device-independent functions for device-dependent routines such as the routine to advance buffers.

All device-dependent code for one device is normally included in a device-service routine for that device. There is only one service routine for a given type of device, regardless of the number of such devices or units in the configuration. These service routines work for any possible number of units. Therefore, any configuration-dependent code is included in COMMON (or COMMOD for disk).

The Device Data Block (DDB) for a device is normally included in the service routine. Where there are separate controllers for several devices of the same type, only a small amount of code depends on which controller is being used. The DDB for each device is defined in the driver module along with the interrupt code that is unique to the device. At system initialization, the monitor creates a working copy of the DDB from the prototype in the driver. In addition, device-specific interrupt service code is built and linked into the CONSO skip chain or set up for vectored interrupts, depending upon the type of device. Refer to Chapter 11 for more detailed information on device detection.

Also included in each device-service routine are routines to perform device-dependent functions for each UUO. The entry points for these routines are put into the Device Dispatch Table, whose base address is included in the DDB. These routines are called only as subroutines from the device-independent routines in UUOCON.

Finally, included in each service routine is the interrupt routine for that device. The interrupt routine gets control when the corresponding device has caused a priority interrupt. It must perform the actions required by the device and then dismiss the interrupt without interfering with the interrupted process.

9.4 Device-Independent Functions

This section examines the functions performed by the device-independent routines in processing buffered I/O. It begins with a general discussion of actions taken by each UUO routine. First, it follows the steps taken by a program reading a file from an arbitrary device. Then, it follows the steps taken by a program writing a file. This section concentrates on major concepts; descriptions are not complete in every detail. However, once you are thoroughly familiar with the material in this section, you should be well prepared to go to the listings for complete information.

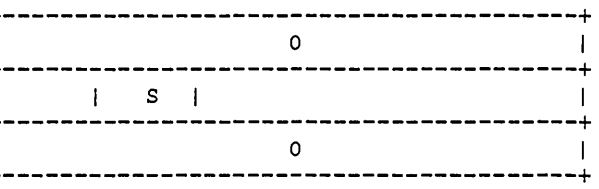
9.4.1 INIT or OPEN UUO

The INIT (or OPEN) UUO is the means by which the user program specifies a device that it wants to use. The main function of INIT is to find, or in some cases set up, a DDB for the specified device. The DEVSRC routine performs this function. It first searches for a DDB assigned to the job having a logical device name that matches the argument of the INIT. If this fails, it looks for a DDB having a physical device name that matches the argument. Finally, if a generic device name (for example, MTA) is given, it looks for a DDB that is appropriate for that generic specification.

If a DDB is found, the ASSigned-by-PRoGram bit, ASSPRG, is set in the DEVJOB word. The DDB address is placed in the appropriate entry in the user's Job Device Assignment table (JDA), thus establishing the fundamental link with a software channel. The user bits in DEVIOS, including the data mode, are initialized according to the program's specifications. Assuming buffered-mode I/O, the byte size field in the buffer ring header is initialized

according to the data mode. The first and third words of the ring header are cleared. Hence, after INIT, the ring header appears like the one in Figure 9–1.

Figure 9–1: Buffer Control Block or Buffer Ring Header



If the specified device does not exist or is not available, the INIT routine gives an error return on the UUO decoder. There is no call to the service routine for device-dependent functions.

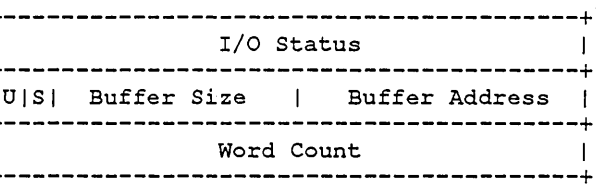
9.4.2 INBUF UUO

The user program may execute an INBUF UUO to ask the monitor to set up an input buffer ring of any specified number of buffers. If the program does not execute an INBUF UUO, the same routine in the monitor is called on the first INPUT UUO to set a buffer ring of *n* buffers, where *n* is the default for a given device type.

The buffer ring is set up in the user address space beginning at the job's first free location (.JBFF), and .JBFF is updated to point to the first location after the buffer ring.

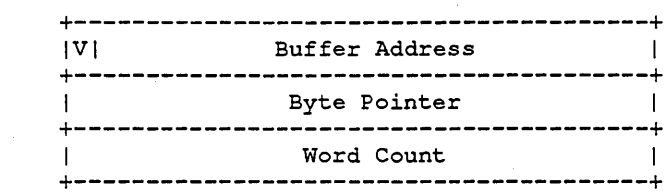
The length for each buffer (three words of header and a device-specific data area) is obtained by calling a device-dependent routine. Normally, the length, including buffer header, is two greater than the value obtained from the DEVCHR word in the DDB. An example of a buffer header block is shown in Figure 9–2.

Figure 9–2: Buffer Header Block



The buffer linkages are set up, and the use bit in each buffer header is cleared, making the buffer available to the *filler*. The first buffer address is put into DEVIAD in the DDB and into the first word of the user's buffer control block. The sign bit of this word in the buffer control block is set, indicating that the buffer ring has been initialized. Refer to Figure 9–3.

Figure 9–3: Buffer Control Block



No call to a device-service routine has been made at this point.

9.4.3 INPUT and IN UUOs

With each INPUT (or IN) UUO the user program asks that a new buffer of data be made available to it. The major functions of the UUO are to ensure that the next buffer is full, set up the necessary pointers and byte count in the user's buffer control block, and return to the user. If necessary, it initiates a request to the device. In the case of non-sharable devices, this, in effect, starts the device.

On the first INPUT after the INIT, if the buffer ring has not previously been set up by an INBUF, it must now be set up. A ring of buffers is set up by the same code described above for INBUF.

On all INPUT UUOs except the first, the use bit is cleared on the buffer previously available to the user. Clearing the use bit indicates that there is no "good" data in the buffer and makes the buffer available to the "filler." This informs the interrupt routine that it may continue writing into this buffer after it has filled the previous buffer.

One very important function of the INPUT routine is to start the device. Whenever an INPUT UUO is executed, the device is not running (IOACT is zero), and fewer than half of the buffers are full, the device-service routine is called to start the device. Since starting the device requires an actual I/O instruction, it is always a device-dependent function required by the INPUT UUO. Note that it is always necessary to start the device on the first INPUT UUO after the INIT. The function of INPUT is to give the user a fresh buffer of data. Therefore, the next buffer must be determined to be full before control is returned to the user. If the use bit on the next buffer of the ring is set, it is already full (available to the "emptier"). Hence, control can be returned to the user immediately without blocking. If the next use bit is not set, however, the job must not be allowed to continue running until the buffer has been filled. This is the function of a device-independent routine, WSYNC, which is called with a PUSHJ.

WSYNC informs the scheduler that this job is to go into I/O wait. (It sets the wait state code to IOWQ.) It sets up its own return address, from the push down list, as the restart address for the job and exits to WSCHED to perform a partial cycle. This job is put into an I/O wait state, and another job is chosen to run. The ADVBF routine gets the job out of the I/O wait state at interrupt level when the next buffer has been filled (I/O wait satisfied and JRQ bit set). The job continues at the next instruction after the PUSHJ to WSYNC. At this time, the next buffer is full, and control can be returned to the user program.

Note that, to the calling routine, WSYNC appears as a subroutine that can be called to fill the next buffer. From the monitor's point of view, it is simply a way of suspending the job, because it cannot immediately continue. Note also that the UUO processor must be reentrant at this point. Before the job continues running, any number of other jobs may execute this same code. Hence, when WSYNC is called, all variables must be in job-dependent storage locations. Specifically, all stored variables are either on the push-down list in this job's UPT or in accumulators.

When an end-of-file condition is recognized on the device, the IOEND bit is set in the DEVIOS word of the DDB. This is not the end-of-file bit that the user sees, however. The purpose of IOEND is to prevent an attempt to restart the device after it has been stopped at end of file. The user may have several buffers full of data yet to process when end of file is reached on the device.

After the last buffer has been given to the user, and he executes another INPUT UUO, the CALIN routine is called. This routine normally dispatches to the device-service routine to start the device. The CALIN routine does not attempt to start the device, because the IOEND bit is set. Instead, it gives an immediate return with IOACT still not set. When WSYNC is called, it does not put the job into I/O wait, because IOACT is not set. At this time, the user's end-of-file bit, IODEND, is set in DEVIOS. An error is returned to the user, and IODEND is his indication that he has reached end of file. This bit is never stored in the status bits of a buffer header; the user must check for it with a STATO or similar UUO.

9.4.3.1 CLOSE for Input

The CLOSE UUO performs the following functions:

- Restores conditions to the initial state (that is, after the OPEN), in which the system is ready to begin reading another file.
- Clears the use bit in each buffer of the ring.
- Sets the ring-header use bit, indicating that the ring has been set up but never referenced.
- Clears both end-of-file bits.
- Calls the CLOSE routine in the device-service routine to perform whatever housekeeping actions might be necessary. (Most devices other than disk do not require any special actions on input close.)

9.4.4 OUTPUT and OUT UUOs

In writing a file, the INIT and OUTBUF UUOs are analogous to INIT and INBUF in reading. Although the user thinks of the OUTPUT (or OUT) UUO as being issued for each buffer of data to be written, the function of the OUTPUT UUO is to provide the user with a buffer to fill.

The first OUTPUT normally does not write any data. The first OUTPUT is a "dummy" OUT, by which the user program asks the monitor to set up the buffer control block so that the user program can start putting data into the first buffer.

On each additional OUTPUT, the user is supplying one buffer of data and asking that a free buffer be made available to him. The full buffer is made available to the interrupt routine by setting the use bit (available to the emptier). If the device is not running, the device-dependent routine can be called to start it. Then the use bit of the next buffer is checked to see if the user can be allowed to put more data into it. If the use bit of the next buffer is not set, the buffer is empty (available to the next filler), and the job can be allowed to continue running immediately. If the use bit is set, there is still data in the buffer that must be written out. WSYNC is called to put this job in I/O wait until the interrupt routine restarts it. When the next buffer is free, it is cleared to zeroes, and the buffer control block is set up to allow the user to start filling the buffer.

9.4.4.1 CLOSE for Output

The CLOSE for an output file ensures that any remaining buffers are written out, keeping the job in I/O wait until all buffers' use bits have been cleared. The device-dependent routine is called for any device-dependent functions. Then the buffer ring is restored to its initial state. The buffer control block is also reinitialized with its use bit being set to indicate an unused ring.

9.4.5 RELEASE UUO

The RELEASE UUO countermands the OPEN. It first does a CLOSE on the software channel for both input and output, as appropriate, and puts the job into I/O wait until the device is inactive. The device service routine is called for any device-dependent actions. If the channel on which the RELEASE is being done is the highest channel in use by this job, the word is updated with the number for the highest channel in use, USRHCU (in the UPT). The ASSPRG bit is cleared in DEVIOS, and unless the ASSigned-by-CONsole-command bit, ASSCON, is set, the job number is cleared from DEVJOB. Hence, the RELEASE makes a device available for other jobs if it was not assigned by an ASSIGN command. If the device was disk, the DDB (which is set up by either the INIT or an ASSIGN command) is deleted when the job number is cleared.

Chapter 10

Device Service

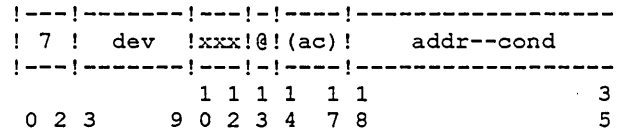
This chapter focuses on hardware-device service and the data structures necessary to communicate with the devices. To better understand the methods of device I/O, some knowledge of hardware principles and of data structure generation and linkages is necessary.

10.1 Hardware Instructions

The PDP-10 instruction set has several instructions reserved for device control and I/O initiation. These are referred to as the I/O or input-output instructions. They are not identical for all types of CPUs: KL10s and previous processors use one group of I/O instructions; KS10s use a different group.

10.1.1 KL10 I/O Instructions

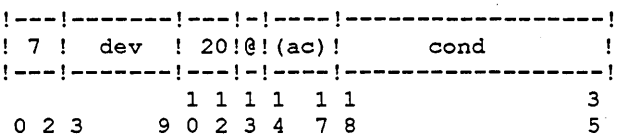
A KL10 I/O instruction is designated by 111 in bits 0-2; that is, its left octal digit is 7. The instruction format is:



Bits 10-12 are given as a two-digit octal number to select one of eight I/O instructions, which are described here in terms of their general effects for handling external devices. *Dev* addresses the device that is to respond to the instruction. The format thus allows for 128 devices codes, of which the KA10 uses the first two, the KI10 the first three, and the KL10 the first six. The first group of devices used by the processor are referred to as internal devices. Bit 13 (the indirect bit) and bits 14 through 17 (the index *ac*) are the same as all other instructions. Bits 18 through 35 contain either an address or a collection or mask of condition bits, depending upon the I/O instruction.

10.1.1.1 CONO

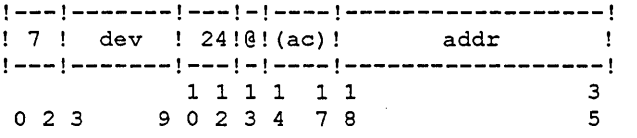
Conditions Out



Sets up device *dev* with the effective initial conditions *cond*. The number of condition bits in *cond* that are actually used depends on the device.

10.1.1.2 CONI

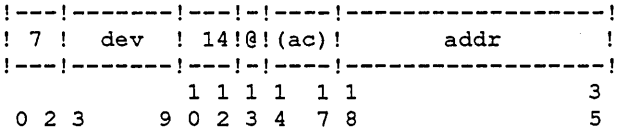
Conditions In



Reads the input conditions from device *dev* and stores them in location *addr*. The number of condition bits stored depends upon the device; the remaining bits in location *addr* are cleared.

10.1.1.3 DATAO

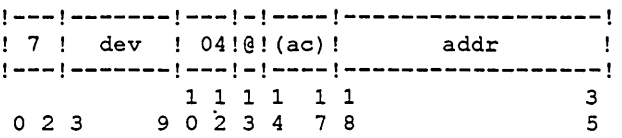
Data Out



Sends the contents of location *addr* to the data buffer in device *dev* and performs whatever control operations are appropriate to the device. The amount of data actually accepted by the device depends on such factors as the size of its buffer and its mode of operation. The original contents of location *addr* are unaffected.

10.1.1.4 DATAI

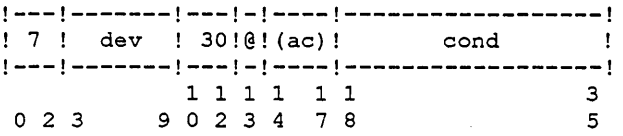
Data In



Moves the contents of the data buffer in device *dev* to location *addr* and performs whatever control operations are appropriate to the device. The number of data bits stored depends on such factors as the size of the device buffer and its mode of operation. Bits in location *addr* that do not receive data are cleared.

10.1.1.5 CONSZ

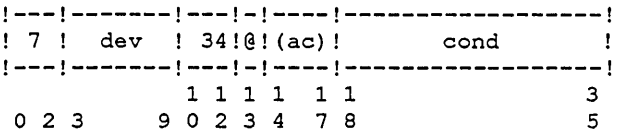
Conditions In and Skip if Zero



Tests the input conditions from device *dev* against the effective mask *cond*. If all condition bits selected by 1s in *cond* are 0s, skips the next instruction in sequence. If the device supplies more than 18 condition bits, only the right 18 are tested. Condition bits in the left half word can be tested by reading them with a CONI and then using a test instruction.

10.1.1.6 CONSO

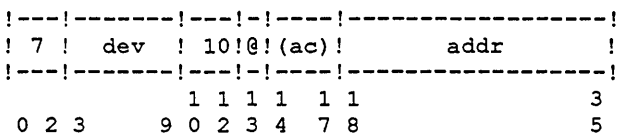
Conditions In and Skip if One



Tests the input conditions from device *dev* against the effective mask *cond*. If any condition bits selected by a 1 in *cond* are 1, skips the next instruction in sequence. If the device supplies more than 18 condition bits, only the right 18 are tested. Condition bits in the left half word can be tested by reading them with a CONI and then using a test instruction.

10.1.1.7 BLKO

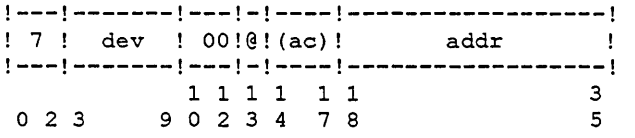
Block Out



Adds one to each half of a pointer in location *addr* and places the result back in *addr*. Then performs a data output (DATAO) instruction using the right half of the incremented pointer as the effective address.

10.1.1.8 BLKI

Block In



Adds one to each half of a pointer in location *addr* and places the result back in *addr*. Then performs a data input (DATAI) instruction using the right half of the incremented pointer as the effective address.

10.1.2 KL10 I/O Instruction Summary

The BLKI and BLKO instructions are unique in that their behavior varies depending on whether they are executed as a priority-interrupt instruction. If the BLKI or BLKO is not executed as an interrupt instruction, and the addition has caused the count in the left half of the pointer to reach zero, the monitor goes on to the next instruction in sequence. Otherwise, it skips the next instruction. If the BLKI or BLKO is executed as an interrupt instruction, and the addition has caused the count in the left half of the pointer to reach zero, the monitor executes the instruction in the second interrupt location for the level. Otherwise, it dismisses this interrupt and returns to the interrupted program.

Note

A BLKI or BLKO instruction is effectively a whole in-out data-handling subroutine. It keeps track of the block location, transfers each data word, and determines when the block is finished. Initially, the left half of the pointer contains the negative of the number of words in the block; the right half contains an address one less than that of the first word in the block.

The above eight instructions differ from one another in their total effect, but they are not all different with respect to any given device. A BLKO acts on a device in exactly the same way as a DATAO; the two differ only in counting and other operations carried out within the processor and memory. Similarly, no device can distinguish between a BLKI and a DATAI. Also, a device always supplies the same input conditions during a CONI, CONSZ, or CONSO, whether the program tests them or simply stores them.

Hence, the eight instructions can be categorized in four types, represented by the first four instructions described above. Moreover, a complete treatment of the programming of any external device can be given in terms of these four instructions, two of which are for input and two for output. The four exhaust the type of information transfer that occurs in the I/O system.

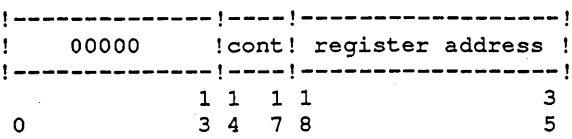
Every device requires initial conditions; these are sent by a CONO, which can supply up to 18 bits of control information to the device control register. The program can determine the status of the device from up to 36 bits of input conditions that can be read by a CONI (but only 18 bits can be tested by a CONSZ or CONSO). Some input bits simply reflect initial conditions but are subject to subsequent adjustment by the device, and still others may have no direct connection with the output conditions.

Data is moved in and out in bytes of various sizes or in full 26-bit words. Each program transfer between memory and a device data buffer requires a single DATAI or DATAO. Every device has a CONO and CONI, but it may have only one data instruction unless it is capable of both input and output. A DATAI that addresses an output-only device simply clears location *addr*. On the other hand, a DATAO that addresses an input-only device is a no-op. When the device code is undefined, or an addressed device is not in the system, a DATAO, CONO, or CONSO is a no-op, a CONSZ is an absolute skip, and a DATAI or CONI clears location *addr*.

10.1.3 KS10 I/O Instructions

Unlike other processors, the KS10 has no special format for I/O instructions. Instead, they are the same instructions that handle the peripheral equipment, the console, and memory status; although for consistency, they do have 1s in the left three bits of the opcode field. UNIBUS-type devices, as all peripheral equipment on a KS10, are handled through UNIBUS adapters. There are several instructions reserved for this purpose, and they are described here in terms of their general effects for handling external devices.

As in all instructions, the processor does effective address calculations, but for the I/O instructions, it ignores the results and recomputes an effective I/O address. The I/O address specifies an I/O register in some UNIBUS device or in the console or memory controller. For convenience, this effective address will be referred to as *addr*. An I/O address is analogous to an extended virtual address in that it has a fundamental length of 30 bits, but not all of its bits are implemented in a given processor. In a KS10 I/O address, the right 18 bits are the register address, and the left 12 are the controller number, of which only 4 bits are implemented. An I/O address thus has this format:



Bits 0-13 must be zero. *Cont* is the controller number. Of the 16 possible controller numbers, only three are currently used: 0 addresses the console and memory controller; 1 addresses UNIBUS adapter 1; 3 addresses UNIBUS adapter 3.

Controller	Register Address	Specifies
0	100000	Memory status
0	200000	Console (microcode only)
1	400000 - 777777	Adapter 1 UNIBUS registers
3	400000 - 777777	Adapter 3 UNIBUS registers

The I/O address calculation is like an effective address calculation in which the result can be "global," that is, can have more than 18 bits. When the result is an 18-bit "local" register address, it is automatically interpreted as specifying controller 0. The calculation is limited to one level of indirection or indexing or both, and any intermediate result that is used as a memory address must be local, because the KS10 is confined to section 0.

If there is no indexing or indirection, the I/O address is simply *addr*.

If there is indexing only, and the left half of the right half of *ac* is negative, the I/O address is the local sum of *addr* and the right half of *ac*.

If there is indexing only, and the right half of *ac* is positive, the I/O address is the global sum of *addr* and the right half of *ac*, but remember that bits 0-13 must be zero.

If there is indirection only, the I/O address is the contents of location *addr*.

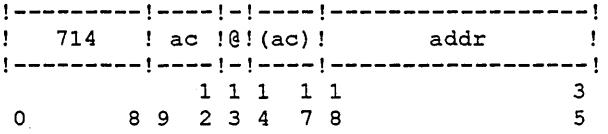
If there is both indexing and indirection, the I/O address is the contents of the location specified by the sum of *addr* and the right half of *ac*.

Note

An index register can supply the entire I/O address, but it can also be used to supply only the controller number when *addr* is the register address. This latter technique is useful for employing common code for multiple adapters.

10.1.3.1 BSIO

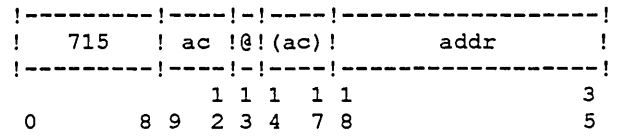
Bit Set I/O



In the word read from I/O register *addr*, sets bits corresponding to 1s in *ac* and writes the result back in register *addr*.

10.1.3.2 BCIO

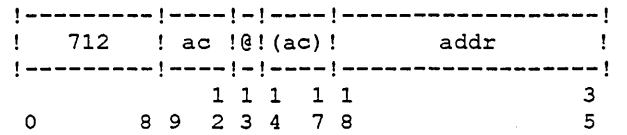
Bit Clear I/O



In the word read from I/O register *addr*, clears bits corresponding to 1s in *ac* and writes the result back in register *addr*.

10.1.3.3 RDIO

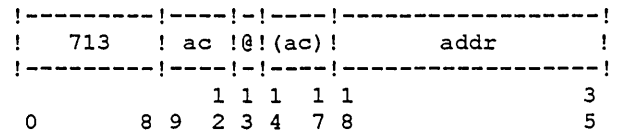
Read I/O



Reads the contents of I/O register *addr* into *ac*.

10.1.3.4 WRIO

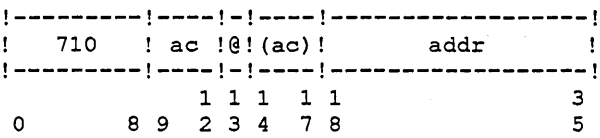
Write I/O



Writes the contents of *ac* into I/O register *addr*.

10.1.3.5 TIOE

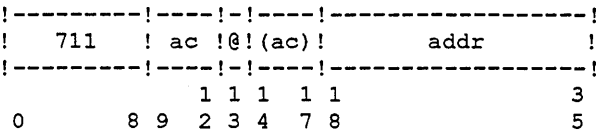
Test I/O Equal



If all bits of I/O register *addr* corresponding to 1s in *ac* are zero, skips the next instruction in sequence.

10.1.3.6 TION

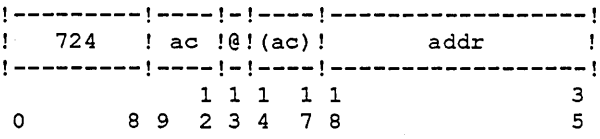
Test I/O Not Equal



If not all bits of I/O register *addr* corresponding to 1s in *ac* are zero, skips the next instruction in sequence.

10.1.3.7 BSIOB

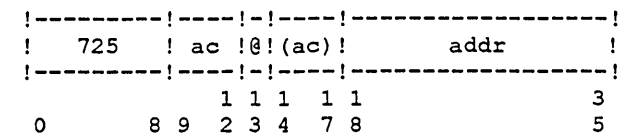
Bit Set I/O Byte



In the byte read from I/O register *addr*, sets bits corresponding to 1s in *ac* bits 28-35 and writes the result back in register *addr*.

10.1.3.8 BCIOB

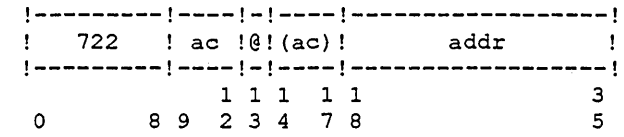
Bit Clear I/O Byte



In the byte read from I/O register *addr*, clears bits corresponding to 1s in *ac* bits 28-35 and writes the result back in register *addr*.

10.1.3.9 RDIOB

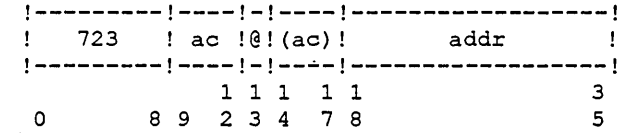
Read I/O Byte



Reads the contents of I/O register *addr* into *ac* bits 28-35. Clears *ac* bits 0-27.

10.1.3.10 WRIOB

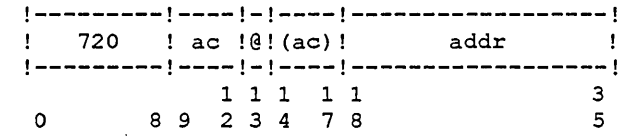
Write I/O Byte



Writes the contents of *ac* bits 28-35 into I/O register *addr*.

10.1.3.11 TIOEB

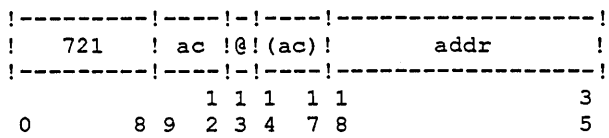
Test I/O Equal Byte



If all bits of I/O register *addr* corresponding to 1s in *ac* bits 28-35 are zero, then skips the next instruction in sequence.

10.1.3.12 TIONB

Test I/O Not Equal Byte



If not all bits of I/O register *addr* corresponding to 1s in *ac* bits 28-35 are zero, skips the next instruction in sequence.

10.1.4 KS10 I/O Instruction Summary

UNIBUS devices generally have data registers and control/status registers. Frequently, a single I/O address specifies two registers, one for reading and one for writing. A control register and status register in a device usually have the same address and also have bits in common; that is, information loaded into some of the control bits can be read as status. Ordinarily, a device is set up by reading status or testing individual status bits.

10.2 Device Overview

Many types of devices can be connected to a DECsystem-10. They can be divided into three main categories:

- Simple devices
- Controller-oriented devices
- Data Channels

10.2.1 Simple Devices

The simplest and oldest group of devices exists on the I/O BUS. They can transfer a single word or byte of information at a time, usually using a DATAI or DATAO instruction. The CPU is interrupted upon completion of a transfer. Occasionally, a device of this type uses two priority-interrupt channels, one for data transfers and one for device control. Data is transferred on a high-priority channel, usually to avoid missing data due to a slow response to an interrupt. Device-control functions are relegated to a lower-priority channel where the status or condition(s) of the device are not likely to change quickly. The data structures to maintain this type of device are minimal. Usually, a single DDB is all that is needed. A good example of such a device is the BA10-based CR10 card reader. The BA10 serves as a passive interface to the I/O BUS; whereas the CR10 is the device that actually performs the I/O.

10.2.2 Controller-Oriented Devices

A *controller* is an intelligent piece of hardware that can direct one or more devices. Controller-oriented devices may still interrupt on a per-word or byte basis, but usually require the execution of a single instruction (BLKI or BLKO) at interrupt level. The combination of the BLKI or BLKO instruction and the short interrupt-level routine is an indication that these devices are capable of a higher data-transfer rate than the previously discussed I/O BUS devices. Again, the use of two priority-interrupt channels is common, with data transfers on the high-priority channel and device-control functions on the lower-priority channel. These devices also have a relatively simple data structure, the DDB, but in this case, controller information as well as drive-specific information must be maintained. A good example of this type of device is the TD10-based TU55/56 DECtape drive. Here, the TD10 is both an I/O BUS interface and a controller.

10.2.3 Data Channels

A data channel is the most sophisticated and intelligent [1] type of device.

Typically, a DATAI or DATAO instruction is used to initiate a data transfer, but the channel is responsible for transferring the entire buffer without CPU intervention. This is often referred to as a *Direct Memory Access* (DMA) device. A channel may be a self-contained device such as a CI20, or it may be a controller capable of interfacing to several devices, such as a TM78 magtape formatter. Other types of channels are merely interfaces to non-*Digital* hardware. A good example is the DX10/IBM channel interface.

This device and others like it are often incorrectly referred to as controllers, when their real function is to provide a hardware-compatible port between the DECsystem-10 and a foreign device. In the case of the DX10, it connects to the I/O and memory busses, and responds like a standard I/O BUS device, but device commands are translated into IBM channel signals suitable for communicating with IBM Tape Control Units (TCUs).

The program (the bootstrap, diagnostic, monitor, and other components) uses DATAI and DATAO instructions to control the channel but not to actually transfer data, as in the case of the older I/O BUS devices. The DATAI or DATAO instructions are typically used to select registers within the channel (controller) or one of its connected devices. Once selected, the registers may be read or written with control information necessary for the channel to transfer data. For example, before a transfer can take place, the program might select a "drive" register in the channel, and store into this register the drive number on which the data transfer will take place. Another register may need to be selected to store the address of a command list that dictates how, where, and in what direction to transfer the data. Once the various channel registers have been properly initialized for the upcoming transfer, the final step is usually turning on the *go* bit. This instructs the channel to actually start the transfer.

---

[1] *Intelligent* is a relative term. Some controllers exhibit a high degree of competence with respect to the functions they must perform. This is seen in the manner in which data is transferred or errors detected and processed with little monitor intervention, a DX10 for instance. On the other hand, logic without discipline is not a measure of intelligence, as can be seen with the CI20.

™ IBM is a registered trademark of International Business Machines Corporation.

Note

The steps necessary for setting up and starting the channel vary greatly depending upon the type of channel. The description above is generic and not truly indicative of any particular device.

10.3 Monitor Data Structures

The monitor maintains various data structures for device-I/O purposes. Some of the more important data structures are:

1. MDT - MONGEN'ed device table
2. CHN - Channel data block
3. KDB - Controller data block
4. UDB - Unit data block
5. DDB - Device data block
6. DRV - Driver dispatch table

These data structures are arranged in general order of importance with respect to the monitor's use of the items at system startup time. At this time, the monitor performs *autoconfiguration* and dynamically creates all the necessary data structures. Autoconfiguration is discussed below, after a description of the data structures and the devices that use them.

10.3.1 MONGEN'ed Device Table (MDT)

The MDT contains information that allows the monitor to identify a particular device and associate properties or parameters with that device. There are two MDTs for each device driver assembled into a monitor. When MONGEN is run, it asks the user about the device drivers to be assembled into the monitor. The result of any reply is written into the configuration file in the form of macros which, when assembled, expand and build an MDT in COMDEV.

Each device driver also has an MDT. This is called the default MDT. Typically, the default MDT contains standard device-code definitions. When the monitor dynamically configures devices, it uses these MDTs to adjust the parameters (device codes in I/O instructions for example) of the newly configured devices to those specified at MONGEN time. If the event the MDT generated by MONGEN and expanded in COMDEV is empty, that is an indication that the standard device parameters are to be used. In this case, the default MDT in the device driver is used.

There is one important difference between the two MDTs. The MDT in COMDEV contains two words that do not exist in default MDTs. These are the first words in the MDT and are used as bookkeeping words by the monitor for counting the numbers of devices on the system. The data in these words is used to generate the device names.

10.3.1.1 MONGEN Device Options

For the purposes of this discussion, suppose you want to build a Tri-SMP monitor with DX10 magtape support, and your only DX10 resides on CPU1. When MONGEN is run, it asks you the following question (long dialogue format):

```
Include DX10/TX01/TX02 tapes (NO,YES,PROMPT)[
   TU70, TU71, TU72, and TU73 IBM channel magtapes are
   available through the use of a DX10 channel interface and a
   TX01 or a TX02 controller.  TX1KON is the driver module for
   these magtapes.
   Respond with one of the following:
   NO      Exclude driver
   YES     Include driver
   PROMPT  Include driver and prompt for parameters]:
```

You supply an answer following the : prompt. A response of NO indicates the device driver (TX1KON) is not to be assembled into the monitor. There will be no device driver support for the DX10 in the monitor you are about to assemble.

A response of YES indicates you want to include TX1KON. No other questions are asked for this device. This is, by far, the simplest way to have DX10 support.

A response of PROMPT indicates you want to include TX1KON, and you have special non-standard parameters that need to be specified. Here, things can get a little complex. Fortunately, the DX10 is a simple case, so your options are limited. Again, the long form of the dialogue is used. (Note: This question is asked only for multi-CPU configurations.)

```
Reserve devices per CPU: [
   In multi-CPU configurations, it is often desirable to
   reserve a number of controller or device slots for each CPU
   which normally has this type of device connected to it.  For
   example, in a dual CPU system, if each CPU has two channels
   for RP06s, one would like the controllers to have the names
   RPA, RPB, RPC, and RPD.

   For devices such as disks, where the controller name varies
   with each driver loaded, this question should be answered
   for each driver.

   For devices such as magtapes, where the controller
   designation (MT) is the same regardless of the driver being
   used, this question need only be answered for the first
   driver.

   Typing CR will default the number of controllers or devices
   to an appropriate value depending upon the driver.]:
```

The proper response to this question is "0,1,0"
where:

- The first 0 indicates you have no tape controllers on CPU0.
- The 1 indicates one controller (the DX10) on CPU1.

- The last 0 indicates no tape controllers on CPU2.

Merely replying YES to the initial question bypasses the reserved CPU counters option. This causes the monitor (at system startup time) to default the reserved counters. In this case, there is a theoretical maximum of 26 controllers (MTA through MTZ) and three CPUs. Since the monitor cannot predict the number of tape controllers, it must assume the worst case. This means that with only a single magtape controller on the system, the DX10 tape drives have the name MTIx rather than MTAx. Although rather obscure, this scheme ensures that each device in the system has the same name regardless of which CPU is the policy CPU at system startup. Note that this feature controls only the name associated with the controller and its drives and, as such, exists for aesthetic reasons. People typically like to see alphabetically ascending device names starting with A.

Next, MONGEN prompts you for additional information. Normally, there is no more to supply at this point, and the response is NO. If, however, you want to configure your device using a non-standard device code, this is the time to do it. For this discussion, assume the DX10 does not use the normal 220, 224, or 034 device codes but instead uses device code 270. MONGEN now prompts for CPU information. (Note: This question is asked only for multi-CPU configurations.)

```
Type a CPU number or press RETURN for all CPUs:
```

If you want the non-standard parameters to apply to all DX10s on all CPUs, then RETURN suffices. Otherwise, enter a specific CPU number. In this example, the DX10 is on CPU1, so the proper response is 1.

Now, MONGEN asks for the device code:

```
Device code (CR,0-774) [
Press RETURN to accept the standard device code definitions.]:
```

In this example, the proper response is 270. For KS10s, MONGEN asks two questions: one for the interrupt vector and one for the UNIBUS/CSR base address.

For MASSBUS devices, MONGEN prompts for a MASSBUS unit number. Because the DX10 is not a MASSBUS device, this question is omitted.

If the non-standard definition being entered is to be applied to a specific drive or all drives on the controller, then the next question is important.

```
Drive, slave, or unit: [
   Type a decimal drive, slave, or unit number, "ALL"  for  all
   units,  or  "NONE"  if  defining a controller parameter.  An
   answer to this question must be supplied.]:
```

where:

- *n* is a drive, slave, or unit number.
- ALL indicates all devices on the controller.

- NONE means that the non-standard definition is not drive specific but instead applies only to the controller. In this example, the proper response is NONE, because a non-standard device code is specified, a quantity independent of the drives on the controller.

The final question is meaningful only if the non-standard information being supplied is meaningful only to the driver. In the example, there is no other information, so a RETURN is all that you need to enter.

```
Data: [
   You may provide device-specific information. The response
   to this may contain a symbolic expression. If you press
   RETURN, this indicates that there is no device-specific data
   other than the previously specified device code
   information.]:
```

The data that could be entered here is meaningful only to the device driver. Therefore, it can be any 36-bit expression that MACRO can assemble. MONGEN cannot make validity checks on the supplied quantity. A typical example where this data might be useful is in the case of a line printer. The monitor can dynamically determine whether or not a printer has lowercase capability in all but one case, the BA10-based printer. The monitor normally assumes that all printers have lowercase capability, so to generate a monitor for a BA10 uppercase-only printer, you need to supply a data word. LPTSER defines the symbol LPT.UC as global. The proper response to this question is "LPT.UC##".

MONGEN assembles the results of the DX10 question into macro definitions that are written into the configuration file. The following is generated for the DX10 question in the example:

```
M.TX01==:1
DEFINE      MDTX01,<
EXTERN      TX1KON
MDCPUN      (00,01,00,00,00,00)
MDKL10      (1,270,0,0,<MD.KON>,0),<
MDTERM
> ;;END DEFINE MDTX01
```

"M.TX01==:1" indicates that the monitor is assembled with DX10 support turned on. The EXTERN of TX1KON causes LINK to load the device driver into the monitor's .EXE file. The MDCPUN macro contains six arguments, one for each possible CPU. The values correspond with the CPU counters specified in the dialogue. The MDKL10 macro generates all the other parameters. The 1 represents CPU1, 270 is the non-standard device code, and MD.KON indicates this is a controller-wide definition. The other zeros represent the parameters that do not apply to the example: the MASSBUS unit number, drive number, and data word, respectively. The MDTERM macro terminates the MDT definition. Currently, it expands to a zero, the normal MDT terminator.

10.3.2 Channel Data Block (CHN)

The CHN represents the highest level in the hierarchy of monitor I/O data structures. As its name implies, it is used by channel devices. [2]

CHN offsets, bits, and other associated symbols are defined in DEVPRM.MAC. The main item that makes one CHN different from the next is the device code associated with it. A CHN may exist for a single device (for example, CI20 or DX10) or may have multiple devices (for example, TM02/3 or TM78). Thus, part of the CHN is devoted to bookkeeping words necessary to maintain one or more devices. Some of the more important common CHN words are explained below.

The CHNDVC word contains the CPU and device-code information. It is used primarily by autoconfigure code to find an existing CHN. Typically, the device driver, after detecting the presence of a channel, calls AUTCON to find an existing CHN. The CHNDVC is then used.

Included in each CHN is an AOBJN pointer (CHNTBP) to a table of KDB addresses. The table is part of the CHN, being dynamically built at system startup time and expanded whenever a new channel device is powered up under timesharing. CHNTBP and the table it points to are most commonly used by the monitor's I/O scheduler routines. Generally, hardware-channel I/O is optimized by looking at multiple paths to a given device. The table allows the I/O schedulers to scan all devices on a single hardware channel, picking the first available device to do a data transfer.

There is also an in-use flag or *busy* word (CHNBSY) for the CHN. In most cases, the CHN is associated with a hardware channel that can perform a single data transfer at a time. Therefore, the monitor's I/O schedulers use this word as an interlock or flag, indicating the availability of the channel to perform a data transfer. For those devices that can perform multiple simultaneous transfers (for example, CI20) or operate in a block-multiplex mode (SA10), this word is ignored. The I/O schedulers for these devices rely on information stored in other data structures (usually the KDB) to maintain those I/O queues.

The other CHN words mostly perform disk and swapping I/O. With few exceptions, these words are misplaced and belong in the respective KDBs, because they pertain to functions specific to a single device on the hardware channel but not to the entire channel.

10.3.3 Controller Data Block (KDB)

The KDB maintains all information pertaining to a controller. A controller is an intelligent piece of hardware that can direct one or more devices. An example of such a device is the TM78 magtape formatter. Common KDB offsets, bits, and other associated symbols, as well as disk and magtape service extensions to the basic KDB are defined in DEVPRM.MAC. Device-driver-specific extensions to the KDB are defined in the monitor module for the respective device drivers. Some of the more important common KDB words are explained below.

[2] Currently, a CHN is required to use the MAPIO subroutine. Because of shortcuts taken in writing some of the KS10 device drivers (for example, LP2SER), a CHN is created solely for the purpose of calling MAPIO.

A few identifying items are associated with each KDB. Given one of these items, the KDB can always be located. KDBNAM contains the sixbit controller name. It is generated by AUTCON when the data structure is created. KDBCAM contains the CPU accessibility mask, a bit for each CPU that has access to the device represented by the KDB. The CPU bits correspond to the bit that resides in the CPU data block (.CPBIT). Therefore, when selecting a KDB for I/O on a given CPU, there is a simple test that indicates whether a CPU can perform I/O on the KDB pointed to by *ac* W. For example:

```
MOVE     T1,KDBCAM(W)   ;GET KDB ACCESSIBILITY BITS
TDNN     T1,.CPBIT##    ;COMPARE WITH THOSE OF THE CPU
JRST     <no access>    ;CANNOT DO I/O ON THIS CPU
<continue>              ;OK TO DO I/O
```

KDBDVC contains a PDP-10 device code or UNIBUS base/CSR address. KDBUNI contains a MASSBUS unit number or -1 of not a MASSBUS device. Given any of the above quantities, a KDB can easily be located for performing I/O or autoconfiguring.

Each KDB contains the address of the driver dispatch table in KDBDSP. The driver dispatch table allows the higher levels of the monitor (the service routines like UUOCON, FILUUO/FILFND/FILIO, and TAPUUO/TAPSER) to perform device-specific functions without knowledge of the device itself. KDBDSP allows this dispatch table to be easily located.

For the purposes of scheduling I/O, the KDB contains three bookkeeping words:

- KDBIUN - the initial unit pointer
- KDBFUN - the final unit pointer
- KDBCUN - the current unit pointer.

Each word contains a 30-bit address. KDBIUN and KDBFUN point to the first and last word of a unit table, respectively, the table being defined as part of the KDB itself. KDBCUN is used by the I/O schedulers as a pointer to the unit currently being examined for I/O readiness.

Among other things, the KDB also has defined words containing I/O instructions that are XCT'ed by the device drivers to do CONI/CONOs, DATAI/DATAOs, and BLKI/BLKOs. Because of the vast differences between doing I/O on a PDP-10 and UNIBUS, these words exist only on a KL10, the KS10 techniques being a bit more cumbersome. For example, to read the device status of a TM02/3 on a KL10, this simple instruction loads T1 with the CONI bits for the KDB pointed to by *ac* W:

```
XCT    KDBCNI(W)          ;CONI DEV,T1
```

On a KS10, the equivalent code is:

```
MOVE   T1,KDBDVC(W)       ;COPY UNIBUS BASE/CSR ADDRESS
RDIO   T1,<(DO.CS2)>(T1)  ;READ TM02/3 STATUS REGISTER
```

Again, the KDB is pointed to by *ac* W. DO.CS2 is the offset from the start of the UNIBUS base or CSR address.

10.3.4 Unit Data Block (UDB)

For those devices that have UDBs, it is the primary data structure for monitor I/O to that device and represents the lowest level in the hierarchy of monitor I/O data structures. Common UDB offsets, bits, and other associated symbols, as well as disk and magtape service extensions to the basic UDB, are defined in DEVPRM.MAC. Device-driver-specific extensions to the UDB are defined in the monitor module for the respective device drivers. Some of the more important common UDB words are explained below.

The common UDB definitions are fairly simple, as the bulk of the data contained within a UDB is very specific to each device. A UDB is identified by a sixbit name in UDBNAM. It contains a physical device number in UDBPDN. UDBDSN is a two-word quantity containing the drive serial number. This is used by AUTCON for linking multi-ported drives to their respective KDBs and for hardware error reporting.

UDBKDB is a table MXPORT words long that contains back pointers to the KDBs that have access to the UDB. MXPORT is defined in DEVPRM.MAC and can vary from 1 through 8. The number of KDBs pointed to by a UDB indicates the number of ports available for the device.

The UDB also contains a CPU accessibility mask for I/O scheduling (UDBCAM). This mask is used in the same way as the KDB mask (KDBCAM). Under normal conditions, UDBCAM should contain the inclusive OR of all the KDBCAM words for the KDBs pointed to by UDBKDB.

10.3.5 Device Data Block (DDB)

The DDB is the primary interface between user-mode I/O programming and the monitor's service routines, (for example, UUOCON, FILUUO, and TAPUUO). For those devices that do not have UDBs, it is also the the primary data structure for monitor I/O to that device and represents the lowest level in the hierarchy of monitor I/O data structures. Common DDB offsets, bits, and other associated symbols are defined in S.MAC. Service-routine extensions to the basic DDB are defined in the monitor module for the service routine in question, with the exception of the disk DDB, which is defined in COMMOD.MAC.

For all device types, a prototype DDB exists. This special DDB is used for creating all DDBs, whether at system startup time or during timesharing. However, the methods for creating and/or deleting DDBs vary with the type of device. For most devices, DDBs are created when the device is detected, usually at system startup. From then until the next system reload, the DDB is reused again and again, often by multiple jobs. For disks, multiplex devices, network tasks, and other special devices, DDBs are created on demand and deleted when no longer needed.

All DDBs, regardless of device type, maintain a number of words, used by the monitor to manipulate user buffers. Information stored includes I/O mode, buffer ring header addresses, I/O section numbers, and CPU accessibility data. DDBs also contain the address of the driver dispatch table, thus granting UUO-level code the ability to perform device-specific functions without knowledge of particular device programming aspects.

Because the service routine is to be assembled independently, the monitor does not know what other devices will be present in the system and therefore cannot fill in the linkage to the "next" DDB (the left half of DEVSER). DDBs are linked by AUTCON at system startup time. The order linkage within a group of DDBs for a single device type is in increasing numeric order (by DDB name), sorted according to ANF-10 station number and device-unit number. Then those groups of DDBs are arranged in alphabetical order based on the device name from the driver dispatch table.

10.3.6 DRV - Driver Dispatch Table

The driver dispatch table (DRV) was a data structure originally created to autoconfigure devices. It has been expanded so that the various service routines could call their device drivers without the need for defining another dispatch table. Common DRV offsets, bits, and other associated symbols, as well as magtape service extensions to the basic DRV are defined in DEVPRM.MAC.

The concept behind the DRV is simple. There are many quantities and functions associated with each piece of hardware that must be made available to AUTCON. The old school of thought was to assemble into the monitor complex data structures and code to be executed for a device. Now, using a DRV and the prototype data structures it refers to, much of the autoconfiguring process becomes independent of the devices on the system.

DRVs are linked together in a forward-linked list by 30-bit addresses. The links are initially generated by .LINK pseudo-ops in MACRO. Extended addressing *fixups* are done at system initialization time, producing the 30-bit addresses. DRVLST in AUTCON points to the start of the linked list. The DRVLST chain allows AUTCON to find any driver dispatch table, along with all the data structures and necessary subroutines to successfully autoconfigure all devices on the system.

There are three main groups of information stored in a DRV: pointers to data structures, addresses of subroutines and tables, and device characteristics. Prototype data structure pointers include the following:

```
- KDB length and address
- UDB length and address
- DDB length and address
- Microcode loader block address
- Interrupt level code length and address
```

Subroutine and table addresses include the following:

```
- Interrupt service routine
- DIAG_. UUO dispatch table
- MONGEN'ed device table (MDT)
- Compatible controller table
- Autoconfigure routine
```

Device characteristics are stored in two words:

- The first word, DRVCNF, contains mostly flags used to describe how to build the data structures for a device. Bits indicate whether it is a real hardware device (such as a line printer) or a software device (such as a multiplex channel). Other bits indicate how the device name is to be built (for example, octal or decimal numbers), whether the device has multi-port capabilities, if the driver runs in extended sections, cache bits, and so

on. A single characteristic byte also exists in this word. It contains the device-type code (.TYxxx). This arrangement of the device-type byte and the collection of bits allows AUTCON to quickly find a driver dispatch, given very little information. For that reason, DRVCNF must not be expanded to contain anything other than one-bit quantities.

- The second characteristic word, DRVCF2, contains only byte definitions. These include device-specific controller-type codes, maximum devices of this type that the system can support, maximum devices allowed on a controller, the highest drive number the controller can handle, and section numbers for data structures.

10.4 Autoconfigure Overview

Autoconfiguration is the process of detecting the devices attached to the system. This process is a collection of subroutines invoked at system startup and occasionally under timesharing. Historically, these subroutines existed primarily in the AUTCON module. A single call to that module was made, causing disks and magtapes to be dynamically built. Calling AUTCON at its entry point was often referred to as *running AUTCON*. Today, the architecture of AUTCON has drastically changed. It is no longer an all-inclusive module containing very specific knowledge about several types of disks and magtapes. Instead, it is a collection of random subroutines intended to be called by device-service routines and device drivers. None of the subroutines has any knowledge of device specifics beyond a distinction between channel types. There is, however, still one entry point which, when called, initiates autoconfiguration. Hence, *running AUTCON* is still valid.

10.4.1 System Initialization

10.4.1.1 AUTCON

AUTCON plays a major role during system initialization. All I/O data structures are built at this time. This is a rather time-consuming process that may account for as much as one half of the system startup overhead.

10.4.1.2 AUTINI

Very early in SYSINI, just after the monitor initializes the memory manager, the routine AUTINI is called. AUTINI initializes AUTCON. This routine sets up the various data structures used by the autoconfigure process. Its first task is to scan INTTAB for any old-style (assembled-in) prototype DDB definitions. A prototype DDB is defined as one that has no name in DEVSER and is used for creating other DDBs. These DDBs are the first to be linked into the DDB chain. DDBs are linked through the left half of DEVSER; the chain is pointed to by DEVLST. The prototype DDB addresses and lengths are also stored in DDBTAB and DDBSIZ, respectively. These are two tables indexed by device type code (.TYxxx).

After all prototype DDBs have been found in INTTAB, AUTINI then scans the driver dispatch tables. For each table in the DRVLST chain, the following steps are taken:

1. Any addresses in the DRV are "fixed up" for extended addressing use. While AUTCON must run in section 1, routines pointed to by the DRV may reside in other sections. Also, explicit section 1 references are inserted where zeros exist to avoid the possibility of

being called out of section 1 and causing a reference to a bad address with an indirect reference through the DRV.

2. The per-CPU device counters are set up in the COMDEV copy of the MDT. These numbers are necessary for AUTCON to correctly generate DDB names.
3. Prototype DDBs are linked into the DEVLST chain, and the appropriate entries are made to the DDBTAB and DDBSIZ tables.
4. Unlike the old-style DDB definitions, the prototypes pointed to by the DRVs do not have interlock words assembled into them. Consequently, the DEVCPU word must be "fixed up" to use the appropriate PI channel interlock.

The remaining work done by AUTINI is largely bookkeeping:

1. Global CPU counters are adjusted. Some devices have common DDB names but different drivers. Line-printer DDBs are all named LPTxxx, yet more than one device driver supports the printers. Magtapes fall into the same category. Since the per-CPU counters reside in the MDTs, the counts of reserved and allocated devices could skew if two printers use two different device drivers. For that reason, AUTINI examines the drive characteristics word (DRVCFG) in the dispatch table, looking for the global CPU counters bits. It then finds other drivers of the same device type (.TYxxx) and sets the counters in all the MDTs to the same values.
2. Compatible controller tables are compared. These tables indicate that a single multi-ported device may have different types of controllers. For example, a multi-ported TU70 may be accessible through three different types of channels: a DX10, DX20, or SA10. Once compared, if different controller types are found, AUTINI then "fixes up" various UDB and DDB parameters. It maximizes the lengths of the UDB and DDB and imposes any cache restrictions that may exist.
3. GENTAB is built and alphabetically sorted. While the format of GENTAB has changed, its purpose has not. GENTAB provides access to DDBs by generic name and forms the basis for the *high speed device search* algorithm. GENTAB is arranged in two-word pairs. The first word contains a three-character, right-justified sixbit device name such, as LPT or MTA. The second word contains the address of the first DDB having the name stored in word one.

10.4.1.3 AUTCPU

The next routine to be invoked is AUTCPU. In SYSINI at the STACON label, AUTCPU is first called for the policy CPU. Then, for each CPU in the configuration, AUTCPU is called using the XCTCPU facility. (XCTCPU allows subroutines to be executed on a non-policy CPU during system initialization.) The asynchronous flag is set so control returns immediately to SYSINI, where AUTCPU is queued up to run on the next CPU, and so on. When AUTCPU has been started on all the CPUs, the policy CPU then waits a reasonable amount of time for each CPU to complete autoconfiguration. If the wait timer expires, and some CPU is still running, a stopcode results.

Note

It is important to know that this is the only time during system initialization that AUTCPU can be safely called. When the monitor is first loaded and started, it checks to see if all the CPUs in the configuration are running, giving the operator the option

to start those that are not running. The next opportunity to start a CPU does not occur until after the system has started timesharing. This behavior differs from earlier monitors, where AUTCON tried to allow a CPU to be started anytime during system initialization. This was the cause of many autoconfigure failures (false devices detected, bad data structures being built, and numerous crashes).

AUTCPU is the heart of AUTCON. It is the one (fairly large) main loop responsible for causing the device drivers to build the data structures for all devices in the system. AUTCPU performs the following tasks:

1. Saves the state of the PI system. This is necessary because, detecting the presence of a device requires interrupts to be disabled.
2. Obtains the AUTCON interlock. Only one process or job should be manipulating AUTCON's data base at a time. The interlock is merely a spin lock, maintained on a per-CPU basis.
3. On KL10s, puts the DTE20s into protocol-pause mode. This allows the KL10 to run for an extended period of time without having to update keep-alive counters. This is necessary because interrupts are disabled during autoconfiguration, and no timely mechanism updates the keep-alive counters.
4. Redirects all typeout (if any occurs) to the CTY.
5. Checks a PDP-10 device code. It does this by saving the current CONI word and trying to give the device a PI assignment. The presence of a device is indicated by the fact that the PI assignment, when read back, matches the one just set. Before proceeding, it resets the original CONI (and possible PI assignment) for the device.
6. Checks a UNIBUS address. It does this by calling the UBGOOD subroutine. A non-existent CSR causes a page-fail trap. UBGOOD intercepts the page-fail traps and takes a skip or non-skip return, depending on whether a trap occurs.
7. Skips the next step if the device in question does not exist.
8. Calls all hardware device drivers in the DRVLST chain at their DRVCFG entry point. A driver may take one of three returns:
   - The device may not be a type supported by the driver (for example, line-printer service called for a TM02/3). In this case, AUTCPU steps to the next driver.
   - The device is a type that the driver supports (for example, MASSBUS disk driver called for an RP06). It builds the necessary data structures and requests AUTCPU to step to the next device code.
   - The device is a type that the driver supports (for example, TM78 driver called for an TM78). It builds the necessary data structures and requests AUTCPU to step to the next device driver, as another magtape formatter may exist on the same RH20.
9. Advances to the next possible PDP-10 device code.
10. Advances to the next UNIBUS address, cycling through all UNIBUS adapters as well.
11. Loops back and tests this device, unless all devices on the system have been tested.
12. Calls all software device drivers in the DRVLST chain at their DRVCFG entry point.

13. Calls NETSER so it can update its configuration tables and broadcast the changes to all nodes in the ANF-10 network.
14. Exits from protocol pause mode on KL10s.
15. Releases the AUTCON interlock.
16. Restores the state of the PI system.

This concludes the main autoconfigure loop. AUTCPU can be called under timesharing as well as from SYSINI. It is invoked by the OPR/CONFIG **AUTOCONFIGURE** command or by executing RECON. UUO function .RCRAC.

10.4.1.4 AUTSYS

This is the final step in autoconfiguring during system initialization. A small number of devices are not autoconfigured but instead have their data structures assembled into the monitor. These devices depend on special INTTAB entries that contain information pertaining to the number of DDBs that must be generated, the length of the DDB, PI channel assignment, and so on. AUTSYS scans INTTAB for these entries, building the necessary DDBs and linking them into the DEVLST chain. AUTSYS replaces the old LINKDB subroutine in previous monitors. As with LINKDB, AUTSYS is not executed under timesharing.

10.4.2 Autoconfiguring Under Timesharing

This process takes one of three forms. It could be as simple as automatically or forcibly adding a single device or drive on a controller, or as complex as configuring all devices on a CPU newly added into the system.

10.4.2.1 Automatically Configuring a Single Device

If a new device is powered on, it is sometimes desirable to have this device added into the system. For many devices, this can be an automatic process. These devices generate online interrupts. The device driver notices it is processing an interrupt for a previously non-existent device and marks the drive number in a bit map. Word KDBNUM (New Unit Mask) in the KDB contains the offset within the KDB of a bit map indexed by physical drive number. When a bit for a drive is set to one, it indicates that the necessary data structures for that device must be built. Interrupt level is an undesirable place for such a time-consuming process to take place. Therefore, the data structures are built during the once-a-second call to the driver. The driver, upon noticing a 1 in the bit map attempts to get the AUTCON interlock and if successful, builds the data structures and clears the bit in the bit map. At this point, the new drive is available for use. If the AUTCON interlock is not available, configuring the new drive is postponed until the next once-a-second call, when the bit in the bit map is again noticed, and an attempt to acquire the interlock is made again.

Note

To minimize overhead, a device driver tries to configure only a single device per once-a-second call. When an entire controller comes online (for example, a TX02 with 16 tape drives), it takes about 16 seconds for all drives to be configured. They appear at the rate of one per second.

You may not always want a drive to be automatically configured. This is a case for the KDBIUM (Ignore Unit Mask) bit map. By using the OPR/CONFIG **SET IGNORE** command, the operator can cause a particular drive to be ignored when the online interrupt occurs. This command is equivalent to DIAG. UUO function .DISDS, sub-function .DISSI. The bit map that KDBIUM points to is the map of drives to be ignored when a device comes online. Like the KDBNUM map, the KDBIUM map is indexed by physical drive number.

10.4.2.2 Forcibly Configuring a Single Device

Devices that do not generate automatic online interrupts on power up may still be configured. The OPR/CONFIG **ADD** <device> command can be used for this function. This is equivalent to executing the DIAG. UUO function .DISDS, sub-function .DISSA, which attaches a single device. When this is done, the DIAG. UUO code invokes the device driver, gets the AUTCON interlock, and calls the driver at a special entry point used to configure a single device. The interlock is always available, in a sense, because the DIAG. code runs at UUO level, and if the interlock is not currently available, the job blocks (is put to sleep) until the interlock is free.

10.4.2.3 Autoconfiguring an Entire CPU

When a CPU is added into the system, initially the monitor has no knowledge of any devices on that CPU. The CPU, while capable of performing compute-bound tasks, cannot do I/O. In the timesharing environment, there is little value in having only a computing engine. Therefore, attaching a CPU (DIAG. UUO function .DISDS, sub-function .DISSA) causes AUTCON to automatically run. The same entry point, AUTCPU, is used under timesharing as during system initialization. In all respects, it functions identically under both conditions, regardless of when it is called.

10.5 Autoconfigure on a Device Driver Level

This section builds upon your knowledge of AUTCON and examines the inner workings of configuring on a device-driver level. There are many different types of devices, and little basis for comparison between them. Therefore, the best way to illustrate what happens at the driver level is to examine some specific devices in the following categories:

1. A single device
2. A single controller on a channel
3. Multiple controllers on a channel
4. A software device

This is not an all-inclusive list. However, it is sufficient to give you a basic understanding of what happens in some of the more common type of devices.

10.5.1 A Single Device

A typical example of a single (and simple) device is a line printer. Almost any LPT driver is appropriate for the following discussion, but the focus will be on LPTSER, the I/O BUS driver.

All device drivers contain a DRVCFG entry point. When AUTCON finds an existing device, it calls all its drivers at the DRVCFG entry point. In the case of LPTSER, this is the routine LPTCFG. Since most I/O BUS devices do not return a unique code (such as a MASSBUS drive type code), it is normally impossible to know that the device code that AUTCON has selected is an I/O BUS printer. The MDT helps LPTSER make this determination.

You may recall that MONGEN generates MDTs with optional non-standard device codes. Most of the time, this is not the case, however, so the device-code field in the MDT for LPTSER (LPTMDT) is zero. Therefore, we must rely on the contents of the default MDT in the driver, because it contains the standard or default device codes for I/O BUS printers. It is inconvenient for LPTSER (and all the other drivers as well) to interpret the contents of both MDTs on each call from AUTCON, so the AUTMDT routine performs that function.

Given the address of the generated MDT and the default MDT, AUTCON determines if the currently selected device code fits the requirements of the driver, LPTSER. A non-skip return indicates the selected device does not match the requirements. LPTSER then takes the skip return back to AUTCPU. AUTCPU interprets this return to mean it must try other drivers. A successful return from AUTMDT is indicated by a skip, and *ac* T1 contains the contents of the matching data word in the MDT. The data word is saved for later reference.

Now, LPTSER checks to see if the DDB for the printer already exists due to a previous call to LPTCFG. It does this by comparing the device code with those stored in existing printer DDBs. The routine AUTFND (Find existing DDB) does this. If a DDB is found, the non-skip return is taken, and *ac* F contains the address. If the DDB does not yet exist, then LPTSER must create one.

It is necessary to determine the name of a DDB before it can be created, because the DDB creation routine stores, among other things, the name and unit numbers. The next free unit number to be assigned is available by calling the AUTADN (Allocate Device Number) routine. AUTADN uses the first two words of LPTMDT to arrive at its results. The first word contains the reserved per-CPU counters, and the second word contains the count of devices allocated on a per-CPU basis. Upon return, AUTADN allocates the next available unit number and loads *ac* T1 with that number. LPTSER has only to put the three-character sixbit generic device name (LPT) in the left half of *ac* T1, put the ANF-10 station number in *ac* T2 (zero means a local device), and call AUTDDB to create the DDB. Creating the DDB involves allocating core, copying the prototype DDB (from DRVDDB), storing the name, unit number, station number, and resolving a few other quantities.

LPTSER then points AUTCON at the I/O instructions in the DDB and makes a call to AUTDVC to fill in and complete the instructions. Once completed, LPTSER performs any printer DDB-specific address fixups or relocations as required by the service routine. There are, after all, some things that AUTCON just cannot do for the driver.

LPTSER then causes the interrupt routine to be built. This is done by pointing AUTCON at the DDB and the prototype interrupt routines and calling AUTICD (Interrupt Code generator). AUTICD dynamically allocates core for the interrupt code and copies the prototype code into it. Again, LPTSER performs any address relocations in the interrupt code to make it functional. AUTCSO is then called to link the interrupt code into the CONSO skip chain.

The final step involves retrieving the MDT data word and testing the LPT.UC bit, the uppercase printer bit defined by specifying non-standard data when MONGEN was run.

At this point, the new line-printer DDB is usable. It may be OPEN'ed by LPTSPL or assigned to a user job. The DDB is permanent and exists until the system is reloaded.

10.5.2 A Single Controller on a Channel

One example of a single controller on a channel is the DX10. As you may recall, the DX10 isn't really a controller but a channel interface to an IBM TCU (in *Digital* terminology, a TX01/2 or an STC 3800). You can, however, think of the combination of a DX10 and a TCU as a single entity, a controller.

As in the case of the simple line printer, AUTCON selects an existing device. It is still up to the service routine (TX1KON) to determine if the device is really a DX10, and it does this in exactly the same way LPTSER did it— by using the MDT.

The DX10 is a channel interface, so it must have a CHN to do its I/O. TX1KON sets up T1 with the necessary channel-type bit, in this case, CP.DX1 for a DX10. AUTCON already knows the device code, so there is no need to pass this information along. Then AUTCHN is called to either find an existing CHN or create a new one if necessary.

Since this device can perform data transfers on one of several drives, a KDB is required. AUTKDB finds an existing KDB or creates one, given the information stored in the driver dispatch table. It also creates a block of interrupt code and links it into the CONSO skip chain. If the DX10 is a vectored interrupt device, the appropriate words in the KDB for maintaining an interrupt vector are set up. Core is also allocated for the initial channel program at this time, the address being stored in KDBICP. Upon return, *ac* W contains the KDB address.

Note

Now is the time to cause the DX10's microcode to be loaded. It is loaded regardless of the state the microprocessor. That is, the monitor cannct guarantee that the currently loaded microcode (if any) is truly functional. As a safety measure, this step is always performed. This step is not necessary for all controllers.

TX1KON then checks to see if each possible drive exists. For IBM channel devices, this is a rather expensive undertaking. Therefore, blocking checks are performed only during system initialization. If configuring under timesharing, a request to read sense bytes is queued to the TCU. The results are received at interrupt level and are processed like the automatic online interrupts generated when the system powers up, described earlier. Assume that the magtape subsystem is being configured at system initialization time.

If the drive responds, and the proper identifying sense bytes are returned, it is time to build the UDB and DDB. If there is no response, or invalid sense bytes are returned, then the drive is assumed not to exist, and TX1KON steps to the next drive.

First, the drive serial number is retrieved. Note that like most devices, DX10-based magtape drives return only a single serial-number word. This word is temporarily stored as the low-order word in the serial-number word pair.

The serial-number words, along with the physical drive number, are used as arguments to the AUTDPU (Dual-Ported Unit check). (The term *dual-ported* is historical and is often used when *multi-ported* is really meant. AUTDPU searches all other magtape controllers for a matching controller type, based upon the contents of the compatible controller tables. If a match is found, then the serial number and physical drive number of the drive being configured are compared to those already configured. If a match results, then a multi-ported drive has been detected, and the appropriate link words are updated in the UDBKDB table for the existing UDB.

Density, track, interrupt bits, and hung device timers are gathered and fed to the routine TAPDRV. This routine (in TAPSER) is common for all types of magtape drives on all types of controllers. It creates the UDB and DDB if it is necessary to do so.

AUTUDB creates a UDB if it does not already exist. One of the arguments in the calling sequence is the UDB table offset (in the KDB). Using this offset, AUTUDB tests the table entry for a nonzero value, a UDB address. This word may have been previously filled in by AUTDPU if the drive is ported to some other controller. Consequently, the UDB pointed to may really be "owned" by another controller. *Owned* is not really accurate, but best describes the situation. In the normal case, the name of the UDB is derived from that of the KDB. For example, if the KDB being configured has the name MTB, then the most logical name for drive zero on the controller is MTB0. If, however, the drive is ported to another controller that was configured first, for example, the MTA controller, then the UDB name is set to MTA0. Now, here's a situation where the MTB controller has access to the MTA0 drive. While this doesn't present any problems from a software standpoint, it may confuse some people who are not aware that multi-ported magtapes use a single UDB and DDB. In reality, one controller has no more influence over the drive than another. Hence, *owner* is used incorrectly.

A DDB is created for the drive if one did not already exist. The KDBNUM bit map entry for the drive is cleared, because the device really does exist along with its data structures. The KDBIUM bit map entry is also cleared, as it makes no sense to configure a device that would immediately be ignored.

Control returns to TX1KON, which steps to the next possible drive until all are tested.

10.5.3 Multiple Controllers on a Channel

This hardware arrangement is not much different from the case of a single controller on a channel. The important difference is that some channels, like a DF10C/RH10, RH20, or SA10 can communicate with several controllers rather than just one. One example of multiple controllers on a channel is the TM78 magtape formatter. The TM78 co-exists with other TM78s or TM02/3s on a single RH20. Each formatter is assigned (by hardware) a MASSBUS unit number. This quantity is referred to by many different names in various hardware and software manuals. Another term is the MASSBUS device code.

The autoconfigure process is the same as in the case of a single controller on a channel, with one exception. After configuring, rather than taking the non-skip return, which tells AUTCPU to step to the next device, the monitor takes the skip return. AUTCPU then calls additional drivers. AUTCPU has no knowledge of what happened on the initial call. It does not know whether the TM78 was configured. It has no need to know.

10.5.4 A Software Device

Software devices are unique in that they are a figment of the monitor's imagination! They are not tangible devices, nor are they associated with a piece of hardware. One example of a software device is the multiplex channel (MPX). Since this is not a hardware device, most of the system initialization logic to configure a device was bypassed. One of the last things AUTCPU does is scan all drivers for software devices.

Like all other drivers, MPXSER has a DRVCFG entry point (MPXCFG). System initialization for this driver is very simple. It needs to initialize (zero) a single counter. To avoid doing this on subsequent AUTCPU calls (either on non-policy CPUs or under timesharing), it also maintains a flag that indicates that MPXSER has been initialized.

MPXSER uses some parts of AUTCON under timesharing, however. When a DDB needs to be created as the result of an OPEN UUO for example, MPXSER first acquires the AUTCON interlock. Since this can happen only at UUO level, AUTLOK always returns with the interlock, possibly after having blocked the job until the interlock is available.

A call to AUTSET is necessary to set up various CPU variables, principally the address of the driver dispatch table (MPDDSP).

Since MPXSER maintains its own copy of device counters in MPXNUM, it's not necessary to call AUTADN to do any allocation. Therefore, it only has to increment its device count and supply AUTDDB with the generic device name of MPX. AUTDDB does all the usual work of dynamically allocating core, storing the device name, and so on.

When a RELEASE UUO is executed, the MPX DDB needs to be deleted. This is done by loading *ac* F with the DDB address and calling AUTKIL (Kill DDB). AUTKIL unlinks the DDB from the DEVLST chain, removes any GENTAB entry if necessary, and returns the core used by the DDB.

10.6 I/O Subroutines

When the implementation plans for the new autoconfigure methods were being drawn up, it became apparent that many devices required the use of similar monitor services. For example, many device drivers needed to refer to MASSBUS registers, yet each driver had its own set of subroutines to perform the task, and the calling sequences were often unique to each driver. Obviously, with a little effort, things could be reorganized and code consolidated. Other examples exist as well.

In addition to the autoconfigure routines, AUTCON is also a repository for common I/O routines. This did not come about by design, but evolved as a matter of convenience, because there is no other good place to put common I/O subroutines. The following sections briefly describe some of the more important subroutines.

10.6.1 MASSBUS Register I/O

MASSBUS channels have several registers. The number of registers vary depending upon the type of channel (RH10, RH11, and others). The registers are accessed using DATAI/DATAOs (KL10) or RDIO/WRIOs (KS10). Some registers reside in the channel itself and others in the devices connected to the channel.

There is always a possibility that an attempt to read or write a MASSBUS register will fail due to faulty hardware. This failure is referred to as a *Register Access Error* (RAE). In order to defend against RAEs, a complex series of instructions is required. Drivers that do not care to recover from RAEs use simple DATAIs and DATAOs. Some of the following subroutines contain code to defend against RAEs.

10.6.1.1 RDDTR

Read Drive Type Register (RDDTR) is used primarily by the autoconfigure routines, but could be used any place where reading the drive type code is necessary. The calling sequence is

```
SKIPA   P1,[MASSBUS-UNIT,,0] ;IF AUTOCONFIGURING
MOVE    W,KDB-ADDRESS        ;IF NORMAL TIMESHARING
PUSHJ   P,RDDTR##
<return>
```

On return, *ac* T2 contains the MASSBUS drive type code. All other *ac*'s are preserved.

10.6.1.2 RDMBR

Read MASSBUS Register (RDMBR) reads the contents of a specified register and defends against RAEs. It is used both by the autoconfigure code and the various device drivers. The calling sequence is

```
SKIPA   P1,[MASSBUS-UNIT,,0] ;IF AUTOCONFIGURING
MOVE    W,KDB-ADDRESS        ;IF NORMAL TIMESHARING
MOVE    T2,REGISTER
PUSHJ   P,RDMBR##
<return>
```

On return, *ac* T2 contains the contents of the specified MASSBUS register. All other *ac*'s are preserved. Normally, RAEs do not happen, so much of the complex code is never executed. If an RAE occurs, RDMBR tries up to 10 (decimal) times to read the register before giving up. If after all retries the error persists, then *ac* T2 is returned to the caller with whatever data bits the channel managed to retrieve from the register. The caller gets no indication that an error has occured.

10.6.1.3 SVMBR

Save MASSBUS Register (SVMBR) is a co-routine used by interrupt levels to save the current address register, which may be in use by a higher (UUO) level routine. The calling sequence is

```
MOVE      W,KDB-ADDRESS
PUSHJ     P,SVMBR##
<return>
```

SVMBR uses no *ac*'s. All *ac*'s are preserved. This routine is not defined in KS10 monitors, as the registers on a KS10 are I/O addresses offset from the UNIBUS base/CSR address for the device in question. Hence, there is no register number to preserve.

10.6.1.4 WTMBR

Write MASSBUS Register (WTMBR) writes the contents of *ac* T2 to the specified register and defends against RAEs. It is used both by the autoconfigure code and by the various device drivers. The calling sequence is

```
SKIPA   P1,[MASSBUS-UNIT,,0] ;IF AUTOCONFIGURING
MOVE    W,KDB-ADDRESS        ;IF NORMAL TIMESHARING
MOVE    T2,REGISTER
PUSHJ   P,WTMBR##
<return>
```

All *ac*'s are preserved. Normally, RAEs do not happen, so much of the complex code is never executed. If an RAE occurs, WTMBR tries up to 10 (decimal) times to write the register before giving up. The caller gets no indication that an error has occured.

10.7 Finding Data Structures

Since AUTCON builds nearly all of the I/O data structures, it is logical to assume that many of the pointers to these pieces of data reside in AUTCON. Knowing just a little information about the type of data structure you are interested in is usually enough to easily locate it.

10.7.1 DDBTAB - DDB Table

The table DDBTAB is indexed by device type (.TYxxx). Each entry in the table contains the 30-bit address of the prototype DDB of its type in the system. Prototype DDBs are also linked into the DEVSER chain. The left half of DEVSER in the DDB contains the 18-bit address of the next DDB in the system. The DEVSER chain is a forward linked list, terminated by a zero.

10.7.2 DEVLST - DDB List

The word DEVLST resides in COMMON and contains the address of the first DDB in the system. The left half of the word contains the DDB address. The right half is unused.

10.7.3 DRVLST - Driver Dispatch Table Chain

The location DRVLST contains the 30-bit address of the first driver dispatch table in the system. Offset DRVNXT in a dispatch table contains the 30-bit address of the next dispatch table in the system. The DRVLST chain is a forward-linked list, terminated by a zero.

10.7.4 HNGLST - Hung-checked DDB List

The contents of this word are identical to that of DEVLST. The word is no longer used by the monitor, as the method for testing potentially hung devices does not do a DDB-by-DDB search of the DDB chain. HNGLST is, however, maintained for the sake of programs that GETTAB its value to find DDBs. It should otherwise be considered obsolete. HNGLST resides in COMMON.

10.7.5 KDBTAB - Controller Table

The table KDBTAB is indexed by device type (.TYxxx). Each entry in the table contains the 30-bit address of the first KDB of its type in the system. Offset KDBNXT in the KDB contains the 30-bit address of the next KDB of its type in the system. The KDB chain is a forward-linked list, terminated by a zero.

10.7.6 SYSCHN - Channel List

The word SYSCHN resides in COMMON and contains the address of the first CHN in the system. The left half of the word contains the CHN address. The right half is unused.

10.7.7 DIAG. UUO Function .DIDVR

Originally written for diagnostic use, DIAG. UUO function .DIDVR reads device registers. You may wonder how it is possible to implement a generic diagnostic function to do something that is quite device specific. That's easy for the monitor to do as long as the calling program supplies all the necessary information. Given a device name, a starting offset, and a word count to return, this function finds the data structure and returns as few or as many words as the program desires.

If this function were to be used as intended, the calling program would most likely GETTAB the offset of stored register values and supply that offset to the DIAG. UUO. For example, a diagnostic program could read the offset to the start of the stored MASSBUS registers in the KDB, using GETTAB **%LDMBR**. Then it would retrieve those registers from the monitor, using DIAG. function .DIDVR as follows:

```
        MOVE    T1,[%LDMBR]     ;GETTAB ARGUMENT
        GETTAB  T1,             ;READ OFFSET FROM MONITOR
          HALT                  ;ANCIENT MONITOR
        HLRM    T1,OFFSET       ;STORE OFFSET IN RH OF WORD
        MOVE    T1,[-23,,ARG]   ;23 WORDS AT LOCATION "ARG"
        DIAG.   T1,             ;READ REGISTERS INTO "DATA"
          HALT                  ;PRE-704 MONITOR
        <program continues>

ARG:    EXP     .DIDVR          ;DIAG. FUNCTION CODE
        SIXBIT  /RPA/           ;DEVICE NAME
OFFSET: XWD     -20,0           ;-NEGATIVE COUNT,,OFFSET
DATA:   BLOCK   20              ;STORAGE FOR 16 REGISTERS
```

Now suppose you have to diagnose a software problem with a line printer, and you have reason to believe there is something wrong with the DDB. You may retrieve the entire DDB, using DIAG. UUO function .DIDVR and save yourself the trouble of having to find the DDB in the monitor. To begin, put the appropriate DDB name in ARG+1. Assuming the length of the DDB in question is 61 (octal) words, in word OFFSET, put -61,,0. The BLOCK 20 becomes a BLOCK 61, because you are returning the entire DDB. All that is left to do is put -64,,ARG in *ac* T1 and execute a DIAG. UUO. Note that the DDB length of 61 words plus 1 word for the data structure pointer (-61,,0) plus 1 word for the device name plus 1 word for the function code word equals 64. The DIAG. UUO expects a negative argument block length in the left half of the *ac* and the address of the argument block in the right half of the *ac*. Upon completion of the UUO, the entire printer DDB is returned, starting at location DATA.

10.8 Device Service Routines

The device dispatch table in the DDB, not to be confused with the driver dispatch table, DRV, provides a standardized set of entry points for all UUO-level functions for this device. There are two possible table formats: short and long. The short table format contains only the basic entries required of all service routines (initialization, IN, OUT, RELEASE, and others). If the device requires any additional functions (such as LOOKUP for directory devices), it must have a long dispatch table. The DVLNG bit in the DEVMOD word of the DDB specifies the format the corresponding dispatch table has. The base address of the dispatch table is contained in the right half of the DEVSER word of the DDB.

Most of the UUO-level routines depend so much on the nature of the device that they handle, that little can be said about them in general. The initialization routine is called during system initialization and performs whatever functions might be appropriate. Generally, all condition bits for the device are initialized, and its priority-interrupt level assignment is cleared. The RELEASE routine usually performs this same function. The CLOSE routine does whatever is appropriate for the completion of a file on its device. For example, the paper tape punch routine punches several inches of leader. The disk routine adds an entry for the new file to a directory upon output CLOSE.

The only routine in the dispatch table that is not the device-dependent part of the UUO is the hung-device routine. When a transfer is started on a device, a hung-device timer is initialized in the DEVCHR word. Each second, the clock-interrupt routine calls DEVCHK in UUOCON. Here, the timer is decremented for each active device. The expected interrupt clears the field. If the field is, however, decremented to zero, the interrupt has not occurred within a reasonable amount of time. Assuming that the device is hung, the monitor dispatches to the hung routine in the device service routine. This routine can try to either recover from the hung condition or reinitialize the device. If the device-hung routine gives a skip return, no further action is taken by DEVCHK. If the device-hung routine gives a non-skip return, DEVCHK calls the DEVHNG routine in ERRCON. DEVHNG clears the IOACT bit for the device and types an error message on the job's controlling TTY. The job is stopped unless it has enabled error trapping for hung devices.

The function of the input and output routines is to start the device. Normally, this is done by executing a DATAI/DATAO on a KL10 or a RDIO/WRIO on a KS10. For disks, however, usually only a request is added for a transfer to an appropriate queue, and the actual transfer is started at a later time. The IOACT bit is always set before returning to device-independent code, indicating that an interrupt is expected from this device. As long as IOACT remains set, the device-independent code does not call the device-dependent code again, because the function of *starting the device* does not need to be performed.

Several other housekeeping functions are performed. The hung time is initialized, if one is specified for this device. The IOFST bit in DEVIOS is set to inform the interrupt routine that the next interrupt is the first for the buffer.

Chapter 11
Disk service

The disk-service routine is the most complicated of all the service routines, because it must manage a sharable resource, and it has a complicated in-core data base. This module explains how jobs compete for use of the disk, how that competition is resolved by the queue mechanism, and how I/O is done within the framework of the file structure.

All device-dependent functions for disk files are performed by a group of modules known as the *disk service*. The disk service performs two different and logically independent types of functions: I/O operations and file operations. I/O operations are the reading and writing of specific blocks on specific units. File operations involve the processing of directories, pointers, and related items. The file-processing software accepts logical requests stated in terms of file names, file structures, and relative blocks within a file. From such requests, it sets up physical requests for operations on specific blocks that can then be handled by the I/O software. The file processor frequently calls upon the I/O processor to read and write various special disk blocks.

The disk software includes several modules that are assembled separately and included in the monitor, as needed, at load time. Most of the executable code is included in FILFND, FILIO, and FILUUO. (These modules are frequently referred to as FILSER, the name taken from the early disk service module that was eventually separated into the three we have today.) These modules perform all operations that are independent of the controller type and are present in all monitors. There are separate routines to handle controller-dependent functions on each controller. These routines (RPXKON for RH10/11/20s, RNXKON for DX20/RP20s, RAXKON for HSC/CI20s, DSXKON for SA10s, and others) are loaded only if needed. SWPSER creates I/O requests for the swapper and interfaces with FILIO. The data base for the disk service is contained in the modules named COMDEV and COMMOD.

11.1 Hardware Principles

Each disk unit is connected to a channel controller, and all communications to that unit must go through the controller.

A channel is connected to the CPU by way of an I/O BUS, MASSBUS, or Interconnect Port Adapter. Internally to the CPU, each of the above-mentioned paths uses the CBUS, which provides *Direct Memory Access* (DMA) facilities. Thus, memory accesses do not interrupt the CPU. Ultimately, data passes through the MBOX and in or out of memory.

Each disk transfer is started by the CPU executing a DATAO to a specific controller. As a result of the DATAO, a word is sent to the controller. This word specifies one particular unit of those connected to the controller, a physical disk address (track number and other items) and the core address of a list of core areas. The transfer is to or from consecutive locations on the disk, but may be scattered among an arbitrary number of core areas. The length and address of each of these core areas are on the list whose core address is sent to the controller. This list is known as the *Channel Command List*. The total length of the areas on the Channel Command List determines the number of words transferred.

Before a transfer can be started, the unit must be positioned, and the controller and the data channel must be idle. All are busy until the transfer is complete. The CPU, however, is needed only for the time required to send the instruction to the controller. It can then go on processing while the transfer to memory takes place. When the transfer is complete, the controller causes an interrupt on its assigned channel.

Note

Some units can perform implied positioning. That is, the monitor is not required to position the heads before a transfer is to take place. Still other units have no separate positioning commands. The monitor attempts to optimize I/O and minimize head movements by performing head positioning before starting a transfer. Regardless of the unit capabilities, the movements of the I/O requests through the various disk queues assume independent positioning will take place. However, for some units, certain states within I/O queuing are essentially noops.

Before a request can be positioned, the unit must be idle. The unit is then busy until it reaches the designated track, but the controller is almost immediately available for other operations. When the unit reaches position, it informs the controller. The controller causes a priority interrupt at that time if it does not have a transfer in progress. If the controller is busy when the unit reaches position, there is no interrupt, but an attention bit for the unit is set in the controller. When the interrupt occurs upon completion of a transfer, the attention bits indicate which units reached position (or had errors) during the transfer.

Note

Some controllers, such as an RH10/11/20, can initiate multiple positioning requests to idle units even though an I/O transfer is taking place on a busy unit. Other controllers, such as a CI20/HSC, allow multiple transfers on one or more units. And block multiplex channels, such as an SA10, allow multiple operations, but only one to any given unit.

The basic addressable unit of disk storage is a sector. It is sometimes useful to know which sector (of each track currently accessible) will reach the read-write heads next. Therefore, the controller has a sector counter for each of its units. The contents of the sector counter for a given unit may be obtained by reading the appropriate device register for the channel in question.

11.2 Structure of disk files

To the software, the basic unit of disk storage is a block, which is always 128 words. Any number of blocks may be combined to make up a file. To normal user programs, disk blocks can be read or written only as part of a file. The file is identified by a file name and extension, and by the project-programmer number of its owner. The program may read or write the blocks of a file either sequentially or randomly (directly). Likewise, the file may be accessed in either buffered mode or dump mode. The structure of a file is independent of the manner in which it was written or is to be read.

The first block of every file is a *Retrieval Information Block* (RIB). The RIB contains a great deal of descriptive information about the file, and tells where the data blocks of the file are located. The RIB itself, however, is not a data block and is never seen by a program reading the file nor directly written by a program writing a file. The monitor reads and writes RIBs as necessary in order to perform functions requested by user programs.

Files are usually written as groups of consecutive blocks. There is a pointer in the RIB corresponding to each group. The pointer tells the location of the first block of the group and the number of blocks in the group. It is desirable to have as few separate groups as possible.

11.3 Directories

The locations of all files belonging to one user are found in a *User File Directory* (UFD) for that user. The UFD is itself a file with an RIB and the normal structure of a file. The file name of a UFD is the binary project-programmer number of the user. The extension is always UFD. The data blocks of a UFD contain two-word entries. Each entry points to the RIB of a file belonging to that user, and specifies its name and extension.

All the UFDs belong to an "artificial user" with project-programmer number [1,1]. No other files belong to [1,1]. Hence, the UFD for [1,1] is a directory to the directories. It is commonly called the *Master File Directory* (MFD). The collection of files consisting of an MFD, all the UFDs to which it points, and all the user files to which the UFDs point to is called a *file structure.*

11.4 File Structures

As a collection of files, the file structure is logically independent of any hardware considerations, such as units and controllers. In actual practice, however, there are several restrictions. All the files on a single pack or unit must belong to the same structure. A single structure may be spread over several separate units.

The file structure, rather than the unit, is the logical entity recognized by the file processing software. On every LOOKUP or ENTER, a structure or list of structures must be specified. (Note that a file name and extension and project-programmer number uniquely identify a file only within a given structure). If any part of the structure is removed from the system, the entire structure becomes inaccessible. There is only one case in which data is accessed without necessarily being part of a file structure: the swapper addresses the disk system in terms of physical disk addresses.

11.5 Allocation of Disk Space

Before a block can be added to a file, it must be allocated for that file. Disk space is allocated in clusters, where a cluster is a fixed number of consecutive blocks. On each unit, there are *Storage Allocation Tables*, or SAT blocks that have a bit corresponding to each cluster of the unit. If a cluster is allocated, the corresponding bit is set in the SAT block. The bit's being set prevents that block from being allocated to any other file.

The number of blocks per cluster is a parameter of the file structure and can be changed only when the structure is *refreshed*, or *reinitialized*. The total number of clusters for a unit depends on the size of the unit and the number of blocks per cluster. There may be more clusters than can be accounted for with a single SAT block. In this case, there are as many separate blocks of SATs as necessary, and each SAT block is physically near the blocks that it describes. All the SAT blocks for a file structure are combined into a file called SAT.SYS. This file is initially set up by the REFRESH code, and the information in it is updated regularly as the system operates. However, SAT.SYS is not normally read or written as a file. There is, in core, a *Storage Allocation Pointer Table* (SPT) for each unit, which tells the physical disk address of each SAT block for that unit. When the monitor needs to read or write a SAT block, it sets up a request for the specific block that is needed.

When an SAT block is in core, it resides in a *Storage Allocation Block* (SAB). All the SABs for a unit are linked together and to the SPT for that unit. If a unit has several SAT blocks, all of them may, or may not, be in core at one time. The number of SAT blocks to be kept in core is a parameter of each unit. This parameter may be changed without needing to refresh the structure. The SABs and SPTs are kept in Section MS.SAT of the monitor's address space.

Disk space is allocated in two different ways. If a user is writing a file and reaches the end of the space previously allocated, additional space is allocated at that time. If possible, the space is allocated immediately after the last group, so that an additional pointer will not have to be set up. The number of blocks to be allocated is a parameter of the structure and may be changed without refreshing. The user may explicitly allocate any number of blocks at the time he builds a file, by doing an extended ENTER. These blocks are allocated as a single group of consecutive blocks, allowing the file to be written or read with the least amount of overhead processing. When the file is closed, any unused blocks are returned.

The disk I/O-processing software maintains information about each piece of disk hardware in three main data structures whose definitions can be found in DEVPRM.MAC. These include the *Unit Data Block* (UDB), *Controller Data Block* (KDB), and *Channel Data Block* (CHN). The I/O processor acts on requests set up by other processors. Each request resides in a disk device data block, and specifies a unit, block number, core address, and operation to be performed. Significantly, the number of words to be read or written is not specified initially, but is determined just before the transfer is initiated. A *disk device data block* (DDB) has all the standard features of any DDB, plus a great deal of additional information unique to a disk. Disk DDBs are set up dynamically as INIT UUOs, and ASSIGN commands, which give a logical name to disk, are executed. There is, therefore, a DDB for each user software channel that may do disk I/O. Every disk transfer is the result of a request being set up in a disk DDB and presented to the I/O processor. This includes reading and writing of user files, swapping transfers, and all transfers done by the monitor for its own purposes.

11.6 Request Queues

When an I/O request is presented to the I/O processor, the transfer or positioning is started immediately if all the necessary devices are available. Sometimes, however, the request must be added to a queue of requests for a specific device. If the request requires positioning, it is added to the *Position Wait* (PW) queue for a specific unit. If the request does not require positioning (that is, the unit is positioned properly), it is added to the *Transfer Wait* (TW) queue for the data channel. The queues are formed simply by linking together the DDBs beginning with the UDB for a PW queue or the CHN for a TW queue.

Every time there is a disk interrupt, each unit that needs positioning is positioned for one of the requests in its PW queue. Then a transfer is started for one of the requests in the TW queue for that channel. Two optimization routines choose the request to process next.

11.7 Optimization Routines

The positioning and latency optimization routines try to choose the best request to process next from the PW and TW queues. To decide what is meant by *best* is somewhat difficult, but there are two basic considerations. First, an attempt is made to minimize the time that each unit is not doing data transfers. In addition, an attempt is made to try not be grossly unfair to any individual request. It is undesirable to delay one request indefinitely in favor of requests that can be processed more efficiently. Therefore, each optimization routine chooses, every so often, the request that has been waiting the longest.

Fairness counts are maintained for positioning and for transfers on each data channel. Each time there is a transfer-done interrupt, the fairness counts for that channel are decremented. On an interrupt when the positioning fairness count has expired, each unit that needs positioning is sent to the track required by the oldest request in its PW queue. Similarly, if the fairness count for transfers has expired, the transfer is initiated for the oldest request in the TW queue. Whenever either count expires, it is reset to a value that may be specified when the monitor is built.

Bottlenecks can occur at both unit and channel levels. Contention at these levels can be reduced by spreading demand over several channels, and if possible, by avoiding keeping high-use system files, user files, and swapping space on the same unit.

11.8 Data Structures

Many data structures support disk service. They are fairly complex, mainly because each data structure is often linked to one or more other disk-related data structures. Because there are so many data structures, it is difficult to define one without making references to others. So, they are presented in the following sections in a hierarchical fashion and logically separated into groups that are file-structure, data-file, and hardware related.

11.8.1 File Structure Data Base

User file data resides on *file structures*. A file structure is a logical organization of one or more (up to 63) disk packs, arranged by the monitor for reading and writing data on behalf of a timesharing user. This organization makes it possible for a user to perform I/O to any data file without specific knowledge about the physical disk types involved or the number of disks that constitute the file structure. Once a file structure is made available (mounted) on the system, it may be referred to as a device. The monitor always treats file structure names as physical devices.

All user disk information is stored as named files according to a method that allows the information to be accessed by name instead of by physical disk address. A named file is uniquely identified in the system by a file name and extension. A file name consists of one to six alphanumeric characters, and extensions can have between zero and three alphanumeric characters. File names and extensions are stored in ordered lists called directories. A directory is no different from any user data file, except that its contents describe file names, extensions, and locations on a structure.

11.8.1.1 TABSTR

This is a table that contains the addresses of each *structure data base* (STR) in the system. The table is indexed by file-structure number, the range being from 1 to 36. File-structure numbers are used only by the monitor. A user has no need to know the number of a file structure. Therefore, it is impossible for a user to obtain this information. File structures are typically referred to by name.

TABSTR is defined in COMMOD.

11.8.1.2 Structure Data Block (STR)

One STR exists for each file structure mounted on the system. The STR contains, among other things, the structure name, the file-structure number, its size, the amount of free space remaining on the structure, the number of jobs that have mounted the structure, and the owner (if any). The STR also contains the address of the first *Unit Data Block* (UDB) in the structure. +STRUN1

When a structure is defined (mounted by a privileged program), a STR is dynamically allocated out of section zero free core. When a structure is removed (dismounted by a privileged program), the STR is returned to the free core pool.

11.8.1.3 Job Search List (JSL)

A JSL defines a list of one or more structures that represent device DSK. If a user program specifies DSK for the device name and instructs the monitor to read a file, the monitor looks for the file starting with the first structure in the JSL, and tries each additional file structure in the JSL until the file is found or the end of the JSL is reached.

The JSL resides in the PDB. The format is a simple nine-bit byte stream, each byte containing a file structure number or special codes that delimit portions of the search list.

11.8.1.4 System Search List (SSL)

An SSL defines a list of one or more structures that represent device SYS. In all other respects, the use of the SSL is identical to the JSL.

The SSL resides in COMMOD at location SYSSRC. The format is the same as the JSL.

11.8.2 File Data Base

A data file is the logical organization of one or more disk blocks that contain user data and one or more disk blocks of monitor-related data for retrieving the file. File I/O is performed in a fashion independent of the type of disks involved.

11.8.2.1 Job Device Assignment Table (USRJDA)

The USRJDA is part of the job's UPT, at location USRJDA, with an extension that .USCTA points to. It has one entry per user channel. Each entry is zero if the channel has not been initialized. Otherwise, it has flags in the left half, indicating which UUOs have been performed on behalf of the channel, and the right half contains the address of a DDB. In particular, it contains the addresses of any disk DDB that the user has initialized as an I/O device.

11.8.2.2 Disk Device Data Block (DDB)

One DDB must exist for each file opened by a user program. This is true not only for disks, but for all other devices as well. The disk DDB is different from most other DDBs in that one does not exist for each device, as is the case for unit record devices.

When a disk file is opened, the monitor dynamically creates a DDB by allocating per-process (funny space) core. A prototype DDB (DSKDDB) is copied into the newly allocated core, and relevent words are then filled in with data specific to the file being opened. When the file is closed and the software channel released, the DDB is returned to the funny space core pool.

The disk DDB definitions are in COMMOD.MAC.

11.8.2.3 Retrieval Information Blocks (RIB)

An RIB resides on disk and contains all file attributes and data necessary for the monitor to locate the user data blocks associated with a file. There are usually two RIBs associated with each file on disk. The first is called the *prime* RIB. The second is the *redundant* or *spare* RIB. The spare RIB is a copy of the prime RIB and immediately follows the last data block in the file. It is written by the monitor but never read. It is used by disk damage assessment and recovery programs.

An extensible portion of the RIB contains retrieval pointers, or words of information that describe regions of the disk that hold user data. The monitor allocates disk blocks for user data as the file is being written. To optimize disk usage, the monitor attempts to allocate contiguous regions on disk. However, due to fragmentation, this is not always possible. Therefore, it is conceivable that for a very large file, or on a badly fragmented disk, the storage area for retrieval pointers may be exhausted. When this occurs, the monitor creates an *extended* RIB on disk. The last retrieval pointer in the prime RIB points to the extended RIB. In the event that the retrieval pointer storage is exhasted in the extended RIB, a second

extended RIB is created. A file may have a maximum of eight extended RIBs. Regardless of the number of extended RIBs, a spare RIB is always written following the last data block.

A retrieval pointer has one of two formats. The first format is always found in the first retrieval pointer of prime RIBs. It is called a *change of unit* pointer. Because a file structure can span more than one physical disk unit, a change of unit pointer is used to direct the monitor's I/O to a particular logical unit within the structure. The second type of retrieval pointer is used to describe a region of the disk that contains data. The items contained within this pointer are a cluster count, a checksum, and a cluster address. The cluster count indictates the size of the disk region (in clusters) where data is stored, and the cluster address indicates the beginning of the data. The checksum is used as a consistency check to insure that data within the first block of the region is valid.

11.8.2.4 Access Tables (ACC)

An ACC block is created for every different version of an opened file. The ACC is essentially a cache of frequently referenced per-file data. The information contained within an ACC block includes the highest relative block allocated, the written file size, file read and write counts, address of the *Name Block* (NMB), address of the *Project-Programmer Block* (PPB), and portions of the RIB for quick file access. Every time a file is opened for reading, the file's read counts are incremented. When a file is closed, the read or write counts are decremented appropriately. If both counts are zero, the ACC is said to be *dormant*; that is, there are no users reading or writing the file. Dormant ACCs are not deleted. Instead, they remain available to the monitor to provide quick access on subsequent operations to the same file.

ACCs are dynamically allocated out of FILSER core. This core pool is created at system initialization time and consists of a fixed number of words. When the FILSER core pool is exhausted, the monitor scans the ACCs, looking for those that are marked as dormant. Dormant ACCs may be deleted, thus providing the necessary storage for additional ACCs or other FILSER-related core blocks.

The ACC definitions are in COMMOD.MAC.

11.8.2.5 Name Blocks (NMB)

To optimize locating files, the monitor memorizes file-name information on a per-directory basis in an NMB. The NMB contains the ACC address, the directory use count, and the *know* and *yes* words. Use counts indicate how many users of a directory exist. The *know* and *yes* words are bit masks, one bit for each structure on the system. Bit n in the *know* mask indicates the monitor knows whether or not a file exists on file structure n. The corresponding bit in the *yes* mask is turned on if the file definitely exists.

NMBs are dynamically allocated out of FILSER core.

The NMB definitions are in COMMOD.MAC.

11.8.2.6 Project-Programmer Blocks (PPB)

PPBs are used to retain directory-related information. One PPB is created for each logged-in PPN, or for every file which is opened in a unique directory. The PPB contains a link word to an NMB, and like the NMB, also contains the *know* and *yes* masks. These two masks are used for the same purpose as their counterparts in the NMB, except that the masks refer to directories rather than to specific files within directories.

PPBs are dynamically allocated out of FILSER core.

The PPB definitions are in COMMOD.MAC.

Chapter 12
Scanner Service

The terminal scanner service (SCNSER) is the interface between the terminal device service routines and the monitor. It processes special terminal characters and directs the I/O to the correct process. This chapter presents the important scanner service data structures and concepts.

All device dependent functions for terminals are performed by the Scanner Service comprising SCNSER and an additional routine depending on the type of scanner. This additional routine might contain actual I/O instructions, and will contain the beginning of the interrupt routine, and other sections which vary according to the scanner being used. The bulk of the service routine is independent of scanner type, and is contained in SCNSER. The data line scanner acts as a relay station connecting the DECsystem–10 to all user terminals. Terminals communicate with the scanner and the scanner communicates with the DECsystem–10. Along with every character received from the scanner is a line number identifying the terminal from which it was sent. Similarly, a line number must be included with every character sent to the scanner. Whenever a line scanner has finished an operation, it causes a transmit done interrupt to call scanner service. The scanner scans the flags for all terminals and causes an interrupt in the DECsystem–10 whenever any terminal is ready for service. One of the major functions of the scanner service is handling these interrupts.

SCNSER considers all data line scanners to be equivalent interfaces for allowing terminals to communicate with the DECsystem–10. While the various interfaces may have significant protocols and complexities of their own, they will not be covered here.

12.1 Data Structures (General)

The DECsystem–10 has two major divisions of data structures for its terminal data: line-based information and job-based information. Loosely speaking, the line information is used primarily at the device interrupt and clock interrupt levels, and the job information is used mainly at UUO level.

12.1.1 Line Information

There are three main structures for line information: LINTAB (the line table), the Line Data Block or LDB, and the TTY chunk.

12.1.1.1 Line Data Blocks (LDBs)

LDBs contain information about a terminal line. There is one LDB for each terminal line which is can be connected to the system. They are created at once-only time in the SCNCFG routine. While there is a free pool of available LDBs for dynamic terminal connections, the LDBs themselves statically allocated in memory. This allows commands to be typed on a terminal without the need to allocate a new data structure to handle the command, and without need to buffer characters in a special way until the LDB can be created.

For a complete description of an LDB, see the monitor tables or the definitions in SCNSER.MAC. However, it is important to remember that the LDB contains the following information:

1. Pointers to the various input and output chunk streams
2. Line status bits
3. Line characteristic bits
4. Horizontal position counter
5. MIC information
6. User-defined break characters
7. Count of characters to echo
8. A pointer to the associated TTY DDB (if assigned)

12.1.1.2 LINTAB

LINTAB (the line table) is used to locate the LDB for a particular terminal line given its line number. It contains one entry for each terminal in the system, including PTYs and CTYs. Each entry LINTAB + n contains the global address of the LDB line number n.

12.1.1.3 TTY Chunks

The TTY chunks are a set of eight† word blocks of core, in which the first word is used to maintain doubly-linked lists. They can be in one of five data streams for an LDB, or on the "free list".

The remaining words in the TTY chunk contain 12-bit bytes for character data information (three per word). Each byte contains either a character code, possibly with flags, or a special function code. These function codes are referred to as "meta characters".

The chunks are allocated and placed on the free list by the SCNCFG routine at system startup. Location TTFTAK is the head of the list, from which free chunks will be allocated. The list tail is pointed to by TTFPUT. If any further chunks are released, they will be linked after this one. The count of free chunks is contained in TTFREN.

† The size of the chunks can be changed with MONGEN, by changing the value of symbol TTCHKS. This value must be a power of two and at least four. Eight is the default value.

12.1.2 Job Information

The job-related terminal information is contained in the following data structures and tables: the TTY DDB, TTYTAB, and the JDA.

12.1.2.1 The TTY DDB

TTY DDBs are assigned and deassigned from a pool of available DDBs as jobs are created and destroyed or terminals are initialized and released. The TTY DDB contains information which relates to the job, such as the following:

1. Pointers to the user buffers
2. Device physical and logical names for the terminal
3. I/O status and usage information (DEVIOS, DEVSTA)
4. Device mode information (DEVMOD)
5. A pointer to the LDB

The TTY DDB is a short-dispatch DDB.

12.1.2.2 TTYTAB

TTYTAB is a table in COMMON which has one entry per job and points to the TTY DDB of the controlling terminal for that job. A zero entry indicates that the job has no controlling terminal, which means that the job number is not assigned, or is in the process of being created or destroyed.

12.1.2.3 USRJDA

The Job Device Assignment table, or JDA, is part of the job's UPT, at location USRJDA with an extension pointed to by .USCTA. It has one entry per user channel. Each entry is zero if the channel has not been initialized. Otherwise, it has flags in the left half indicating which UUOs have been performed on behalf of the channel, and the right half contains the address of a DDB. In particular, it contains the addresses of any TTY DDBs which the user has initialized as I/O devices.

12.1.3 Linking Job and Line Information

As noted above, the LDB and TTY DDB contain pointers to each other. These links are established by the following events:

1. A command which requires a job number to be assigned is typed on an unused line.
2. An ATTACH command is typed which does not need to run LOGIN.
3. An unused line is initialized as an I/O device.

The links are broken by these events:

1. The user detaches from the line.
2. A job logs out or is destroyed without ever logging in.
3. An attached line is disconnected from the system (e.g., a dataset is hung up).

The association for an ATTACH command or UUO are made by TTYATT, the association for a new job or command are made by TTYATI, and that for an I/O device is made by GETDDB.

The routines to break the links are TTYDET, TTYDTC, PTYDET, and PTYDTC. The DTC flavors will wait wait for command processing to complete if necessary, and are for asynchronous disconnects which could conflict with command processing. The TTY flavors will force a job which is not logged in to be destroyed.

12.2 Chunk Management

As I/O devices, terminals are unique in having buffer space in the monitor. One important consequence of this is that a job may be swapped out while having terminal I/O in progress. Also, since a user may type on a terminal at any time, the monitor must have a place to put the characters. Characters from a terminal keyboard are stored in an input chunk stream until they are requested as input by the program or as a command by the monitor. Characters put out to a terminal are stored in an output chunk list and are sent to the terminal by an interrupt routine as the line scanner requests them. Chunks can be allocated from the free list to one of five streams associated with an LDB:

1. The input stream
2. The echo filler stream
3. The primary output stream
4. The output filler stream
5. The out-of-band stream

Each such chunk stream has a set of words in the LDB to describe it:

1. The count of characters in this chunk stream
2. The list head (where to remove the next character)
3. The list tail (where to add the next character)

The input stream is logically divided into two streams, to keep track of which characters have been echoed. The input stream also has a pair of counters for deleted characters, since we can't really delete a single character from the middle of the chunk stream efficiently.

12.2.1 Initial allocation

The total number of chunks to be allocated is found at system startup in the left half of location TTCLST in COMMON. This location is initialized with the value of TTCHKN when the monitor is built. TTCHKN defaults to the value TTCHKK * TTPLEN, but can be changed with MONGEN. It is usually best to not to define TTCHKN, however. If it is necessary to override the default number of chunks, the value of TTCHKK should be set in the MONGEN dialog. Its default value is sufficient to allow for one chunk for each of the five data streams for each LDB, plus at least TTYWID characters more per LDB. If there are not many lines defined in the system, TTCHKK will be increased beyond this value in an attempt to ensure that there will be enough TTY chunks.

In the above calculations, TTPLEN is the total number of LDBs in the system (TTYs and PTYs), and TTYWID is the assumed average TTY WIDTH setting of terminals which will be connected to the system. TTYWID defaults to 80, but this value can be changed with MONGEN.

12.2.2 Dynamic Chunk Allocation

Characters are placed in and removed from the TTY chunks using three macros: LDCHK, LDCHKR, and STCHK. Macros are used rather than subroutines to speed up the handling of characters in the monitor. This expedient adds very little to the size of the code. These macros do the following:

LDCHK
Take a character out of a chunk without returning any chunks to the free list. This is most useful when echoing input.

LDCHKR
Take a character out of a chunk, and return any just-emptied chunk to the free list.

STCHK
Put a character in a chunk, allocating a new chunk from the free list if necessary.

These macros must be called only while scanner interrupts have been disabled by obtaining the SCNSER interlock. The SCNOFF macro disables scanner interrupts and the SCNON macro enables them again. Both are defined in S.MAC.

12.3 Terminal I/O Overview

In this section we give a general overview of the major structure of I/O processing in SCNSER, to lay the foundation for a more thorough examination of its methods. SCNSER contains both terminal-specific device-dispatch code for I/O UUOs, and some UUOs which are unique to terminals. It also functions for UUOCON as a device service routine, even though it depends in turn on other modules of the monitor to function as device service routines for it. Some of these, in turn, will be dependent upon still other modules to serve as device drivers, etc., for many levels. Since SCNSER is also called from clock level, we will examine it briefly from each of these three levels: UUO, clock, and interrupt.

12.3.1 UUO Level

Viewed from UUOCON as an I/O device, a terminal is actually quite simple. When it receives an input request from the user's program, it tries to fill the user's buffer with data from the terminal, and will place the program into an input wait state (TI) if the input request can't be satisfied immediately. When UUOCON gives it an output request, it extracts the data from the user's buffer and sends it in the general direction of the terminal as fast as it can, returning if it can send all the data immediately, and blocking only if it cannot empty the user's buffer right away.

What SCNSER is really doing at UUO level for output is taking data from the user's buffer and stuffing it into the output chunks until they're full or the user's buffer is finally empty. After each character it deposits into the output chunk stream, it makes sure that the output routine knows that the line has data to send. For input, SCNSER reads only fully-echoed

characters from the input chunks and stuffs them into the user's buffer. The buffer is terminated when it fills or when a break character is finally deposited into it. The definition of a break character varies with the current I/O mode.

12.3.2 Clock Level

SCNSER is called every clock tick† or so to start terminal output. This call appears just after label STOPAT in module CLOCK1. This call will result in calling a device-dependent routine to send the data out from the DECsystem–10 to the destination terminal. Once the line has finally been queued to the device-dependent output routine, it is the responsibility of interrupt level to finish the data transmission and to keep trying to empty the output chunks for the line.

SCNSER is also called every second to perform some timing and housekeeping function. This is where idle lines are disconnected and lines which appear hung are given another chance to try to complete their output.

SCNSER is also responsible for maintaining the count of commands waiting to be processed (COMCNT). COMCON is called to process any pending commands when COMCNT is positive. SCNSER has a set of routines for the use of COMCON which are very similar to those provided for user terminal I/O, except that the monitor processes the characters directly rather than moving them into and out of user buffers.

12.3.3 Interrupt Level

This is where we find the most complicated portions of terminal service. At interrupt level, SCNSER must receive characters from the device drivers, determine what sort of special processing they need, and insert them into the input chunk stream. It must also process output characters, determine whether they need special processing or filler characters and synchronize them as necessary with any special terminal functions which may have been requested at UUO level. SCNSER must also handle the echoing of characters here, and will wake up blocked jobs which have finally received sufficient input or now have enough room in the TTY for more output. It is also at this level that COMCNT might be incremented, thus initiating command processing.

12.4 Terminal I/O Details

In this section we analyze the flow of characters through SCNSER in greater detail. We will see just how various special terminal functions are implemented and synchronized with ordinary user terminal I/O.

We will ignore such details as the handling of user buffers as much as possible, since they are not unique to scanner service. We will start with some relatively simple cases, and add complexity as we go. To begin with, we will assume that only TTCALLs are used for I/O, and that no unusual special characters have been enabled.

---

† Actually, once every M.STOF + 1 clock ticks. M.STOF can be defined in MONGEN. Its default value is 0. Its value must be one less than a power of two. This frequency can be patched by changing the value in the right half of location STOPAT.

12.4.1 Output

Output is the easiest case to consider. It can come from two sources: UUOCON (the user wants to print something) or COMCON (the monitor wants to print something). Terminal output is essentially non-blocking: once the output has been placed into the TTY chunks for the line, the job or process can continue to run without having to wait for the output to finish.

12.4.1.1 A Simple OUTCHR

Consider the case of the OUTCHR UUO TTCALL 1,. When the UUO is issued, control passes from UUOCON to TTYUUO in SCNSER. This routine will check to be sure that the user is attached to a terminal line (it will block waiting for that to be true), and it will also check to be sure that the terminal is at user level rather than command level. It will then dispatch to routine ONEOUT. This routine will record user response data (if necessary) and will fetch the user's character for output. If the character is not null, it will inserted in the output TTY chunks and the line will be queued for output if it was previously idle. These steps happen in routine TYO7W, which will queue the output in routine PTYPE if it is a PTY or in routine TOPOKE (typeout poke) if it is a real terminal.

TYO7W checks to be sure that there are enough free chunks available, that this is not a data line which has been detached, and that CTRL/O has not been typed to suppress the output. If these conditions are not satisfied, the output may be throw away or the user's job will block, whichever is appropriate to the failing condition.

At this point, the UUO will return to the user program, which is still runnable. The character as not yet reached the terminal, nor even the device service routine, but it has been queued to the TTY chunks for output.

So when does the output get started? During the once-a-tick code of the monitor cycle, there is a test at location STOPAT in module CLOCK1 to see whether it is time to scan the terminal output queues. If it is time to do so, we execute the instruction

```
PUSHJ   P, .CPSTO##
```

.CPSTO is a location within the CPU data block which holds the address of the terminal output routine, SC*n*TIC. This is a routine defined in COMDEV which is simply a set of calls to routines in various device drivers to start output which has been queued to formerly idle lines. Here is a list of some sample routines which could be called:

CTYSTO
CTY output service for the KS10

TTDSTO
CFE-based terminals (KL10 only)

NETSTO
ANF-10 network terminals

NRTSTO
DECnet virtual terminals

DZSTO
DZ11-based terminals for the KS10

We will ignore the specifics of any particular driver for now. It is sufficient to note that each LDB is associated with a particular queue of terminals, and that the header of that queue is known to the device service routine. This routine will then call routine TOTAKE in SCNSER to get the address of the next LDB which has been placed on its queue. Some device drivers can only start output for one line at a time, but others can start output for several lines. Any driver which can start several lines in one clock tick will have a loop which calls TOTAKE until it gives the non-skip return, indicating that the queue has been emptied.

In any case, once the device driver has found an LDB which has been queued, it will check to see why it was queued, and whether it can service the request for this line. The bit L1RCHP (change hardware parameters) may be set, in which case the driver will check to see whether any parameters have changed which its device or protocol needs to know about. If so, it will update the device characteristics. At this point, if this line is marked as idle for output, it was queued only for a characteristics change. In the case of our example, this is not so, and we proceed to determine whether we can perform the requested output. For now, let us assume that we can.

The device service routine will then call XMTCHR for this line, to obtain an output character. If this is not successful, the routine will either return to SC*n*TIC or it will loop back to call TOTAKE again, depending on the capabilities of the device. In our case, however, this call will succeed. We will do whatever device-dependent processing is required to queue this character to the eventual output device. For most of the device-dependent processors, we will not process this line further, but we will wait for an interrupt (which we have probably just initiated). For network lines, however, we may well loop calling XMTCHR and queueing characters to the output device until its message queue is filled or we empty the output TTY chunks.

Assuming a non-network line, our processing of this line is complete at clock level. Any further output processing (including additional characters queued for this line by the user program, which is still running) will wait until interrupt level.

When the device completes the request to send the character to the terminal, it will give a "transmit done" interrupt. This will cause the device driver to call XMTCHR again, possibly sending another character (or string of characters). If there are no more characters to send, the line will be marked as idle once again, and it will once again need to be queued from clock level for any additional output.

12.4.1.2 A Complication

Yes, that was the simple case. Now it is time to start adding complications. We now turn our attention to the OUTSTR UUO (TTCALL 3,). In addition, we suppose that the user has enabled TTY CRLF processing, and that the UUO begins with the line near or at the right margin as set by the TTY WIDTH command.

Once again, UUOCON will dispatch to routine TTYUUO in SCNSER. There again, we will block until we are attached to a terminal at user level. We then dispatch to routine OUTSTR. Here, we record the terminal response if appropriate, and we will address check the string

if it is not in a sharable page. This is done to avoid a page fault in the middle of the UUO, and the problem of restarting the UUO and possibly seeing the first part of the string twice†

We will then loop over the user's string argument looking for a null character to terminate it, and sending any non-null characters found to the chunks by calling TYO7W, the same routine which was used by ONEOUT. If the string is long, we will allow the monitor cycle to run by calling SCDCHK after every 200_8 words of the user's string.

Things proceed much as before, until we need to insert a carriage return, linefeed pair (CRLF) into the output. To see how that happens, we need to delve into the mechanics of the routine XMTCHR.

XMTCHR starts out by checking to see whether we have been at interrupt level too long. If seven ticks have gone by since we last ran the monitor cycle, we refuse to transmit any more characters. The once-per-second code will eventually restart any lines which this leaves in a hung state.

After that, we test to see whether the line is in a special output state. If so, we dispatch through table XMTDSP to handle the condition. In our case, at the moment, the line is not in a special output state, so we proceed to look for a character to send from the output TTY chunks. We interlock the chunk database, test and decrement the output character count, and remove a character from the output chunk stream. We check to see whether it needs expansion as a two-part character (it doesn't), and we release the interlock.

We then check to see whether we may have just counted down the number of characters to 50_{10} If so, we call XMTWAK, which will remove the user's job from TO state if it was blocked for terminal output.

We now check whether the character we extracted from the chunk stream was a meta character which needs special dispatching. Again, in our example of an OUTSTR UUO, this is not the case. Our character was not queued in image mode, and let's assume that it does not normally need special output handling.

Finally, we check to see whether we are about to try to print past the right margin, and find that we are. We fudge some bits for communication with SETCRF, the routine which will set up our free CRLF, and then we call it. SETCRF will return non-skip, so we will loop back to XMTCH1, where the test for special output states is made.

This time, however, we do have a special output state. We dispatch through XMTDSP and arrive at XMTESP. From the echo/fill chunk stream we will extract a carriage return, and finally exit through XMTCN7 back to the device driver. The next call to XMTCHR will dispatch to XMTESP and return a linefeed through XMTCN7.

The call after that will dispatch to XMTESP, find nothing more to do there, and clear LOLESP to avoid returning there again. It will then loop back to XMTCH1, where we will find that we still have a special output state. We will then dispatch to XMTREO, from which we will finally return the character which caused us to try to print beyond the margin.

† handling of the string in OUTSTR is actually a bug, even though it has never been reported as such. Can you tell why it is a bug? Can you tell why it has always worked in practice?.

12.4.1.3 Additional Complications

Now, let us consider output from the monitor, rather than the user. COMCON uses routine CCTYO rather than TYO7W, but mostly looks the same as user output. The main differences are that COMCON cannot block for output, and it can use meta characters directly.

For example, COMCON will normally call COMFLM when it wants to print a prompt. This routine will insert the meta character MC.FLM into the output chunks. When XMTCHR finds this character, it will jump off to XMTMET, which will call METFLM to handle this function. METFLM will decide, based on the carriage position at that point in the output, whether to print a CRLF.

Functions of the TRMOP. UUO will also insert meta characters into the output chunks.

12.4.2 Input Processing

Terminal input is generally more difficult to understand because of echo processing the fact that the job will probably block and need to be requeued when a break character is ready to be read by the program. Thus, we shall once again start with a simple case.

12.4.2.1 A simple case

The program performs an INCHWL UUO (TTCALL 2,). When this monitor call is issued, the job goes into TI wait until a break character is received. Only the first character in the chunks is returned at that time. Successive UUOs return the remaining characters one at a time.

When the INCHWL is executed, control passes from UUOCON to TTYUUO in SCNSER, and from there to the routine INCHWL. This routine in turn calls TWAITL to wait for a line, TYI to get a character and PUTWDU to give the character to the user. If a line has already been received, the return from TWAITL is immediate, otherwise the job will be placed in \TI wait and will not be run again until a break character is received. In other words, the job blocks.

So, how does the job get requeued to read its input? First, a line has to be received by the DECsystem–10 from the terminal. Once a character is received, the device service routine will call SCNSER at one of the following two entry points: RECPTY, for those drivers that use the LDB themselves; or at RECINT for those which use only a line number. (RECINT falls into RECPTY, so we will ignore the difference here.)

RECINT must handle many special cases before the character is ever placed into the input chunk stream. If the MIC interlock is not free for this line, the character must be deferred into the RECINT queue, RECINQ. If the line is a local dataset which needs to be ignored until its carrier stabilizes, the call to RECINT must be ignored. If this line is a remote terminal, which is not in use by a job, and the system is stand-alone, the user will be told that without the chunks ever being touched.

After all these tests, which merely determine whether this line is allowed to receive input, the character which has been received must be checked to see whether it needs special processing. Most special processing will be avoided if the line is in image mode or packed image mode (PIM).

If the character is to be received, and it does not need special processing at this time, and there is sufficient room to store the character into the input chunk stream, then the character will be stored by SCNSER. If the character is stored, then the user's program may receive a software interrupt informing it that input is available. Finally, TOPOKE will be called, to start echo processing, and SCNSER will return to the device service routine.

Echo processing is handled through XMTECH. When no characters are present in the output chunk stream, XMTCHR jumps to ZAPBUF. ZAPBUF is responsible for waking up the user's job if it was blocked in TO state, and for calling XMTECH to provide echo of the user's input. XMTCHR will only give a non-skip return if it cannot continue with the output stream due to a characteristics change, or if we need to execute the monitor cycle, or if neither output nor echo need to be done.

XMTECH will check whether echoing is needed or even allowed, based on such things as whether a CTRL/R is being processed, or whether deferred echo is in effect and the program has not asked for any input. XMTCHR is also responsible for most of special character processing. This includes such things as the conversion of a space to a CRLF when an automatic carriage return (ACR) setting is in effect, the conversion of lowercase input to uppercase, and the input line editing features of CTRL/U, CTRL/W, and CTRL/R, etc.

The processing of such special characters is left to XMTECH, rather than being handled in RECINT, so that a user who sets deferred echo will see the characters behave as expected according to the I/O mode of the program which reads them, rather than the modes in effect at the time they were received (possibly as typeahead) by the DECsystem–10. XMTCHR will handle expansion of two-part characters, and the processing of break characters. When a line break is examined for echo processing, either the user's program or COMCON may be notified of available input.

If the character requires echoing, it will be returned by XMTECH on behalf of XMTCHR. If not, XMTECH will keep advancing the chunk stream until no more echo processing can be done.

Finally, after XMTECH has requeued the user's job, it calls TYI, as previously mentioned. This routine, like all terminal input routines, will eventually reach TYICC4. This routine checks for interlocks with XMTECH due to CTRL/R processing, and eventually will either return a failure to its caller (if in asynchronous mode) or will return a character (possibly with expansion in the case of two-part characters).

12.4.2.2 Complications

Control characters tend to need special handling. Some few of them need it at RECINT time, but most can wait until XMTECH. Some of those which cannot wait are CTRL/O, CTRL/C, CTRL/S, and CTRL/Q. Some which can wait are CTRL/R, CTRL/W, runout, and carriage return.

When a rubout or similar editing character is processed, it would like to be able to leave a hole in the TTY chunks. Since this is not feasible, however, it marks the deleted characters with the bit CK.NIS (not in stream). Both XMTECH and TYICC4 will skip over any characters so marked is though they had never been present.

When a carriage return is processed, it might want to become a CRLF. Since it is in the middle of a chunk, and might have other characters following it already, this cannot be done. Instead, it becomes a the meta character MC.NL. This meta character will be expanded as a two-part character into a CRLF, as was desired.

When an eight-bit ASCII character is received for input to a program which is running in seven-bit mode, it must be translated to the appropriate seven-bit fallback representation. This usually involves expansion into two or three characters. Similarly, output from a program using eight-bit I/O which is destined for a seven-bit terminal must be expanded.

In order to see how control characters, two-part characters, and other special characters are handled, look at the tables CHTABL, METABL, CHREQV, and METEQV. Also examine the routines TPCOUT, TPCECH, TPCINP, and TPCCOM.

12.5 Pseudo-Teletypes (PTYs)

For detailed information on the purpose and use of PTYs, see the Monitor Calls Manual Here, we only note the distinction between the different kinds of PTYs. There are old-style PTYs , the newer or full-SCNSER PTYs, and the latter are frequently given special handling when they are in use for batch processing, so we can say that batch PTYs are a third kind.

The old-style PTYs are not capable of supporting terminal types, eight-bit ASCII, or indeed most of the special features which distinguish a terminal from any other bi-directional I/O device. Routine PTYPUT in SCNSER handles input destined for a terminals which are associated with old-style PTYs.

The full-SCNSER PTYs are capable of supporting anything which a normal user terminal can support, with the exception of image mode I/O. Packed Image Mode (PIM) I/O is allowed, however. This flexibility of full-SCNSER PTYs makes them quite useful for virtual windowing programs, but the fact that they will support CTRL/S and CTRL/Q , along with TTY STOP mode, can sometimes get a user into trouble.

The batch PTYs do not do free CRLF processing or similar such terminal-specific formatting, but since they are otherwise a full-SCNSER PTY, they do support most terminal functions. In case of any doubts, search in SCNSER for references to the routine PTBTCH (defined in PTYSER). This routine is used to confine the full terminal functions to full-SCNSER PTYs which are under the control of interactive jobs. Search also for references to the bit LDLFSP, which is the full-SCNSER PTY bit in the LDB.

12.6 Macro Interpreted Commands (MIC)

The MIC facility is a feature of TOPS–10 that allows a user to execute a command file at the terminal. The commands are processed in a similar manner to BATCH commands with several notable exceptions. The commands are processed for the user directly, not through another job logged in on a PTY. The user sees the commands and their results printing directly on the terminal. The batch controller (BATCON) is not involved at all. For a complete description of the features and operation of MIC, see the documentation supplied with it on the distribution tapes. This discussion is concerned with the special processing in SCNSER to accommodate MIC.

The MIC system revolves around a copy of MIC.EXE which is always running as a detached operator job. That one copy of MIC controls all jobs that are using the MIC facility. In the low segment of this MIC "master" is a process data block (PDB) for each job wanting to use MIC. This PDB holds such items as the file from which to fetch commands, the arguments from the DO command line, and information about labels with the file. The master responds to the needs of the "slaves", feeding them command lines from the appropriate files. The command lines are sent directly to the terminal's input chunk stream where they can be processed by the usual means.

When the user issues a DO command, COMCON sets up and runs the MIC program. The user enters a section of code different from that of the MIC master, setting up the PDB for itself. Once the user exits from MIC, the master takes control. It will open the command file. For each line in the command file, it resolves the parameters and then issues a TRMOP. UUO, function .TOTYP. This places an ASCIZ string directly into the terminal's input chunks. The routine in SCNSER to handle this function is TOPMTY. Characters are entered one by one via calls to RECINM, which is in the receive interrupt routine. SCNSER then performs the usual echoing, eventually notifying COMCON or the scheduler when a break character is received.

SCNSER also watches for the ERROR character and the OPERATOR character when they have been set. It notifies the MIC master job of any changes in state of the user terminal or job in which MIC might be interested. When not servicing slaves, the master executes a HIBER UUO. Thus, SCNSER takes care of notifying MIC of significant events simply by awakening that job. This is handled in the routine MICWAK.

Note that there is nothing to prevent the user from typing during the processing of a MIC command file. However, if the master decides to type while the user is typing, neither is likely to get the intended results.

Appendix A
EBOX/MBOX Accounting

A.1 Summary

A customer finds that 20% of his KL goes missing – lost or not charged. This appendix describes the problem and questions the idea of accounting by time.

A.2 Introduction

We recently had a problem on-site in which it appeared that a great deal of computer time was being lost in prime shift. That is, when we totaled over a prime shift the figures for lost, null, and overhead, and added these to the time charged to users (from the accounting files), we could not account for 20% of the elapsed time. Stated like that, that's a significant amount of the KL that was being lost or given away, and we were required to find it.

Note

The site was a 1090 with 512K of MH–10 Memory.

A.3 Explanation

The problem primarily comes down to EBOX/MBOX accounting and cache. This was noted by Claude Barbe in "Copy'n Mail", in which he concluded that:

Note

"EBOX/MBOX accounting is as good as your cache hit ratio"

Unfortunately even this is not quite true. When EBOX/MBOX accounting is used, use of the system is measured by two hardware meters counting micro instructions (EBOX) and memory references (MBOX). This is converted to time by the use of two divisors, which are CPU-model dependent.

The MBOX counts all memory references, irrespective of whether the required data is in cache or not. Its divisor is such that, if a program of the form JRST . is running, then the total time charged (that is, null + lost + overhead + user) approximates, very closely, 100% of elapsed time. JRST . is completely cache effective.

If a completely cache ineffective program is running (for example, a large JRST . + 4 loop) the time charged is around 18% of elapsed time.

So far this seems to agree with Claude's statement. However, in attempting to map a relationship between cache hit ratio and charged time during prime shift, we find that similar cache hit ratios give widely differing charged times, and similar charged times come from different cache ratios. Part of the reason for this is that a cache hit is two quite different things.

The whole problem condenses rather neatly into two simple programs. Consider:

```
        (A)                              (B)

SETOM   (T2)                     MOVE    T1,(T2)
ADDI    T2,1000                  ADDI    T2,1000
AOBJN   T3,.-2                   AOBJN   T3,.-2

                (outer loop control)
```

- Now, the program JRST . was 100% cache hit, charging 100% of elapsed time.
- Program (A) here is 100% cache hit, charging 74% of elapsed time.
- Program (B) here is 76% cache hit, charging 53% of elapsed time.

The obvious questions here are:

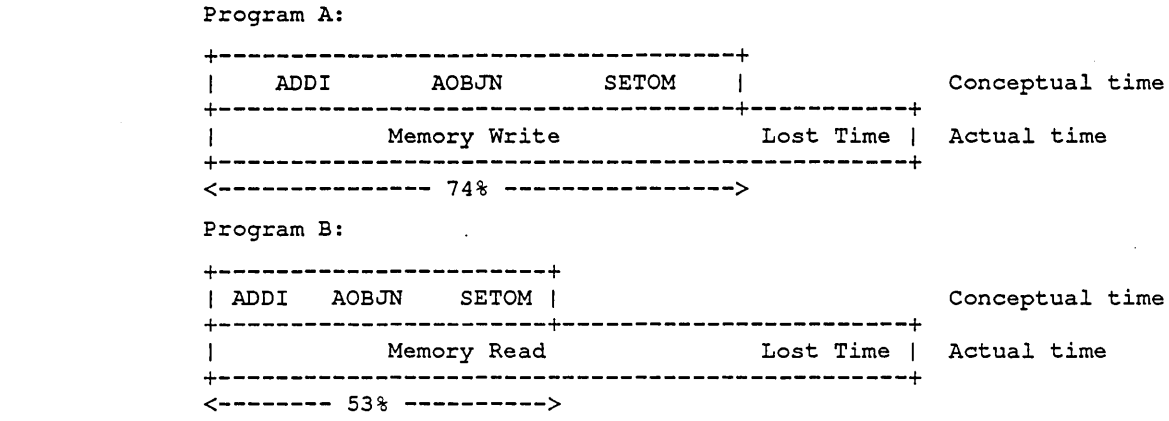
1. Why does A at 100% cache effective charge 74% while *JRST* . charges 100%?
2. Why is Program A 100% cache effective and Program B only 76%?
3. Why should Program A charge 74% but Program B charge 53% of elapsed time?

It is best to answer the second question first. All B's (and A's) instructions are in cache since they are at a virtual page offset which does not clash with the page offset of the memory reference, and therefore need never move. B is doing a read every third instruction, which always fails to find a match in cache, since there is only space there for four memory locations of page offset 0 (or whatever). The three instruction fetches +1 memory fetch give us a cache hit ratio of three in every four, or 75%. Program A is the same but does a write to memory instead. It appears that, because we always put the write in cache we score a cache hit, whether or not it was first necessary to write away valid written data. Because a cache hit is now two different things—no memory reference required on a read, on a write it matters not—Program A is 100% effective, against Program B's 76%. (Strictly, of course, a cache hit can be consistently defined as the referenced data being put into or gotten from cache with no regard to whether a memory referance was required, but that differs certainly from my original conception of it).

The clue to the differing charged times came in the ratio 74% to 53%, which nicely matches the ratio of read time to write time for MH memory, that is, 1.767 to 1.267. Observe what happens with Program A in Figure A–1.

The SETOM requires that a valid written word in cache first be put in memory. We must now wait for the memory cycle to finish before we can fetch and execute the ADDI and AOBJN, and fetch the SETOM. All of these come from cache, and do not reference memory. Execution of the SETOM then causes another memory reference, and so on.

Figure A–1: Comparison of Program A and B Charge Times



Program B is very similar but requires a read from memory. With the much larger memory cycle time involved, we charge for an even lower percentage of the real time, only 53%.

Obviously, varying these two themes on the cache availability of the instructions, this effectiveness on cache, and whether we read or write, allows for a totally variable charge range.

A.4 Why EBOX/MBOX ?

If EBOX/MBOX accounting causes such problems, why use it? The answer is that it provides beautifully consistent user accounting. Regardless of system load, time of day, or scheduling vagaries, when a user runs a job it always costs the same amount. This is a good thing, and compares favorably with KI accounting, which may in extreme cases show a factor of 15 difference in charges between a Sunday run and a prime-shift run on the same job. Unfortunately, in being so consistent, EBOX/MBOX gives the internal accountants, that is, the computer department in whatever form, a real problem. How do they charge for the missing time?

It is my belief that all these problems are caused by the historical practice of accounting for computer usage by time. The EBOX/MBOX meters provide a consistent, tidy account of CPU usage in digital units, a measure of work done (Klergs?). This is rightly quite independent of prevailing conditions. The real error comes in reducing it to time values. It is a matter for site accountants to fix, by the normal in-house procedures for dealing with overheads, the means of rating Klergs directly in cash terms, taking account of the probability range of the site-specific percentage of "hidden" time taken by users. They must avoid the unnecessary, and demonstrably illogical, intermediary step of roughing it out in units of time before converting to cash. Monitor and documentation should certainly avoid the same trap, and refer to accounting units as such and never in time units. After all, no one would expect to charge for line printer output on a time basis, using the optimum time required to fill a page.

A.5 Conclusions

1. EBOX/MBOX accounting is beautiful and consistent.
2. There is a philosophical error in converting EBOX/MBOX units to time.
3. Sites selecting EBOX/MBOX should realize and utilize the consistent digital measure of work done returned by the two meters, converting that measure directly into cash independent of time.
4. Digital's accounting and documentation should match this thinking and a write into cache should only be considered a cache hit if no memory write-back is required.

Index

A

B

C

D

E

Q

R

S

T

U

V

W