TOPS-10 MONITOR INTERNALS

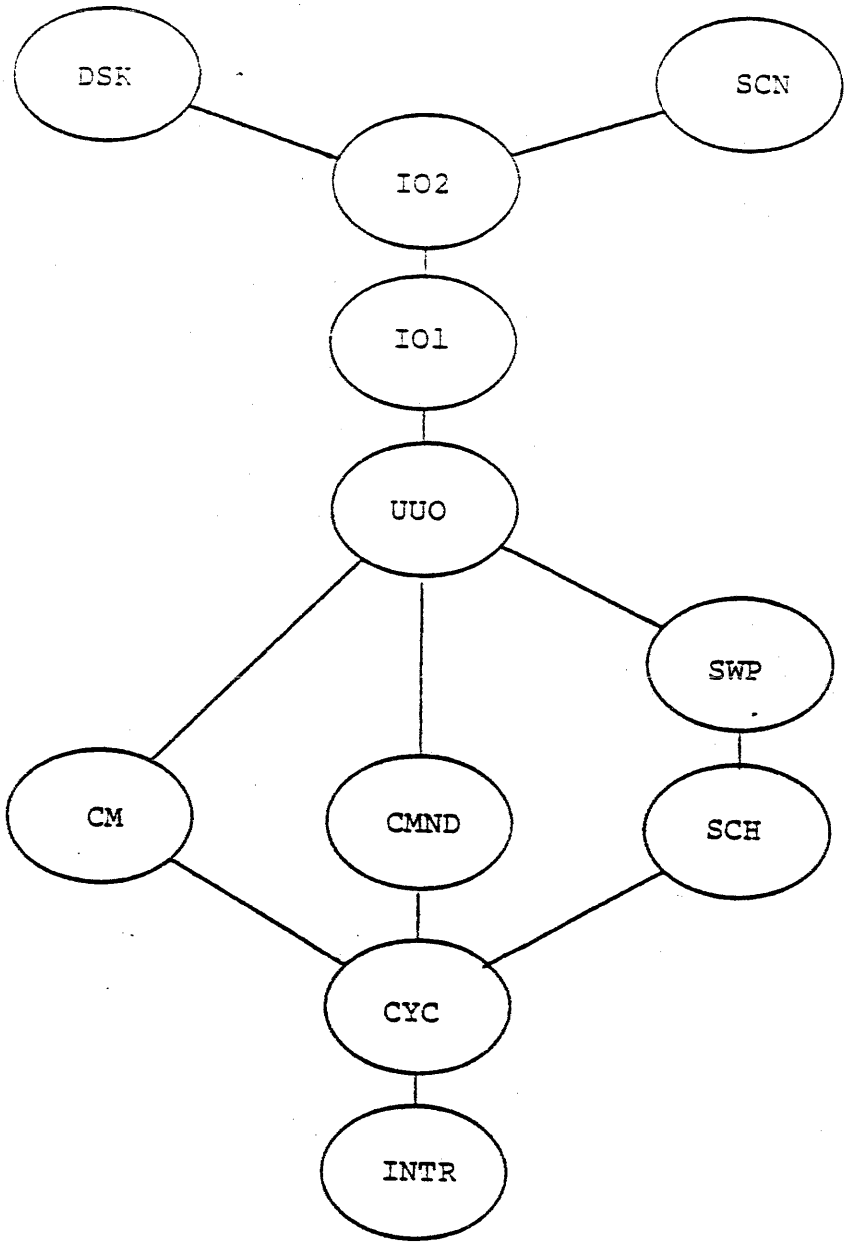
Educational Services
Digital Equipment Corporation
Bedford, Massachusetts
Revision 6, November 1980

EY-CD150-HO-006

CONTENTS

TOPS-1Ø MONITOR INTERNALS

Introduction to TOPS-1Ø Monitor



COURSE MAP

This page is for notes

INTRODUCTION TO TOPS-10 MONITOR

INTRODUCTION

This module prepares the student for the later modules by introducing basic information about the KL10 processor and the TOPS-10 monitor. Included are such topics as: monitor addressing, coding conventions, monitor building, the EPT and using microfiche.

RESOURCES

Microfiche of the TOPS-10 Monitor

TOPS-10 Monitor Internals Course Supplement

Course Material - Chapter 1
Supplement - KL Document Pages 1-14
38-45, 63-65
- Graphics Section 1

LECTURE OUTLINE

Introduction to the TOPS-10 Monitor

I. Course Introduction And Administration

II. Monitor Addressability
- A. Review of Paging
- B. Page Map Entries
- C. Monitor and Mapping
- D. Per Process Area
- E. Exec Execute
- F. Exec Virtual Memory

III. The Monitor as an Event Processor
- A. Traps
- B. Interrupts

IV. EPT Revisited

V. Monitor Components and Building

VI. Monitor Coding Conventions

VII. Monitor Tables

VIII. Microfiche Usage

IX. GLOB

X. Universals Files

This page is for notes

INTRODUCTION

The DECsystem-10 consists of hardware and software designed to allow users to run a variety of programs efficiently and conveniently. Throughout most of this text, we shall be concerned with the details of the system software. Initially, however, it is necessary to develop some basic concepts and define terms involving both the software and hardware.

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

User Program Addressing

The DECsystem-10 has several hardware features 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 core memory only within areas assigned to it by the monitor. Also, certain instructions can not be executed in user mode. These include all I/O instructions, and the instructions to control memory access and mode of operation.

Each program consists of instructions, constants, and data areas, which may be either one or two segments of the user's virtual space. The address provides a mapping from a virtual address to a physical address.

The users virtual address space, as well as physical memory, 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 assignes physical pages to a user's segment, it builds a page table, telling where each of the user's virtual pages resides in core.

Thus, in the KL-10 processor the mapping of a user virtual address to physical address is accomplished by the user's page map.

On an indirect memory reference, the paging mechanism is used for each memory reference made in the effective address calculation. The addresses 0-17 always refer to the hardware accumulators. The KL has 8 sets of accumulators or fast register blocks; three sets in use by the monitor, one set for the current user and one set for the KL microcode. The other 3 sets are available for real-time programs. The KI-10 has 4 sets of fast register blocks (ACs). The monitor uses one set. The current user another, and the other 2 sets are available for real-time programs.

Also associated with each user's page is an access bit which provides protection for the monitor and other user programs when this user is running. The monitor fills the page table and sets the access bit only for those entries which are allowed to be accessed. A zero access bit in the page table will cause a reference to the associated page to initiate a page failure trap to the monitor.

Before the monitor allows the user's program to begin, it passes the address of the user's page table to a hardware register called the user base registers, UBR. Now it's ready to start up the user's program in User Mode. It does so, and the hardware, because it is executing in User Mode, uses the specificed user's page map to map the virtual addresses into physical addresses.

To speed up memory referencing time, (current scheme would require two actual memory references: one to obtain the memory mapping data and one to obtain the user's mapped memory reference) the last 512 distinct pages referenced (32 on a KI) have a copy of their associated physical page numbers and access bit information stored into a special memory in the Central Processor. Thus, only if the information of the page referenced is not in this special memory (called a KL Hardware Page table or a KI Associative Memory) must 2 actual memory references into core be made.

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 (virtual address space) is divided into 2 parts or segments a pure or reentrant segment and an impure segment. The reentrant segment will normally consist of all the constants and instructions which do not change during the program execution. Since this part of the program does not change, a single copy in physical memory may be shared by more than one user program. That is the same physical page numbers for the pure segment will appear in more than one page map. All parts of the program which are subject to change must be set up separately for each user.

The impure segment of the program begins with virtual address 000000 and can go as high as 777777. However, that would leave no room for the pure segment. The pure segment usually begins with virtual address 400000 extending as high as 777777.

Because the virtual addresses in the pure segment are 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 these 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 user virtual addresses in the middle of the range of valid ones.

Monitor Calls

The monitor performs a number of services for user programs, including all I/O operations. The instruction codes from 040 through 077 provide the means for programs to request the monitor to perform these services. The operation codes, called Unimplemented User Operations or UUOs, have no hardware function except to give control to the monitor. When a UUO is executed, a routine in the monitor 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."

Interrupts

The KL-10 processor has a multiple level priority system. There are seven levels of priority with level one being the highest priority and level seven being the lowest. Each I/O device is assigned a level and can interrupt any programs running at a lower priority level. Interrupts can also be requested programatically.

When an standard interrupt occurs on level N, the next instruction is taken from physical address 40 + 2*N, and is executed in executive mode. When a vectored interrupt occurs the device/controller requesting the interrupt supplies, to the CPU, a Function word the contents of which is used to determine what instruction should be executed to service the interrupt. This allows transfer of control directly to the device service routine rather than a Fixed address of 40 + 2*N as with a standard 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 which interrupts 60 times per second (actually at power line frequency if it makes a difference). This clock interrupt guarantees that the monitor will always get control back from a user program. One sixtieth of a second is, therefore, the basic unit of CPU time which the monitor allocates to a program. Upon each clock interrupt, the monitor reconsiders the question of which programs to run.

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 as a service. A function performed without a user request would then be a control function.

Requests for service can come from user terminals, as monitor commands, or from user programs, as UUOs. The most important command is the command to run a program. In response to this command, the monitor assignes core memory to the user job, reads a executable file into core from some other storage medium, and adds the program to the set of programs sharing the CPU. The most frequent requests for service from programs are 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 disk, starts I/O transfers, and handles the resulting I/O interrupts.

Control functions are performed as necessary by the monitor, according to algorithms which attempt to give optimum overall system performance. One of the most important of these functions is dividing the available CPU time among the active user programs. Jobs must be stopped when clock interrupts occur, and their computational 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 core and which to "swap out" to drum or disk memory. In addition, it must decide where to put jobs in physical core as they are swapped back in and when they change in size.

Structure of the Monitor

The monitor consists of many separate and more or less independent routines and modules which are called according to events which occur within the system. Figure INTRO-2 shows a functional diagram of the major routines, and the control paths between them. 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 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 is the routine which handles commands typed by users. It frequently calls the SAVE/GET routine and the Core Management module in processing commands to set up and run various programs. The Scheduler is the routine which decides which user program to run during the next jiffy. The Context Switching routine restores the computational state of the chosen job and then allows 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 core memory and drum or disk, attempting to keep the highest priority jobs in core.

The UUO Processor responds to all requests for service by user programs, and specifically handles all I/O required by user programs. The UUO processor itself is device independent. It will be the same in all monitors, regardless of the hardware configuration for which they were built. The device dependent code requred for any device will be included in a device service routine for that device. Any given monitor will be built for a specific hardware configuration, and will contain device service routines for the devices in that configuraton.

The Core Management module handles all changes in size of user jobs and all changes in their physical core locations. It is 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 transfer. The interrupt 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 be disabled for a few instructions when it is critical that no interrupt should occur.

The Monitor as an Event Processor

Overall the Monitor can be envisioned as a real time program which responds to events which occur within the system. The routines which 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 console. Each I/O device interrupt is an event which results in the execution of a specific interrupt routine.

There is a well-defined function which the monitor performs in response to each event. However, a given event will not necessarily result in the same action every time it occurs. The specific action taken on a given event depends on the state of the system. The system state is represented by many variables in core memory and device registers and depends on the past history of the system. Given the state of the system, the monitor will perform a specific predictable function in response to any specific event.

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. It switches control among the user jobs so that each user appears to have a system to himself. The Monitor runs and stops user programs according to the user's commands. It handles all I/O operations, according to requests from user programs. It attempts to allocate all system resources in such a way as to give the best overall system performance.

EXERCISES

This laboratory exercise requires the student to use the microfiche of the current TOPS-10 monitor. For each starred (*) question, the student should supply the module name(s) and CREF line number(s) of the relevant code, e.g., COMMON line 4627.

*1. What does JPOPJ1 do?

*2. Describe the PJBSTS byte pointer.

*3. Where are CPU data blocks defined? How many were defined for this system? Why?

*4. Where are the definitions and redefinitions of the C and V macros used in constructing the CPU data blocks?

*5. What location contains "overhead" time?

*6. Where is the serial number of CPU1 stored for a dual-CPU system?

*7. Find the definitions and explain the interrelationships between M.JOB, JOBMAX, JOBN, SEGN, and MD.SEG.

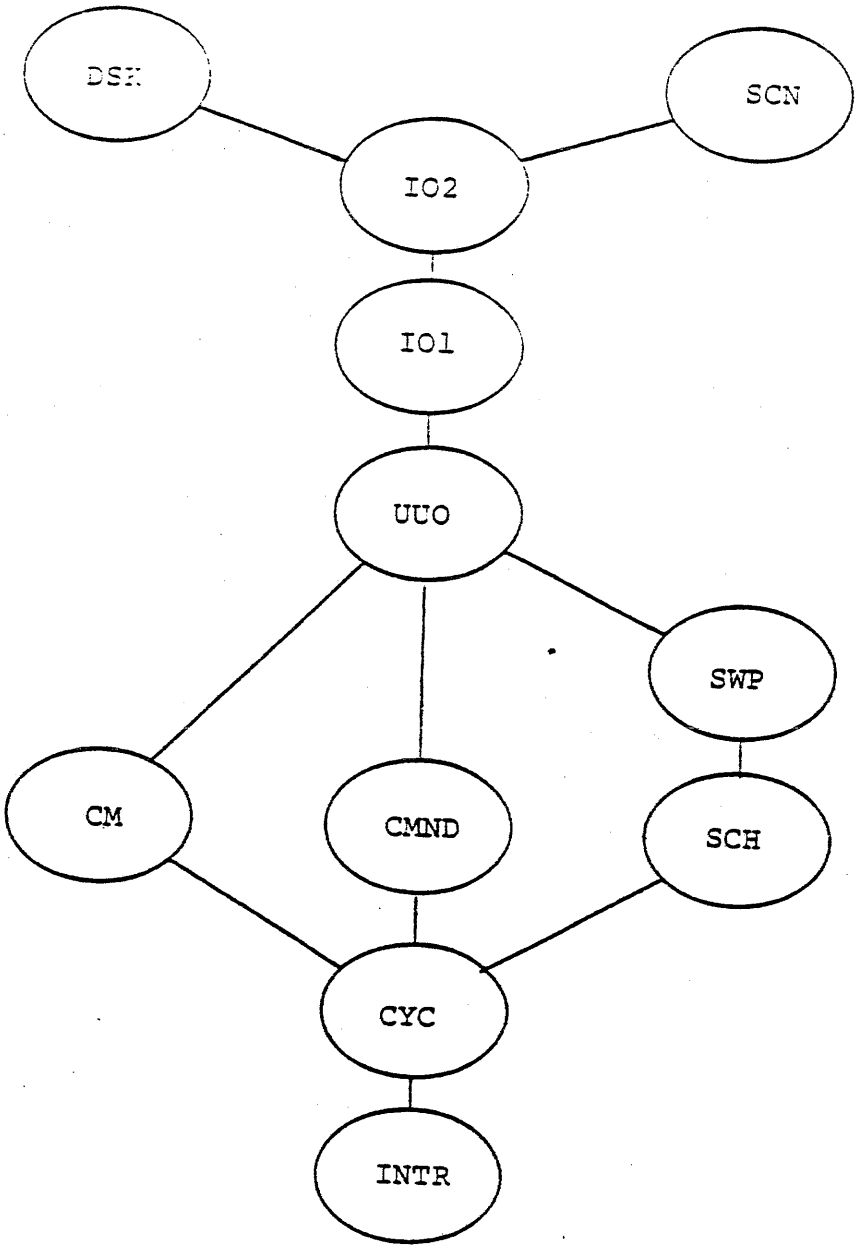
*8. Which feature test switches must be on in order that the JBTUPM table will be generated? How big will this table be?

*9. How is the decision made whether or not to load the device service routine for the incremental plotter? Is it loaded in this monitor?

*10. Describe the effect of the SAVE4 subroutine on the pushdown list. What happens if a call is now made to CPOPJ1?

TOPS-10 MONITOR INTERNALS

The Monitor Cycle



COURSE MAP

This page is for notes

THE MONITOR CYCLE

INTRODUCTION

The heart of the monitor is the monitor cycle. Time accounting, command processing, scheduling, swapping and context switching takes place in this cycle every clock tick. It is this cycle that allows TOPS-10 timesharing to work effectively by reallocating resources periodically. All knowledge of the monitor is built upon an understanding of this process and serves as the real starting point for the course.

RESOURCES

TOPS-10 Microfiche

TOPS-10 Monitor Internals Course Supplement

Course Materials - Chapter 2
Supplement - KL Document Pages 33-37 -
Graphics 2-All

LECTURE OUTLINE

I. The Control Routine
   A. What and Why
   B. Overview Level 3 and 7 Timing Charts
   C. Level 3 Detail
   D. Level 7 to RSCHED
   E. Level 7 RSCHED to End
   F. Partial Cycles

THE CONTROL ROUTINE

Every sixtieth of a second the Monitor repeats an overall cycle which ends with giving control to a user program. In this cycle the monitor performs all functions which are repeated on a regular, periodic basis. These include all monitor funcitons except servicing interrupts and UUO's. The functions which are performed in-line within the control routine will be discussed in this chapter. We will be particularly interested in the manner in which the major routines are called, and the circumstances under which the entire cycle is repeated.

Time accounting

At the beginning of the cycle, we account for use of the CPU during the previous cycle. If a user program was running, we add the elapsed time to the total for the job in its Process Data Block (PDB). If the Null Job was running we add the elapsed time 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 which could not be run for some reason, the time is considered "lost" CPU time. If there was no job in a Run Queue 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 which might seem peculiar at first. Original KA systems measured time in sixtieths of a second, or "jiffies".* Various system programs such as SYSTAT, and possibly some user programs, look into the Monitor's tables, and expect these times to be in jiffies. It has been found desirable to measure CPU times with a higher degree of resolution then the jiffy; this is useful for charging usage of the machine for partial jiffies. Hardware is available to measure time internals with 1Ø micro-second resolution, the DK1Ø real time clock (KA and KI) and the time base meter on the KL. In order to use the additional precision, without having to change all the programs which look at these time intervals, we maintain the interval in two parts. The table entries which have historically been in jiffies are still in jiffies. An additional word is used to hold the excess beyond the last

*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.

even jiffy, in units of jiffies*10-5. When the interval is updated, the incremental time is added to the excess. If the excess goes past one jiffy, then the excess and the jiffy total are each corrected. The code which updates time intervals is written so that it will work with no changes on systems which do not have a DK-10 or time base meter. On these systems it will simply appear that every interval happened to be an intergral number of jiffies.

On systems using a DK10 or 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 excluded or included in user runtime as determined at MONGEN time.

KL processors have two additional clocks, or meters, that can be used for even a higher degree of accuracy and repeatablity than the DK10 or time base meter. These meters are called the EBOX and MBOX meters. The EBOX meter counts EBOX cycles during instruction execution while the MBOX meter counts memory references. These meters can be operated in a mode such that they are stopped during interrupt processing hence the current job will not accumulate EBOX cycles during interrupt processing. The EBOX and MBOX counts are scaled by the appropriate scaling factors to be equated to CPU run time as obtained from a DK10 clock or time base meter. Job accounting via the use of EBOX - MBOX accounting is more accurate 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.

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

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 has been set up, we decrement the remaining time each time the job accumulates another jiffy of CPU time. When this time limit, in table JBTLIM, expires the job will be stopped. A timesharing user can type a CONT command, but a batch job will be aborted by the Batch Controller.

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, i.e. waking a job that is just going to sleep. Each request includes the address of the routine to be called, the time interval in jiffies, and seven bits of data to be passed to the routine. The requests are stored in 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. A counter is decremented on each clock tick to indicate when another second has elapsed. Each time this counter expires, the Once-a-Second routine is called. Among other things, this routine checks for hung I/O devices. it also maintains a counter, and calls a Once-a-Minute routine.

Major Routines

Next, the monitor checks if there are any commands waiting to be processed. If there are, the monitor calls the Command Processor. The Command Processor will choose one of the waiting commands to interpret and process, and will then return to the Control Routine. The Command Processor is a major routine, which will be treated in detail in a later chapter.

The scheduler, which is also a major routine, is called next. The Scheduler requeues any jobs which have changed state during the last cycle, and then determines which user program should be run on this cycle. It also calls the Swapper, which is another major routine. Although the

Scheduler determines which user program is to be run next, it does not give control to that program directly. Instead it returns to the overall Control Routine which called it.

Context Switching

Before going to the user program, the monitor must restore all conditions which affect its execution to the state which they were in when the monitor last interrupted it. The same conditions must be saved for the user program which ran last. The software conditions are represented by data stored in a part of the user page map page (JOBDAT on non VM systems). On non-VM systems this information, which must not be changed except by the Monitor, is copied from the Job Data Area into the CPU Data Block before the program is given control. When the program is stopped, this information is copied back into the Job Data Area. This portion of the Job Data Area is said to be protected, since it is put out of reach from the program while the program is running. The Monitor also prevents the user from doing input into this area, or changing it with a D (Deposit) command. The information contained in the protected part of the Job Data Area includes which devices are being used and the address at which the program should be continued.

When virtual memory was implemented this protected part of the JDBDAT was moved to the UPMP to increase its integrity.

The hardware registers which must be restored are the User Page Map Base Register, the 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 User Page Map (UPMP) register is maintained in table JBTUPM, having an entry for each job in core. Restoring these hardware registers is the final action before giving control to the user program.

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 which is not yet available. If a clock interrupt occurs during execution of a UUO, we do not interrupt the job at that point, but allow the UUO to run to completion. The job will then be stopped

in the exit routine of the UUO processor by software action. 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.

2. Clock interrupt occurs during execution of a UUO, and then the UUO is completed.

3. A UUO routine reaches a point where it can not immediately continue. (A clock interrupt may or may not have occurred.)

Clock Interrupt

The interrupt which causes the monitor to begin a monitor cycle originates from the arithmetic processor, device APR on a KA or KI, or the interval timer, device TIM on a KL. These devices are assigned a very high interrupt priority, usually level 1, 2 or 3 for two reasons: (1) the interrupt triggers the maintenance of time of day which must be accurate, (2) the APR device (KA & KI) also interrupts the CPU for various errors which must be processed immediately. 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 which requests the Channel interrupt; the flag .CPTMF is set only by the interval timer (APR) interrupt routine and indicates that an actual clock tick has occurred.

When the software clock interrupt occurs, control passes to the CKØINT routine in KLSER. If the interrupted program was in exec mode, the interrupt is immediately dismissed and the new cycle is delayed until the current monitor function is finished. Otherwise, the user's PC is saved in the Job Data Area, control and the overall cycle is repeated.

The UUO Processor checks if a clock tick occurred while it was running, before returning control to the user program. If it finds .CPTMF set, it passes control to the USCHED routine, which performs a similar function to CKØINT. USCHED sets the "user program" return address to the next address in the UUO Processor. It then saves the necessary AC's and passes control on to RSCHED. Whenever 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.

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, will set up the return address, save the AC's and pass control to RSCHED. When the program is restarted it will be at the point where the monitor routine requested a new cycle. Some functions on the overall cycle will not be performed if the clock has not ticked.

Saving the PC

When the software clock interrupt occurs, the first instruction executed is a JSR. This JSR saves the PC word for the user program at location CKØCHL. If the interrupted program was in user mode the contents of CKØCHL are copied into .CØPC in the CPU Data Block. If the job is not choosen to run nexts its PC is copied from .CØPC in the CDB to JOBPC in the users UPMP. The time during which the program is not running its PC word is preserved in JDBPC. When the job is chosen to run again, the opposite process occurs. The contents of JOBPC is copied into .CØPC during context switching, and control is given to the user program with the instruction:

```
JEN @ .CØPC
```

After execution of this JEN the PC and processor flags have the same value which they had when the program was interrupted.

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

In returning control to the user program, we do not need to be concerned with the manner in which it was stopped. When the PC word is restored from the contents of JOBPC, the program will continue running in the correct state. If it was interrupted in user mode, it will continue with the next instruction that would have been executed. If the program was stopped by a PUSHJ to WSCHED or USCHED, it will continue 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 arbitrary assortment of other programs.

This page is for notes

EXERCISES

All questions marked by a "*" should be answered by specifying routines and line numbers.

*1. Where is control passed to RSCHED?

*2. Where does the monitor give control to a user program? I.e., where is the very last instruction executed in the monitor before control goes to a user job?

*3. At what location is the user program's PC stored when a channel 7 clock interrupt occurs?

*4. Under what circumstances is a jiffy counted as "lost time"?

*5. Where is the job's PC word saved when it is not the current job?

*6. Assuming a job blocks for I/O and transfer of control flows to WSCHED. How does WSCHED supply RSCHED with the PC word to restart the job once I/O is complete?

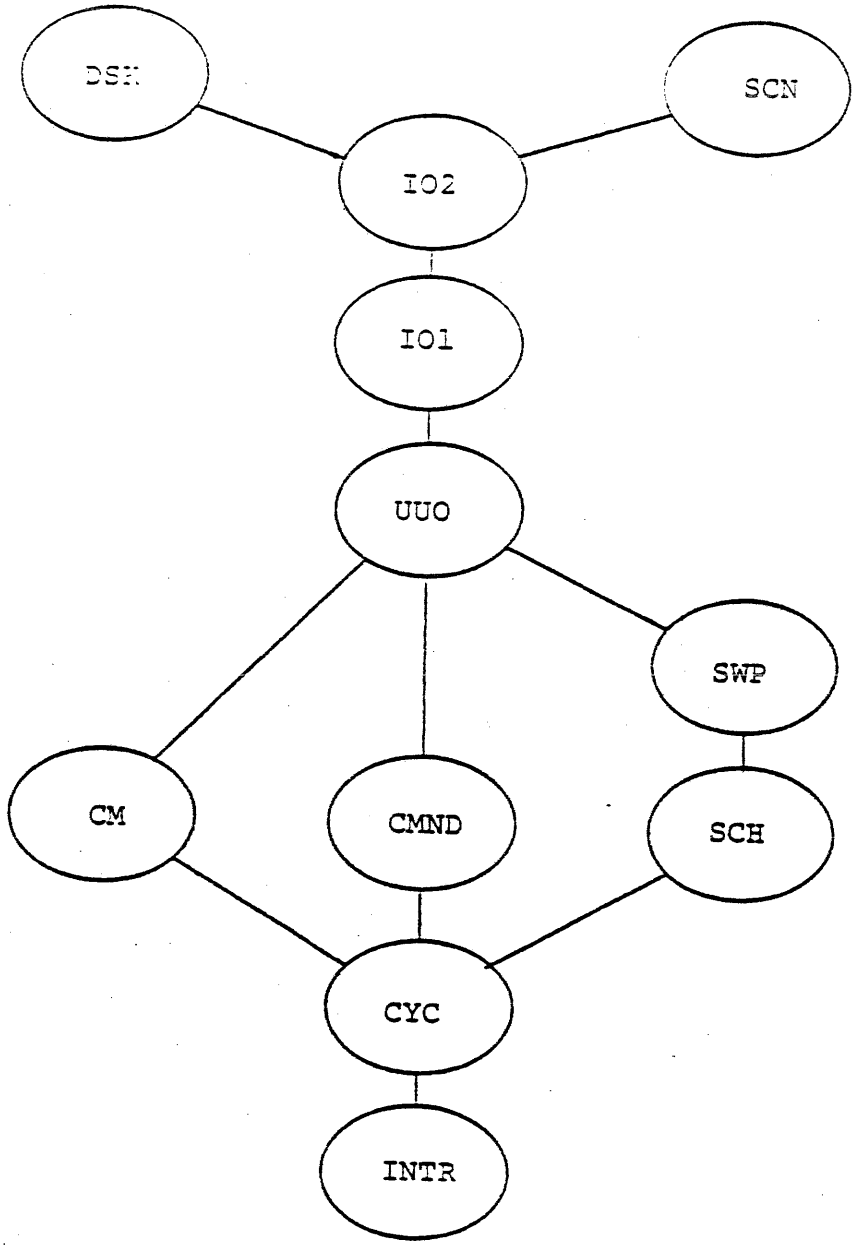
*7. How do USCHED and WSCHED differ? How do you account for these differences?

8. Under what circumstances would a job be picked to run by the scheduler, set up by the context switching routine and then be started up in exec mode?

*9. In what parts of what modules is context switching performed?

TOPS-1Ø MONITOR INTERNALS

Core Management



COURSE MAP

This page is for notes

CORE MANAGEMENT

INTRODUCTION

KL1Ø memory is organized into pages that are shared between the monitor and user jobs. The core management modules handle memory by manipulating the monitor data base and the page maps. Although system programmers rarely would change the core management routines or data base, a knowledge of how they work is necessary because many other features (such as virtual memory) depend on it.

RESOURCES

Monitor Table Descriptions

Microfiche of TOPS-10 Monitor

TOPS-10 Monitor Internals Course Supplement

Course Material - Chapter 3
Supplement - Graphics 3-A11

LECTURE OUTLINE

I. Core Management
   A. Introduction
   B. The Data Base
   C. Demands on CORE1 and General Flow
      1. Core Command
      2. Core UUO
      3. Swapper
   D. Allocation and Assignment Flow Charts
   E. Setting up a UPMP
   F. Virtual Memory

INTRODUCTION

Core management is concerned with the management of both physical pages (memory) and virtual pages (disk space used for swapping and paging). This activity can take the form of simple bookeeping or high level decisions about swapping and the running of programs. This module will discuss the bookeeping functions performed by the monitor modules COREl and KxSER (KLSER for KL10 processors or KISER for KI10 processors).

These modules can be accessed by three different means: 1) by the command decoder (COMCON) when the CORE monitor command is issued, 2) by the UUO handler, UUOCON, when a CORE. or PAGE. monitor call is issued and 3) by the swapper.

Allocation Vs. Assignment

For this discussion, the following terms will be defined:

Allocation - Management of virtual memory (swapping space)
Assignment - Management of physical core

Core management is comprised of both allocation and assignment phases.

CORE MANAGEMENT DATA BASE

PAGTAB - is a table containing many linked lists for all physical pages in the system. There is a one word entry for every physical memory page in the system. All the pages for a job are linked together within PAGTAB. Each entry contains the physical page number of the next entry (page) in the list. For every job, logical page zero is always first in the list. The RH of UPMP+400 points to the physical page number of page zero, i.e. the start of the PAGTAB chain for that job. The UPMP for each job with core is not part of the job's PAGTAB linked list but rather a separate linked list of only one entry pointed to by JBTUPM. PAGPTR points to the first page in a linked list of pages that are not being used and hence are available for assignment.

PAGPTR - is a pointer to the start of the free chain in PAGTAB as shown in the PAGTAB drawing.

BIGHOL - is the number of unassigned physical memory pages (the same as the number of links in the linked list pointed to by PAGPTR).

CORTAL - is the number of free pages (BIGHOL) plus the number of idle and dormant high segment pages.

VIRTAL - contains the total number of free pages in the swapping space.

JBTUPM - is a table whose entries (one per job) point to core resident UPMP.

JBTADR - contains the core address and length for each segment in core for each job.

CORE MANAGEMENT ALGORITHMS

Requests for core generally go through four steps before core is assigned: 1. dispatch to core handling routines, 2. preprocessing and argument checking, 3. allocation and 4. assignment.

Core Command

When a CORE monitor command is issued by a user, monitor control passes to the core routine in COMCON, then to COREØ in the module CORE1, where preprocessing in performed. Next, the CORE1 routine is called. When done, CORE1 falls into CORE1A for assignment. Through the use of the core command a user can give up all of the job's core allocation as well as create a core image starting from Ø core.

CORE. UUO

The CORE. monitor call allows the user to allocate and deallocate pages subject to two restrictions. This UUO does not allow all pages to be deallocated and will not allocate from zero pages.

When this monitor call is issued, control passes to the CORE UUO routine in UUOCON, then to CORUUO to check the arguments. Finally, the same routines as executed by the CORE command are called: CORE1, VIRCHK and CORE1A.

Swapper

The swapper (discussed on day 5) goes staight to CORGET to allocate and assign memory. The general flow of core management for these three processes is summarized in the chart CM-2.

CORE ALLOCATION

Core Command

The CORE monitor command accomplishes its preprocessing at CORE0 before going to CORE1 to allocate the core. CORE0 may also be entered from COMCON when minimal core (2K) must be assigned for the KJOB, RUN, ASSIGN and DEASSIGN commands.

At CORE0 (CM-3), 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, IMGIN in JBTSWP is changed to reflect the amount of core desired by the job as specified in the CORE command. This amount will be allocated and assigned at the time the job is swapped in.

The CORE command, unlike the CORE. UUO, may deallocate all of a job's core.

CORE. UUO

Whenever a CORE. UUO is issued, control comes to CORUUO in CORE1 where two cases must be handled. If the UUO has a zero argument, the core size is not changed; instead, the amount of the user's core is returned. If the job size must be altered, CHGCOR is called to allocate and assign core.

CHGCOR is called either from CORUUO or UUOCON (to set up buffer rings). Its first function is to wait for all I/O to finish so that no data is lost. Otherwise, incoming data

may have no place to go (pages deallocated (decreasing case) or job swapped out (increasing case)).

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 the job is increasing in size but couldn't get core, the job is marked for swap out. This is called the expanding case. The job will eventually be swapped in with the correct amount of core.

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

CORE1 Routine

CORE1 is concerned only with allocation and its main function is to insure that certain conditions are met. Control reaches routine CORE1 in module CORE1 from both the CORE. UUO and the CORE command. The conditions that are tested are:

- Job not locked in core
- Job not expanding into high segment
- Job not exceeding virtual limits
- Job not exceeding VIRTAL
- Job not exceeding CORMAX

If all the conditions are met successfully, the pages are allocated and assigned using VIRCHK (except for two cases noted on the flowcharts) and return goes to UUOCON or COMCON.

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

CORE ASSIGNMENT

Once core allocation has been accomplished, in routine CORE1, and sometimes VIRCHK, the actual core assignment must be done. The two routines responsible for core assignment are VIRCHK and CORE1A. VIRCHK will do core assignment for CORE commands and core UUO's except for changes to sharable

high segments and low segments expanding from zero size while routine CORE1A will do the assignment for the other cases of CORE command and UUO usage as well as for the swapper. In order to understand this division of labor, the following five cases must be considered:

1. Case 1 - In the event that core is being deassigned, the monitor will destroy the UPMP if all of the jobs core is being given up. CORE1A handles this.

2. Case 2 - The job is increasing in size and in not going virtual. VIRCHK calls routine PHYCRZ which then causes PAGTAB to be manipulated in the event core is available or the XPN bit to be set otherwise. (VIRCHK)

3. Case 3 - If the job is increasing in size and is virtual or will be going virtual, the S bit is set in the corresponding page map entries. (VIRCHK)

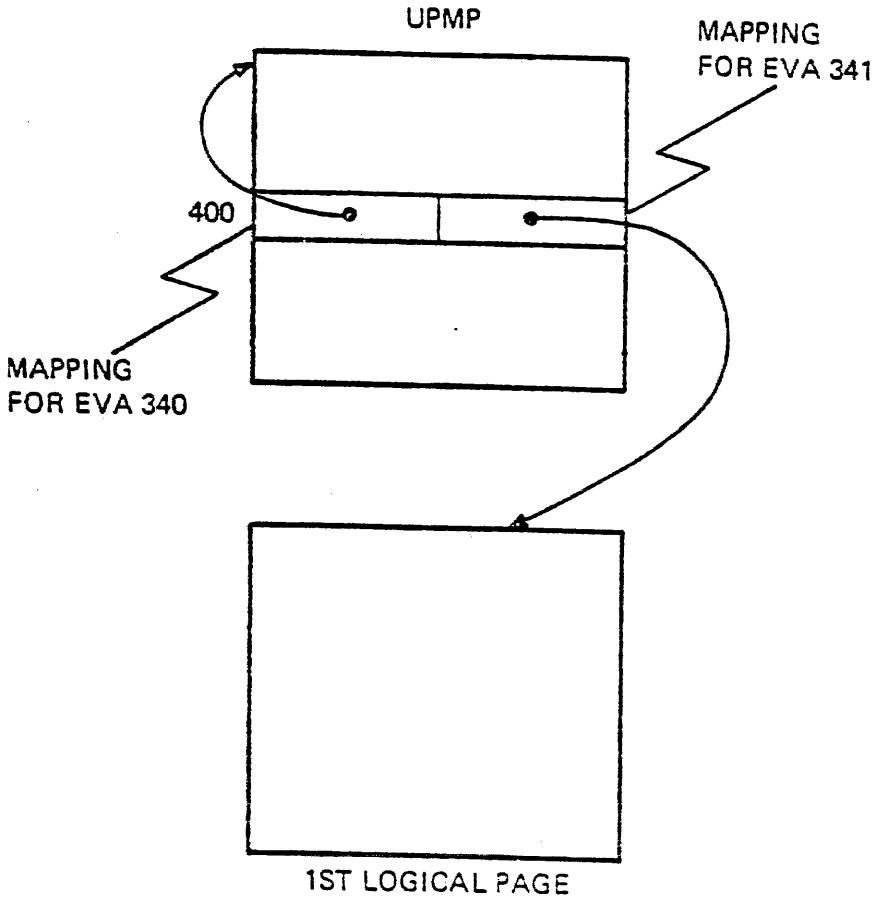
4. Case 4 - Jobs starting with Ø are and requesting an amount not available will cause control to pass to routine CORGT7 so that they may be marked as expanding. (CORE1A)

5. Case 5 - The monitor must perform two steps in this case: build the UPMP and get the core. Once CORE1A is entered, dispatch is made to CORGTØ. There, the pages are assigned in the low segment including one for UPMP and all pages are zeroed. Then the UPMP is built (refer to later in this chapter for more information on the steps necessary to complete this task). Updating JBTADR, JBTREL and the UPMP completes the operation.

Creating a UPMP

When a user job expands from zero core, the job's UPMP must be created. Even though a page has been assigned for the UPMP, it cannot be accessed since all exec mapping for EVA 340 (the UPMP) must go through the still non-existant UPMP. In other words, an alternative mapping scheme must be used to access the UPMP until is has been completely initialized.

The UPMP, besides pointing to all the user pages, must point to itself so that it can be accessed. This is done through location UPMP+400:



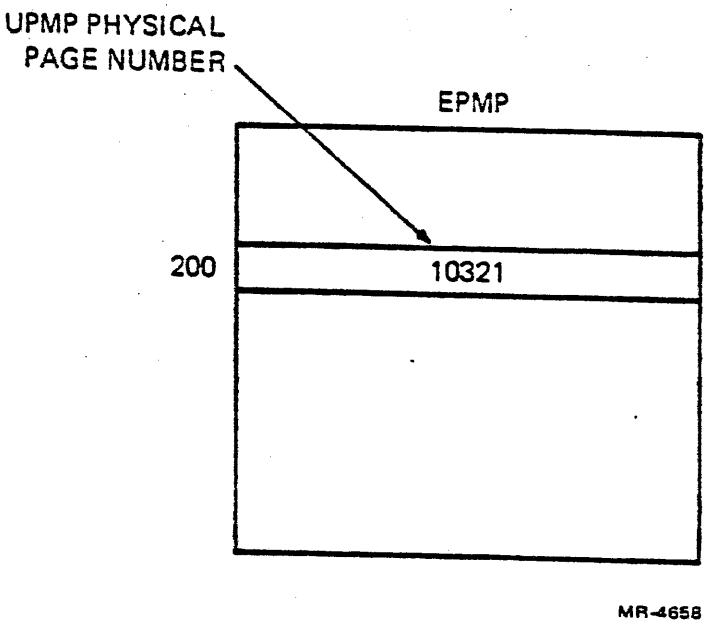
MR-4657

The right half of location 400 (EVA 341) will point to the first logical page of the user's program and the left half of location 400 (EVA 340) will point to the UPMP itself.

Suppose a page has been assigned to create a UPMP and its 13-bit physical page number is known. How can the page be accessed to create the necessary links? EVA 340 cannot be used since the per process area is not set up.

The solution lies in the use of the EPMP. The EPMP always exists and is always pointed to by the EBR register. If the physical page number is placed in a spare mapping slot in the Executive Map Page, it can be accessed through the EVA.

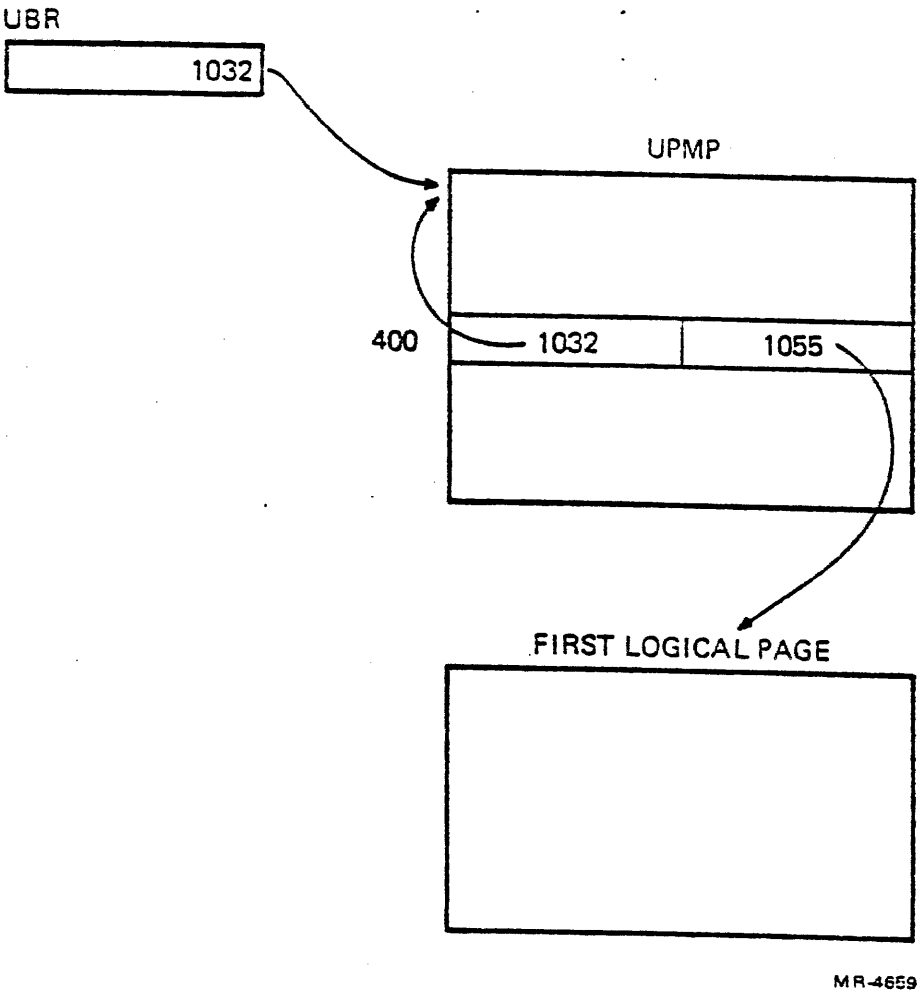
For example, suppose page 1032 has been allocated as the UPMP. The number 1032 is stored in the EPMP mapping location corresponding to EVA 400.



and in JBTUPM. (JBTUPM contains the address of the UPMP for each job.) Executive virtual address 400000 now maps through 1032000 to the UPMP. The UPMP link to itself is accomplished by this instruction:

```
MOVEM   AC,400400       ;AC holds Phy. Page #
```

Since the UPMP can now stand by itself, 1032 is placed in the UBR making it addressable at executive address 340000. Then, the remainder of initialization can be performed.



VIRTUAL MEMORY

Virtual memory was first implemented in TOPS-10 with the release of 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. The overall handling is resolved via a combination of hardware and software.

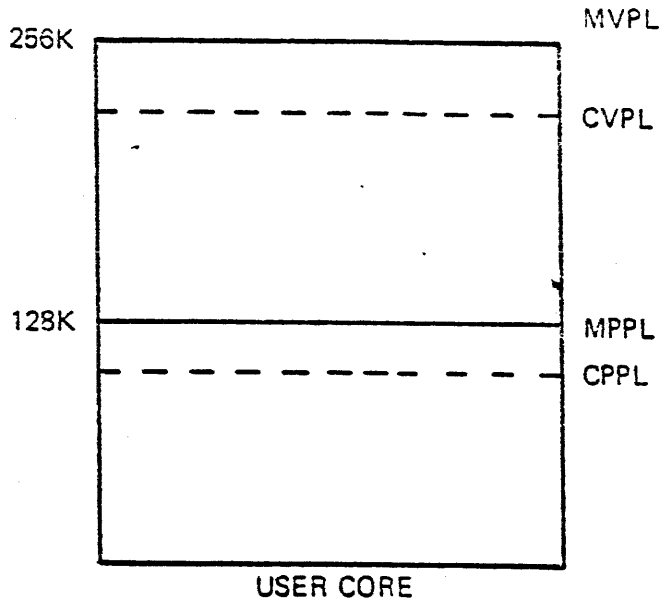
References to addresses in pages with the A bit cleared produce a page fault. The page fault condition, in most cases, traps to a routine called a page fault handler (PFH). The PFH will decide which pages must be brought into core so that the job can continue to run. The system default PFH is kept in SYS:PFH.EXE (SYS:PFH.VMX pre 7.01) but users may write and use their own. In all cases, the user bears the burden of overhead for using virtual memory. The page fault handler resides in user space and is considered part of his program.

There are two sets of controls for virtual memory; one for administrators and one for the user. The administrator may specify for each PPN the:

1. Maximum Physical Page Limit (MPPL) - The maximum number of pages a user job may have in core at any one time.

2. Maximum virtual Page Limit (MVPL) - The maximum size that a program may reach including memory and what is stored on disk.

For example, take a user whose MPPL is 128K and MVPL is 256K.



As long as the program size is less than the MPPL (128K), the program is not "virtual". When the program is bigger than 128K, the program is virtual and not all pages may be in memory at the same time. For users without VM, the MPPL and MVPL are identical.

These parameters are set by the administrator in ACCT.SYS and stored in the Process Data Block when a job is running.

The user also has limits he/she can set for a specific job during program execution.

1. Current Physical Page Limit (CPPL) - the maximum number of physical pages a user may have. This number is always less than or equal to the MPPL and is modified by using the "SET PHYSICAL LIMIT" monitor command.

2. Current Virtual Page Limit (CVPL) - the maximum program size a user may have. This number is always less than or equal to the MVPL and is modified by using the "SET VIRTUAL LIMIT" monitor command.

In lieu of a CPPL, the user may set a physical guideline. The guideline may be exceeded but the job's size will be brought down to the guideline each virtual time trap. (A virtual time trap is defined later).

Given either of these limits, what event will cause a job to run virtual? There are three causes: a RUN or GET command, a CORE. UUO or a PAGE UUO.

Virtual Memory Data Base

When a job is run using virtual memory, there is much more accounting that must be done to keep track of the pages. Because the information is unique to the job, the data base for virtual memory is kept in the UPMP.

AABTAB - is a subtable of bits within the UPMP starting at word 457. There is one bit per page which reflects whether the pages exists or not.

WSBTAB - is a subtable of bits within the UPMP starting at 440. This is the working set table and contains one bit per page. If a bit is 1, the corresponding page is in core.

The UPMP entries for each page are composed of 18 bits. For more information, see the drawing in the supplement that describes UPMP entries.

PAGE MONITOR CALL

The PAGE monitor call allows a user to manipulate pages and the data contained in them. PFHs use the PAGE. UUO for all page manipulation. The calling sequence is:

```
        MOVE    ac,[XWD ftn,,addr]
        PAGE.   ac,
          error return
        normal return

addr:   number of words
        argument 1
            .
            .
        argument n
```

The functions are:

0	Swap a page in or out
1	Create or destroy a page
2	Move or exchange a page
3	Set or clear the access-allowed bit
4	Return WSBTAB
5	Return AABTAB
6	Return the status of a page o Does not exist o Writable o Readable o Accessible o High segment o Shareable o Can't be paged o ABZ
7	Create a high segment from a collection of pages
10	Set or clear the cache bit

PAGE FAULT HANDLERS

Page Fault Handlers (PFHs) manage memory for individual jobs when a job's program exceeds its current physical page limit. It 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 decision as to the specific pages to be swapped in and out is one of the the major

functions of a page fault handler. A list of pages must be maintained according to some algorithm. In addition, page fault handlers may create pages (allocated but zero).

Until a program exceeds the CPPL, a page fault handler 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 gets SYS:PFH.EXE and stores it in page 777 of user space. From that point on, the PFH will manage user core for the job.

Control is sent to the PFH for some types of page failures, a time trap or a potential page failure (UUO call).

Page Failure

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

- o Proprietary Violation - a concealed page has been referenced
- o Page Refill Failure - hardware problem
- o Address Failure - address break
- o Page Table Parity Error
- o AR Parity Error
- o ARX Parity Error
- o Reference to a location whose UPMP entry has A=0

Only this last case will actually be handled by the PFH; the rest are handled by the monitor.

For a KI10, page fault traps go through location 420 in the UPMP. KL10 page faults trap through locations 501 and 502. When the trap occurs, on a KL, the page fail word is stored in location 500 of the UPMP, the current PC is stored in 501 and a new PC is retieved from 502.

The new PC in the page fail word is the address of the SEILM routine in COMMON which will sort out the type of condition and dispatch to the correct location. If the normal condition is met (a page needing to be read in), control passes to USRFLT.

The USRFLT routine in VMSER never returns to SEILM if the user is running virtual. If the A bit is off and the page is in the working set, the A bit is turned on and control is returned to the user program. The other cases (A equal to zero and page not in the working set, time trap, or UUO check) will cause the PFH argument block to be filled and control to be transferred to the user PFH.

The format of the PFH arugment block is shown in the supplement.

Potential Page Failure

This case handles the special situation where a monitor call has been issued and a test must be made to see if the argument block is in core. The UUOCHK routine in UUOCON will perform this test by entering the USRFLT routine at USRFL1 which eventually runs the PFH if necessary to get the page in core. See the flowchart of the USRFLT routine on the previous page.

Time Trap

Time traps are used to maintain the list of pages due for replacement and are initiated by the monitor in the clock cycle. A counter in the UPMP is decremented for the current job every virtual clock tick (runtime). When the counter reaches zero, control passes to USRFL1 and on to the PFH. Time traps also allow the PFH to bring core usage down from the current usage to the guideline by paging pages out.

Page Fault Handler

Finally, the PFH is reached and the decision to page pages can be made. The default PFH (SYS:PFH.EXE) is organized as a second chance first-in first-out (FIFO) algorithm. This means that the first page paged in will be the first page paged out unless there are pages that have not been accessed (the second chance).

Four conditions must be actually handled by the PFH:

1. Page Not In Memory - The decision at this point is made by examining 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 just paged in.

2. Page Allocated But Zero - This case also depends on 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 swapped out and the new page created.

3. "A" Bit Off - Simply turn the bit on.

4. Time Interrupt - The decision at this point depends on the guidelines. If the guideline has not been exceeded, only the virtual time trap counter must be reset to its initial value (1/2 second). If the guideline is exceeded, enough pages with "A" off and "W" on are paged out to bring the number of pages down to the limit, the FIFO list is rebuilt and time trap counter is reset.

This page is for notes

EXERCISES

*1. For what reasons will the CORE1 routine take the error (nonskip) return?

*2. When would a segment be marked as expanding?

In executing the CORE UUO, why isn't there a need to allocate an extra page for the UPMP as the swapper does?

*4. What are the mechanics involved in creating a UPMP?

5. Why is it necessary to clear the hardware page table in the process of creating the UPMP? To set up the User Base Address Register?

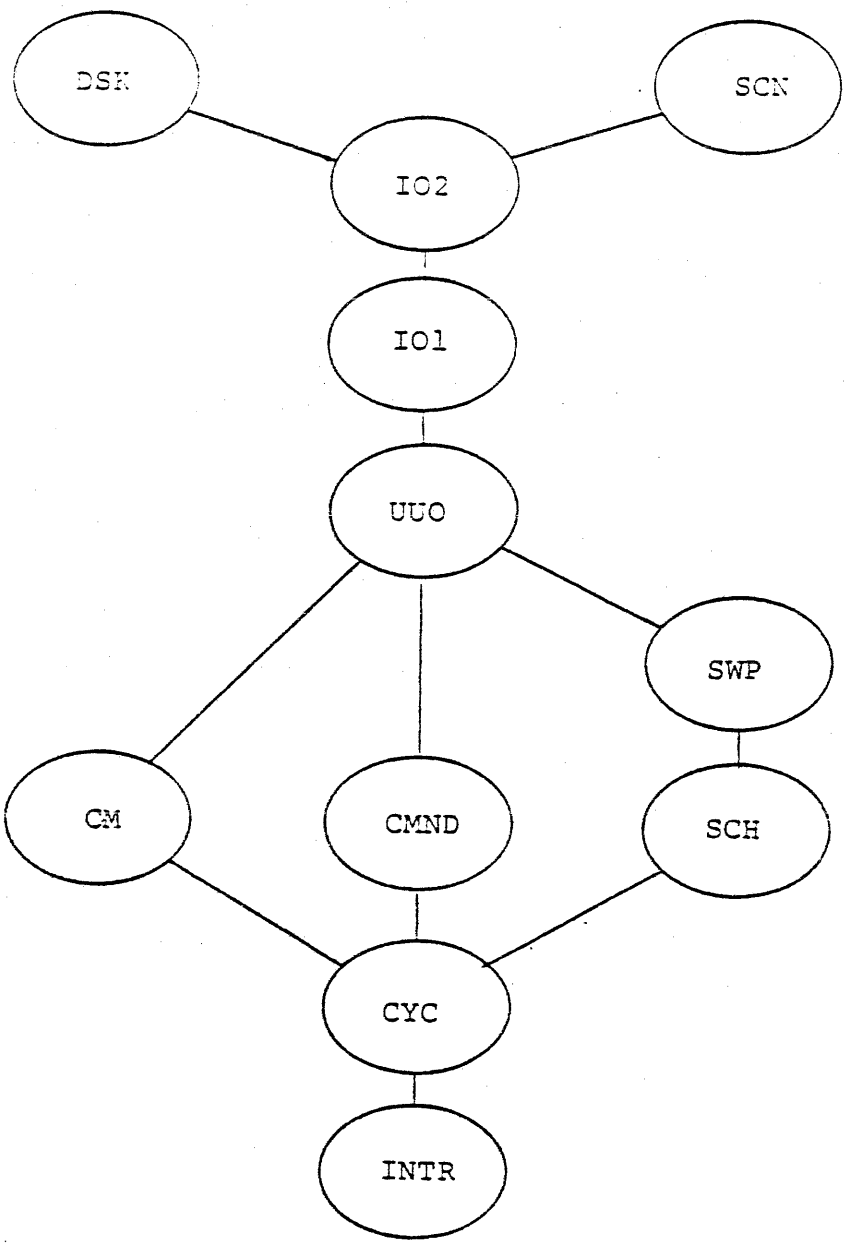
6. When might the size of a swapped out job be changed?

7. When might a zero length job be marked as needing to be swapped out (JXPN set)?

8. When would a job's physical core assignment b echanged without its's allocation being changed?

TOPS-10 MONITOR INTERNALS

The Command Processor



COURSE MAP

This page is for notes

THE COMMAND PROCESSOR

INTRODUCTION

The command processor (COMCON) is called once a tick from the monitor cycle to decipher commands typed at monitor level from a job. COMCON will decide upon the appropriate function and either run a program or handle the request itself. This module will explain how COMCON preprocesses commands and dispatches to various routines, how error checking is done and how a user may add new commands.

RESOURCES

TOPS-10 Microfiche

TOPS-10 Monitor Internals Course Supplement

Course Materials - Chapter 4
Supplement - Graphics 4-All

LECTURE OUTLINE

I. Overview Including Functionality

II. Where Commands Come From and How Do We Know We Have Them
   A. SCNSER Interrupt Level Activity
   B. Forced vs Regular Commands
   C. CMDMAP

III. Data Base
   A. Predispatch Bits
   B. Post Dispatch Bits

IV. Monitor Cycle Entry
   A. CMDMAP
   B. COMDEV
   C. COMCON

V. Command Identification
   A. Command Extraction
   B. Command Validation and Setup

VI. Predispatch
   A. Errors
   B. Delayed

VII. Dispatch
   A. Short
   B. Long

VIII. Post Dispatch

This page is for notes

THE COMMAND PROCESSOR

The Command Processor, COMCON, provides the means for users to request the services of the Monitor. Each time a command is typed on a user terminal, the Command Processor interprets the command and calls a routine to handle it. In this chapter we shall be concerned with the process of interpreting the command and dispatching to the routine to handle it. We shall also look at some important housekeeping functions performed before and after dispatching to the command routine. We shall not be concerned with the functions of the various commands at this time, nor shall we worry about most of the possible errors that might be detected. Several of the most important commands will be considered in detail in later chapters.

As the user types a command, there is an interrupt on each character. The terminal interrupt routine reads in each character and stores it in an input buffer for that terminal. The interrupt routine tests each character to determine if it is a break character indicating 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 is set indicating that there is a command from that line waiting to be processed. This bit is in a table called CMDMAP.

On the next clock interrupt, the overall control routine will check CMDMAP, and finding any bit set dispatch to the Command Processor. The Command Processor will call the routine to handle one line's command, and then return to the control routine.

The Command Processor will not process more than one command on each call. If several commands happen to be completed by users during the same monitor cycle, the Command Processor will be called on successive clock ticks until all the commands have been processed. This policy ensures that too much of any one cycle is not spent within the Command Processor.

Terminal Considerations

There is an input buffer and output buffer within the monitor for each terminal line. All terminal I/O is to and from these buffers. A data block, called the Line Data Block (LDB), contains the buffer pointers and a great deal of additional information about each Teletype 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 will be 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.

In 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 care reader.

The Dispatch Process

When the Command Processor has identified the line having a command, the contents of the first word is extracted from that line's input buffer as the command. Any additional words in the input buffer may be taken as arguments by the routine which handles the specific command, but will not be considered in the dispatch process.

The table COMTAB contains a list of all valid commands, in SIXBIT format. The table DISP has an entry, corresponding to each entry in COMTAB, which gives the address of the routine to process that commmand. The DISP entry also has a number of bits which specify functions to be performed before and after calling the command routine.

The Dispatch Routine gets the first six characters from the terminal input buffer, converts them to SIXBIT, and performs a table lookup on COMTAB. If it finds an entry which exactly matches the command given by the user, that entry identifies the command. If no entry matches exactly, it checks if one and only one entry matches for as many characters as the user typed (i.e. if the user typed an 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 can be identified, the address of the routine to process it is picked up from DISP.

Long Routines

Since the Command Processor operates as a part of the overall Monitor Cycle, and the time spent in it reduces the time spent in the next user program, the command routine must be written to run to completion quickly. Many commands, however, require more time than we can afford to take out of the overall cycle. The way we handle these commands is to set up a Monitor routine to run as the user's job. All the command routine does then is set up the job, and the lengthy processing is then done in the Monitor routine running in the user's time. Such a routine appears very similar to a UUO, except that it will not return to the user program upon completion. Since the processing of these commands uses the user's accumulators and PC, they will not be accepted while the user has a program running.

Many commands which may 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 then giving the appropriate commands to the program. In the command dispatch process these commands are all equivalent to R.

Forced Commands

Certain Monitor routines sometimes need to "cause" a command to e executed for a given line. For example, if a user at a dataset hangs up (or the telephone circuit is broken) we want to DETACH the job. The forced command mechanism allows us to do this. A monitor routine which

wants to force a command to be performed for some line can deposit a "forced command index" into the line's LDB and set the forced command bit, LDBCMF. The Command Processor will not look at input buffer on a forced command. Instead, it will use the forced command index as a pointer into a table of forced commands, TTFCOM. The SIXBIT command from TTFCOM is then processed exactly as a normal command would have been.

Macro Flow

The macro level flow chart of the Command Processor is given in the supplement. The first action taken is to determine the line having a command to process. We do this by checking the CMDMAP bits in COMDEV. Each time a line is chosen, we save the line number in LINSAV and start scanning at the next line on the next call. This guarantees that every line has the same chance of getting its commands processed. Before processing a second command for any line we give every other line a chance.

Once the terminal line has been chosen the LDBCMP bit is checked to determine if a forced command is pending; if so, a sixbit command is extracted from TTFCOM otherwise up to six characters are extracted from the terminal buffer, skipping any leading blanks, and converted to sixbit.

Next we look up the command in COMTAB. If we find the command, we pick up its entry from DISP and do the legality checking. Upon successful completion of the legality checking, we dispatch to the routine for this command.

Upon return from the command routine we do some necessary housekeeping to account for the fact that the command has been processed. Additional housekeeping is necessary if a job was initialized as a result of this command. If appropriate, according to bits in DISP, we type a carriage return line feed and a period. If the command causes a program to run as the user job, we set bits to tell the Scheduler that the job is runnable. Finally if the job was put into the Command Wait Queue, and will not be requeued as a result of the normal action of the command, we mark it be requeued back to its former queue. We then exit to the Control Routine in CLOCK1.

Predispatch Bits

A number of bits in DISP specify conditions to be checked and functions to be performed before dispatching to the command routine.

NOLOGIN specifies whether the job must be logged in before the command is legal. Most commands have this bit set to zero. One obvious exception is LOGIN.

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 will be assigned to the terminal.

NORUN specifies whether the command may be performed for a job which has a program running. All commands which result in setting up a routine to run as the user job will have this bit set. If such a command is typed while the job has a program running, there will be an error message, "Type ^C first."

INCOR indicates that the job must be in physical core if it has any core allocated. This bit does not make the command illegal for a job which has no core allocated. If a command with this bit is given while the job is swapped out (or being swapped) the command must be delayed, and the job will be put into the Command Wait Queue. In the Command Wait Queue it will have very high priority to be swapped in.

The NOACT bit can cause the command to be delayed similar to the INCOR bit. In this case however the job is already in core, because there is active I/O, however we would like the scheduler to ignore running the job, to insure that more I/O will not be started. To accomplish this goal the CMWB is set and the job is eventually requeued to the CMWQ. Keep in mind the use of the CMWB and CMWQ is two fold; it says swap the job in if it is swapped out and don't run me.

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.

NXONLY indicate that the command is illegal for a job running an execute only program. "Execute only" is a specification that may be given to a disk file to allow

users to run it as a program, but not look at it. The purpose of this specificaton is to allow proprietary programs to be executed by users which do not have the privilege of looking at them.

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.

Postdispatch Bits

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

NOINCK indicates that a job will not be initialized as a result of this command. It is set initially in DISP for certain commands. It is set in the accumulator in which the DISP bits are held in case of an error in a command which otherwise would have initialized a job.

NOCRLF and NOPER specify whether a carriage return, line feed and a period should be typed upon completion of a command. NOPER will be set for any command on which we are not ready to accept another command immediately after completion of the command routine. This includes all commands which set up a monitor routine to run as the user job.

There are three bits which say to set the job runnable under different conditions. TTYRNU says to make the job runnable, and switch the terminal to user level. The RUN command is an example which has this bit set. TTYRNC says set the job runnable, but leave the terminal at command level. Both these bits will cause the job to be put into a RUN queue by the Scheduler. TTYRNW says to make the job runnable, but check if it was stopped in terminal I/O Wait. If so, it will be put back into the terminal I/O wait queue. CONT has this bit set. Not more than one of these three bits will be set for any command. None of them will be set for a command which does not make the job runnable.

Command routines which output to the user's terminal simply deposit characters in the output buffer. They do not start the terminal. One of the functions performed upon return from the command routine is to start the Teletype, in case there are characters to be typed. The NOMESS bit suppresses this action for commands which never type a message.

The Command Wait Requeue Bit, CMWRQ, has the functon of getting the job out of the Command Wait Queue. The bit is set in DISP for any command which might cause the job to be put into the Command Wait Queue and which does not cause the job to be requeued. The bit is cleared in the accumulator if the job did not actually have to be requeued to Command

Wait. Upon completion of the command routine, if this bit is set in the accumulator, we mark the job to be put back into its former queue.

The CUSTMR bit is reserved for installations to use for their own purposes, and has no function in the standard Monitor.

EXERCISES

*1. When is the Command Ready bit set in CMDMAP?

*2. Where is the Command Ready bit cleared?

3. What determines whether the line will be at command level or user level upon completion of the command routine?

4. Suppose a user types a line while his program is still running. What determines whether this line will be taken as a command or as input for the program.

5. What determines how many characters a user must type in order to give a valid abbreviation of a command?

6. Why is R not taken as an abbreviation for RUN?

7. What happens if a forced command is set up for a line which already has a command waiting to be processed.?

*8. When is a user assigned a job number by the Monitor?

9. For what reasons might a command have to be delayed?

*10. Why would a job be put into the Command Wait Queue?

*11. If a job is put into the Command Wait Queue how does it get out and where does it go?

12. How do we avoid taking too much of any one Monitor cycle for command processing?

13. After which commands does the Monitor not type a period? Why?

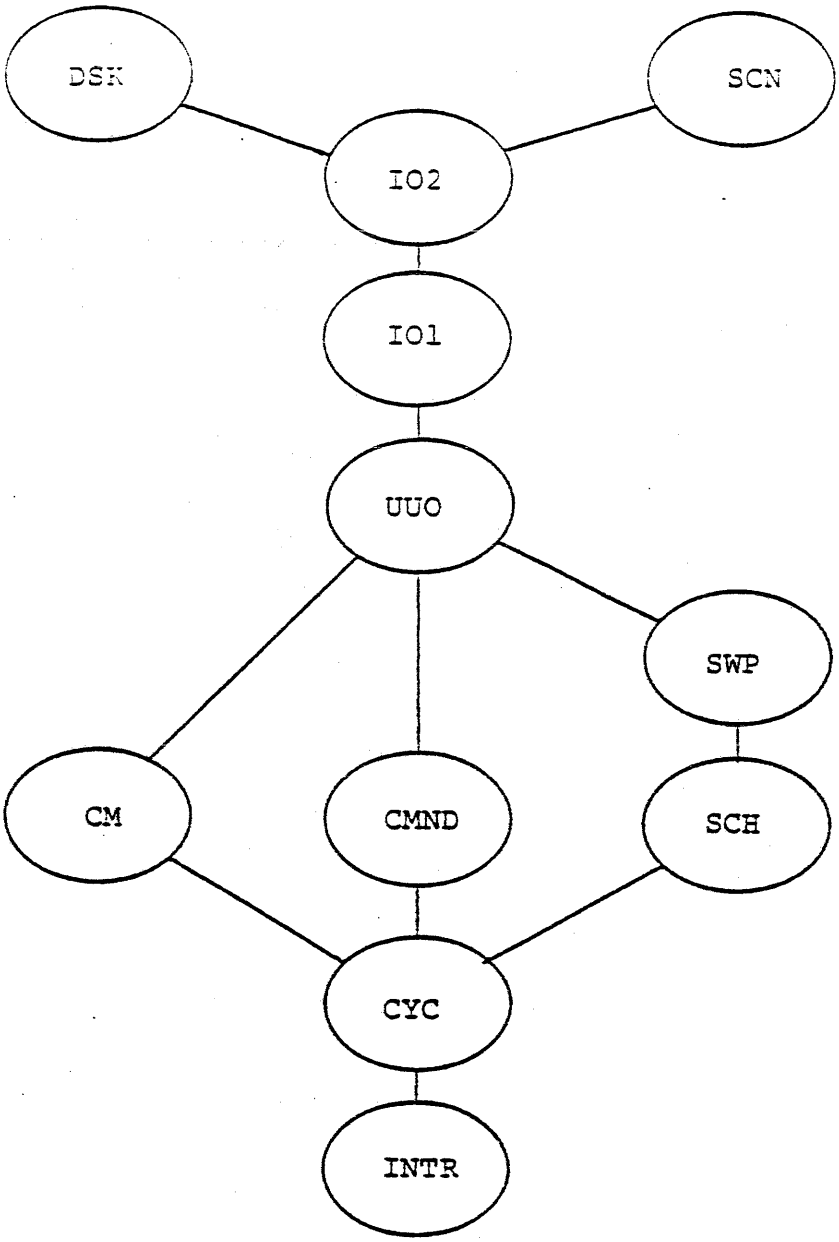
14. What does the Monitor do in response to an invalid command?

15. Why can some commands be performed while the user has a program running while others can not?

This page is for notes

TOPS-10 MONITOR INTERNALS

The Scheduler



COURSE MAP

This page is for notes

THE SCHEDULER

INTRODUCTION

The last of the monitor cycle components is the scheduler which selects and runs a job based upon the need for scarce system resources. This module will explain how jobs are placed in different queues and how they are selected to run based on scheduling parameters.

RESOURCES

TOPS-10 Microfiche

TOPS-10 Monitor Internals Course Supplement

Course Materials - Chapter 5
Supplement - Graphics Section 5-All
- 603A Scheduler/Swapper PLM
  Read Chapters 1,2,5
  Skim Chapters 3,6
- 603A Scheduler/Swapper Flow Charts

LECTURE OUTLINE

I. Overview

II. Data Base
  A. JBTSTS
  B. JBT?Q

III. Job States
  A. Macro Defintions
  B. Long vs Short
  C. Sharable Resource Management

IV. Determining the State of a Job

V. Job Requeuing Overview

VI. Job Requeuing Details
  A. QXFER
  B. Special Cases
  C. Requeue Flow Charts

VII. Queue Scanning
  A. Scan Tables
  B. QSCAN
  C. Examples

VIII. Scheduler Flowcharts

IX. Scheduler Performance

A. Parameters
B. Defaults and Effects
    1. ICPT
    2. QRT
    3. Fairness
    4. Response

This page is for notes

The Scheduler

The Scheduler controls the allocation of system resources among user jobs. Specifically, it selects the job to run next, performs all requeuing, controls the allocation of sharable resources and governs the priority of jobs to be in core memory. In this chapter we look at the overall philosophy of scheduling in the DECsystem-10 and at the specific procedures by which this philosophy is implemented.

Queues

Scheduling in the DECsystem-10 is based on the use of queues and wait state codes. The queue which a job is in, and its position within the queue, determine its 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 will not be able to use the CPU until the resource is available. Jobs which are most likely to be able to use the CPU are in processor queues, called PQ1 and PQ2. All jobs in PQ1 are given high response but for a very short time. Jobs will enter into the back of PQ1 when they come out of any long term wait such as terminal I/O wait, command wait, DAEMON wait, or just starting to run. Certain jobs that have been sleeping or hibernating will also be queued into PQ1 if they were sleeping for more than one second. PQ2 is divided into an arbitrary number of ordered classes (0-32), the number of classes is to be determined at "MONGEN" time. These subclasses of PQ2 are scanned for job selection according to their assigned quotas of CPU time. Jobs will enter PQ2 from PQ1 when they have exceeded the short amount of time that they are allowed in PQ1. In some systems there are also high priority processor queues, HPQ1, HPQ2, etc. Users must have special privileges to get their jobs into these queues. They are normally not used except for real time applications.

Wait state codes are maintained for each job in the JBTSTS table and are useful in defining the notions of short and long term states. Short term states apply to wait satisfied, shareable resource wait, IO wait and short sleep. There are no queues associated with short term states. Jobs in short-term wait states will maintain their position in a run queue but have their wait state code altered. Long term states refer to the processor, command wait, job waiting for DAEMON, teletype IO wait, sleep, event wait, stop and null

queues. Jobs going into long term states enter at the rear of the respective queue (never at the front).

Sharable Resources

There are a number of sharable resource wait states. A sharable resource is some part of the system, either hardware or software, which 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. It must be shared by all jobs doing IO on units which it controls. Only one of these jobs may have IO 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 will be of the form XX'REQ. XX represents a two letter mnemonic for the resource. Each REQTAB entry is initialized to -1. The code which uses the resource begins with an instruction which increments and tests the appropriate entry. If the value is now 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 will become unblocked when it is being considered to run by the scheduler and the resource is available. For some resources such as monitor buffers or executive virtual memory the REQTAB entry will be initialized to -n where n is the number of monitor buffers or executive virtual memory slots.

Another table, AVALTB, contains entries parallel to the REQTAB entries, which are used as flags to the Scheduler that the corresponding resources are available. The flag in AVALTB is set to a nonzero value at the end of the code which uses the sharable resource. It is set, however, only if there is a job waiting for the resource.

Sharable resource management is accomplished by two routines in CLOCK1, XX'WAIT for resource allocation and XX'FREE for resource deallocation (where XX is the two letter mnemonic for the resource). These routines are

called from various modules within the monitor that use the resources.

The code within CLOCK1 for resource allocation does the following:

```
XX'WAIT:
        housekeeping

        AOSG    XX'REQ      ;is resource available
        JRST    SRAVAL      ;yes, go use it
    -go return EVM          ;no, block job
     if necessary
    -start partial cycle via entry to WSCHD1

SCAVAL: housekeeping        ;have resource
        return to calling module
```

The code within CLOCK for resource deallocation does the following:

```
XX'FREE:    SOSL    XX'REQ
            SETOM   XX'AVAL

            return to calling module
```

NOTE that REQTAB entries are incremented and tested with a single instruction. If the resource is given up at interrupt level, we do not have to worry about the interrupt occuring between incrementing and testing.

NOTE also that in the case of magtape usage a job will be 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 will be executing any part of it. Since we do not allow rescheduling due to clock ticks during a UUO, we normally do not ave to worry about another job starting 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 the routine. We expect any job which has been given a sharable resource to quickly go into I/O wait.

Requirements for sharable resources may be nested. A job which owns one resource may have to queue for another. This leads to the possibility of deadlock situation wherein two jobs are each waiting for the resource which the other owns. Neither job will run, and therefore neither will release the resource which it owns. We overcome this problem by a programming convention that any time there are nested requirements for sharable resource they must be nested in the same order. The order convention also prevents more complicated deadlocks involving several jobs.

Core memory is a "sharable" resource for which there is no one queue or wait state. Jobs in any processor queue may be either 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 while 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 which are most likely to do productive work (i.e. 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.

CPU SCHEDULING

The scheduling of CPU time is based primarily on the order indicated by the scheduler scan table, SSCAN. SSCAN indicates that the in-core jobs in HPQ's should be scanned first followed by in-core PQ1 jobs, in-core PQ2 jobs by subclasses and finally in-core background batch jobs. If no runnable jobs can be found the null job will run. There are however, three additional factors influencing the normal selection of jobs. These are the quantum run time, in core protect time (ICPT) and subclass quotas. The 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. Once it expires the job is requeued to the bottom of PQ2. The ICPT in core protect time provides a mechanism wherein the swapper is prevented from immediately swapping out a job which has been just swapped in. Additionally it ensures that a job gets its fair share of the CPU when it is in core. A job's ICPT value will be decremented if it is in the EWQ or SLPQ, or was scanned to run but rejected (e.g. because it was waiting for a sharable resource). Upon expiration of ICPT a job will be requeued to the back of PQZ.

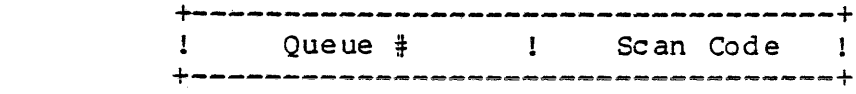
The subclass quota is a percentage of a scheduling interval which is "allocated" to a subclass. During each scheduling interval the scheduler will consider the highest class. If no runnable jobs are found in this class, the next class in the ordering is scanned.

Some Details

The following paragraphs describe some details on the mechanics of queue scanning and queue transfers. They are not essential parts of the scheduler philosophy but they do illustrate the table-driven nature of the scheduler and an understanding of these sections will make it easier to understand the detailed flow charts that follow.

QUEUE SCANNING. There is a generalized routine, called QSCAN, which is used whenever we want to scan through one or more of the job queues. To use QSCAN, the caller must supply a scan table, which specifies which queues are to be scanned and the direction in which each is to be scanned.

A scan table consists of an arbitrary number of words, each with the following format:



Queue # is the negative queue number, and is always written as a label. (e.g. - PQ2)

The scan code has 18 possible values, the following are some of the meanings:

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 forware according to SOSCAN

A zero word terminates the table. (See Table 3-5 in the scheduler PLM, Monitor Internals Supplement for complete list)

To use QSCAN, a routine puts the scan table address into an accumulator, and does a JSP to QSCAN. QSCAN will give a skip return, with the first job number in an accumulator. QSCAN also supplies the address to which the caller may return to get the next job number. Each time the caller returns to that address (with JRST), QSCAN will return to the second word beyond the original call supplying the next job number. QSCAN will automatically step from entry to entry in the scan table, and will give a no skip return if the table is exhausted.

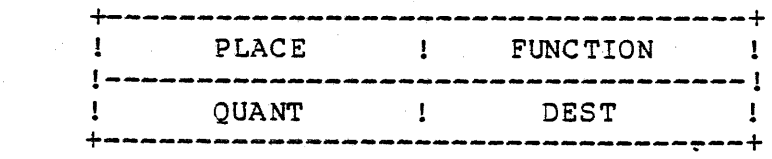
QSCAN is used by the Scheduler in choosing the job to run next. It is also used by the Swapper in selecting jobs to swap in and out.

QUEUE Transfers

All transfers of jobs from one queue to another are performed by the routine QXFER in SCHED1. Transfers may be either to the beginning or to the end of a specified destination queue. The destination queue may 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. And 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.

In requesting a queue transfer, 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 requested. A Transfer Table consists of two words, in the following format:



PLACE will be negative for a transfer to the end of a queue, for a transfer to the geginning of a queue it will be zero.

FUNCTION will have one of three values, QFIX, QLINK or QJSIZ, corresponding to the manner in which the destination queue is to be determined. These are actually labels for the entry points of the routines whthin QXFER which determine the destination queue. However, they might as well be thought of as codes specifying the type of transfer.

On fixed destination transfers, QUANT and DEST are the actual value of the new quantum run time and destination queue number. A negative value of QUANT indicates that the quantum run time is not to be changed. DEST will always be negative.

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

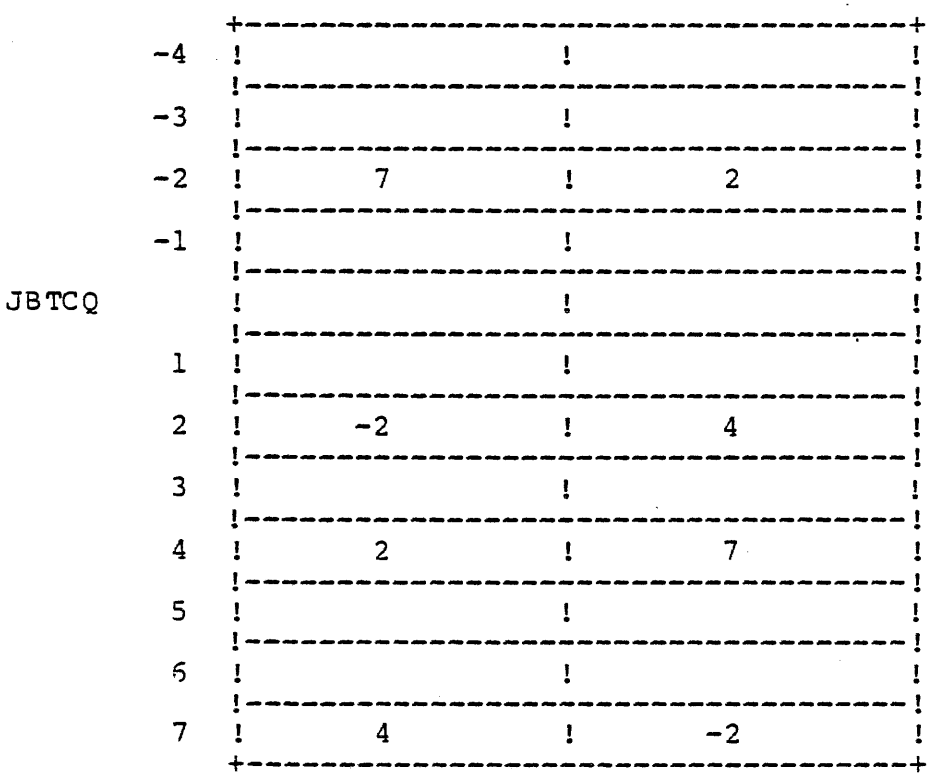
Let us examine actions taken by the Queue Transfer routine on a fixed destination transfer. The calling routine sets up the job number in AC J and the address of a Transfer Table in AC U, and does a PUSHJ to QXFER.

At QXFER AC R 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 will be to QFIX.

At QFIX we check if the job is being requeued to a Run Queue and has successfully requested a High Priority Queue (HPQ). If so, the priority level is obtained from table JBTRTD. The corresponding HPQ number and quantum run time are obtained from tables QTTAB and QQSTAB.

Next, at QFIXI, 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. Note that this procedure works correctly when the job is first or last in its queue, or is the only job in its queue.

Example: Deletion of Job 4 from its queue



Next the job will be inserted into the destination queue. The job will be inserted following either the first link (i.e., the queue header) or the last link - depending on the value of the "PLACE" entry in the Transfer Table.

AC J points to the entry which is to be inserted. AC R will be set to point to the entry which this entry will follow.

AC R 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 R is backed up one entry, to point at the last entry. This is done with the instruction:

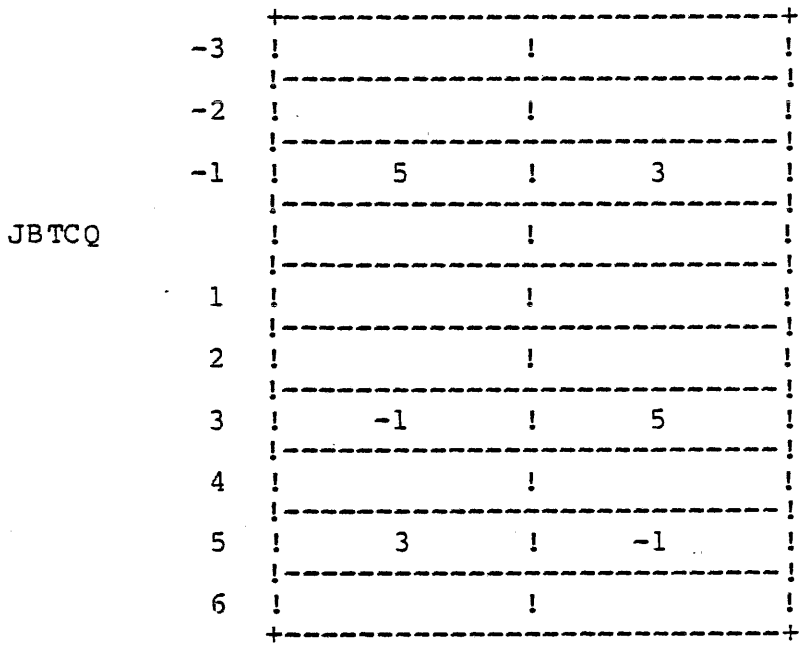
```
HLR R, JBTCQ(R)
```

AC T1 is loaded with the index of the entry in front of which the insertion will be made. This is the value in the RH of the entry to which AC R now points.

Now the new linkages are set up with four easy instructions:

```
HRRM    J,JBTCQ (R)     New "preceding" entry gets (J)
                        as following entry
HRLM    J,JBTCQ (T1)    New "Following" entry gets (J)
                        as preceding entry
HRRM    T1,JBTCQ (J)    This job's entry gets (T1) as
                        following entry
HRLM    R,JBTCQ (J)     This job's entry gets (R) as
                        preceding entry
```

Example: Insert Job 1 at end of queue 1:



R initially contains -1, the index of the queue into which the insertion is to be made.

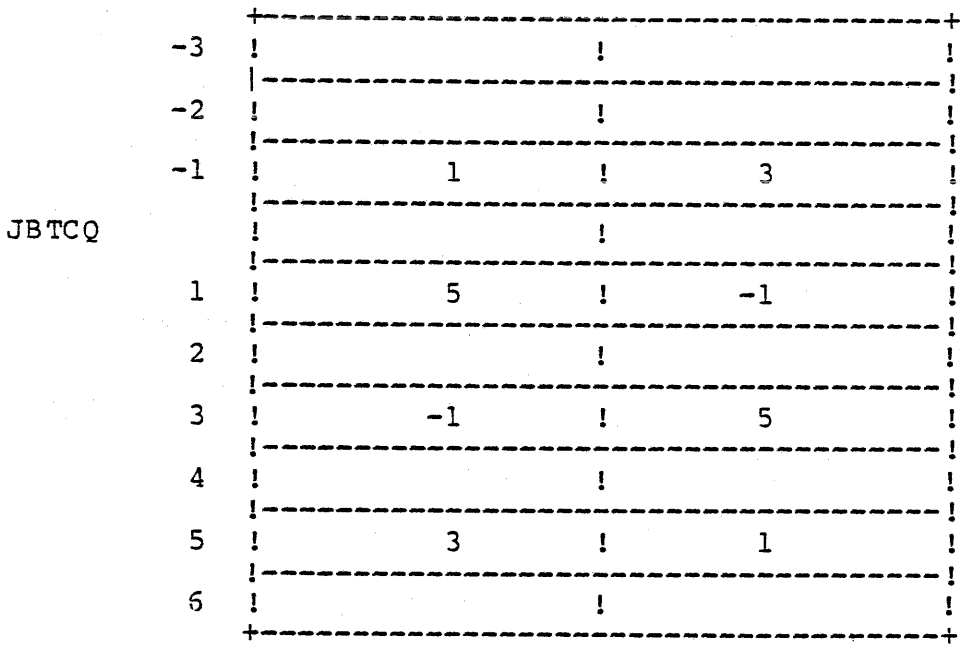
Since the insertion is to be at the end of the queue, AC R is backed up one entry, by loading it from the LH of the entry it points to. Then T1 is loaded with the entry to which R points. The AC's are now set up as follows, all values are relative to JBTCQ:

J points to the entry to be inserted (+1)
R points to the entry after which it will be inserted (+5)
T1 points to the entry before which it will be inserted (-1)

To insert the new job,

(J) is placed into the RH of JBTCQ (R) and LH of JBTCQ (T1)
(T1) RH is placed into the RH of JBTCQ (J),
(R) RH is placed into the LH of JBTCQ (J).

Final result:



After the job has been inserted into its new queue, if QUANT = 0, there is a jump to the routine exit.

Otherwise, QUANT is inserted into the Process Data Block for Job (J) - i.e., the job's quantum run time is reset. Also when QUANT = 0, the Wait State Code in that word is set to 0, we then exit with a POPJ.

On the QLINK and QJSIZ transfers, we must first determine the destination queue and, possibly, quantum run time. This is done by a simple table lookup. The job's current queue number, or its size, is found in the table to which DEST points. The destination queue number is picked up from a corresponding entry in the table. If QUANT is given it is also a table address, and the new quantum run time is picked up from a corresponding entry in that table. Once the destination queue number and quantum run time have been determined, QLINK and QJIZ continue through the same procedure executed for QFIX.

The QFIX transfer is used for all queue transfers except for requeuing done as a result of quantum runtime expiration; in this case the QLINK Function is used. All requeuing is always done to the back of the destination queue. Transfers involving PQ2 will result in an additional transfer in JBTCSQ.

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 will produce 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. It 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. Control flows to the dispsatch address and if necessary routine QXFER is called for an actual queue transfer as described by the transfer table. 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 which the Scheduler checks for individually, and which call for specific transfer tables. For example, if the job's RUN bit is not set, the job will be put into the Stop Queue, regardless of its WSC. If the Command Wait bit is set, it will be 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 after going into the Stop Queue or Command Wait Queue.

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

by the DAEMON is the taking or core dumps of user programs.

DETAILED FLOWS

We are now ready to look at the scheduler in detail. A complete flow chart and program logic manual (PLM) is included in the supplement.

This page is for notes

EXERCISES

*1. Why might a job be unrunnable even though in one of the processor queues? List line numbers to support your answer.

*2. What does the Scheduler do with the job number which it chooses to run next?

3. What is the function of a job's wait state code? Where can the wait state code for a given job be found?

*4. What determines the order in which the scheduler considers jobs for the possibility of running next?

5. What is the meaning of non zero entry in AVALTB?

*6. When is a clock tick considered "lost" rather than simply idle?

7. If there are several jobs waiting for a sharable resource which has become available, which job gets it?

8. When is a job's minimal core utilization cleared by the requeuing routine? Why?

9. When is a job's run bit cleared? Where is the run bit located?

10. Answer the following questions for each of the situations described below.

    i. Describe the events that could cause this situation.
    ii. What transfer table or dispatch address will be used to requeue the job?
    iii. What will the wait state code be after the requeuing?

Question	Current Job	Run bit	JRQ bit	CMWB bit	Wait State Code
a	yes	1	Ø	Ø	IOWQ
b	yes	Ø	Ø	Ø	RNQ
c	no	1	1	Ø	WSQ
d	no	Ø	1	1	RNQ
e	no	1	1	Ø	RNQ

Assume the normal values for all conditions not specified.

In particular, assume that the JDC and JS.DEP bits are always off.

11. Why is the MCU reset every time it expires?

12. Show the changes which would be made to the table if QXFER were called with the following transfer table specified.

400000	QFIX
-1	-3

AC J/ 1

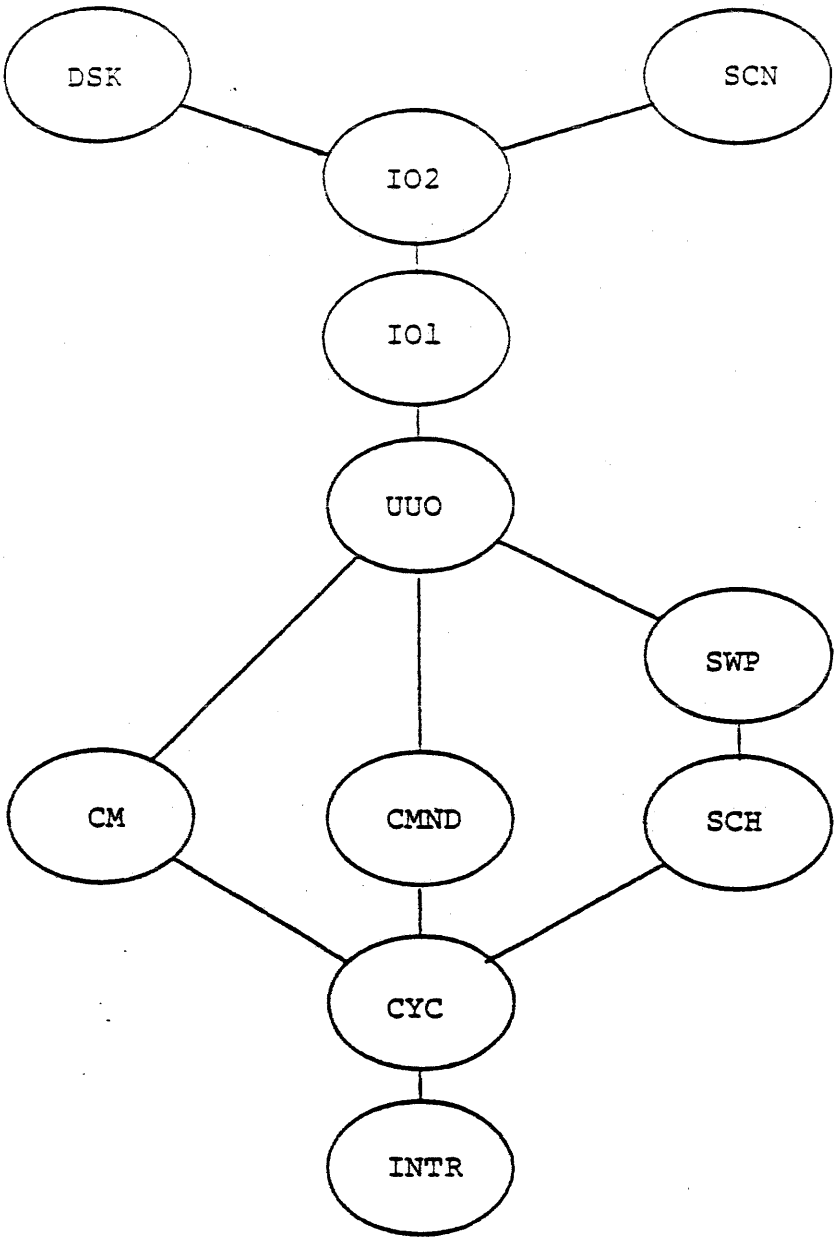
-3	3	5
-2	-2	-2
-1	2	4
JBCTQ	0	0
1	4	6
2	6	-1
3	5	-3
4	-1	1
5	-3	3
6	1	2

13. What is the effect of setting the JS.000 bit?

This page is for notes

TOPS-1Ø MONITOR INTERNALS

The Swapper



COURSE MAP

This page is for notes

THE SWAPPER

INTRODUCTION

The swapper allows TOPS-10 timesharing to run efficiently by moving jobs of variable size in and out of core. This process is made difficult because it must handle not only swap selection, disk I/O, and swapping area maintenance but also take into account scheduler decisions and high segments.

RESOURCES

TOPS-10 Microfiche

TOPS-10 Monitor Internals Course Supplement

Course Materials - Chapter 6
Supplement - Graphics Section 6
- 603A Scheduler/Swapper PLM
  Read Chapter 4
- 603A Scheduler/Swapper Flow Charts

LECTURE OUTLINE

I. Introduction
   A. VM vs. Non VM
   B. Clock Level vs. Interrupt Level
   C. Swap Time vs. Monitor Cycle Time
   D. Swapping IO vs. Paging IO

II. Data Base

III. Major Algorithms
   A. Criteria for Swap-In
   B. Criteria for Swap-Out
   C. Migration
   D. Nofit
   E. High Segs

IV. Disk Swapping Space Management

V. Flow Charts - Cases

THE SWAPPER

Introduction

Swapping allows the timesharing system to present the appearance of having more core memory than it actually has. The core images of some subset of the user jobs are written in disk or drum memory and read back into core memory as required. Ideally, no user can tell whether his job is swapped in or out. Whether or not this artifice will succeed depends on many factors, including the number of users, their interaction rates, their core requirements, the speed of the swapping device, and the amount of physical core memory. In this chapter we first discuss the overall philosophy of swapping on the DECsystem-10 and then the procedures by which this philosophy is implemented. We shall see that the procedures are general enough to allow a great deal of flexibility in policy.

Swapping Philosophy

Under light loading all active jobs will be in physical core. However, if physical core is needed, a job may be swapped out when the program has been stopped and requires a response from the user. Each time the user requests another function there will be a short delay while the core image is swapped in, and the program will run 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 will probably notice no delay. Under these conditions, swapping can be quite successful.

Under heavier loading, the system will be unable to keep all active jobs in physical core. 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 of the possibility of jobs which are in core being blocked by a job which is swapped out. System overhead may spiral due to the requirement for swapping a job many times for a single interaction. The effective size of core memory is reduced because an active job which is being swapped does not contribute to overall system efficiency. It is not available to run, and cannot have an I/O in progress.

However, the physical core memory which it occupies is not available until the swapping transfer is complete. The total system throughput would probably be greatest if runnable jobs were never swapped out. However, we must balance the total throughput consideration against the value of dividing the system resources fairly among all the users. We would prefer the system to appearing twice as slow to all users than four times as slow to half the users.

As swapping is implemented on the DECsystem-10, we do not directly distinguish 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. We define three basic levels of priority, and normally give jobs at each level complete priority over jobs at the next level. Some exceptions are made to this policy in the interest of fairness. (Real time jobs also get special treatment which will not be considered here.)

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 which require the 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 jobs which, in some way, are considered interactive. This class includes jobs doing I/O, all jobs which own shareable resources or are waiting for them, and all jobs which have had a recent user response. We assume that these jobs are the ones which are making the most use of system resources, or whose users will be less tolerant of delay. We hope under normal operating conditions to keep at least the jobs in these top two levels in core.

The lowest priority for core memory is given to CPU bound jobs. We assume that if a job is CPU bound, it is doing a considerable amount of processing, and that the user does not expect immediate response. Also these jobs would not contribute to the total system I/O throughput. Because they are not using system resources other than the CPU and core, there is no advantage to having more than one of these jobs in core at a time.

As we shall see, the actual implementation is somewhat more complicated than that discussed in the preceeding paragraphs. It is important, however, to understand the overall philosophy on which the implementation is based. We shall then consider the details as we reach them.

Mechanics of Swapping - - Data Structures

Two Scan tables, ISCAN and OSCAN, specify the relative priorities of jobs for being in core as a function of the queue positions. The entries in these tables are fixed in any specific Monitor, although they may be easily changed to implement a change in swapping policy. In addition to ISCAN and OSCAN, all the tables which 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 will be the first to be swapped out and will not be considered for swapping in. The Command Wait Queue will have top priority for being swapped in and will be followed by PQ1. In general, queues which are in ISCAN will be listed in the opposite order to OSCAN. All queues (except possibly the IO Wait Queues) will be in OSCAN. ISCAN will specify only jobs which are runnable or soon will be runnable.

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 acts by specifying the time when the job is again 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 units actually being used)

(PROTØ) + [(PROT) * (Size in K)]

PROT and PROTØ 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 PROTØ and Ø seconds PROT. This default ICPT of 3 seconds applies to all job,

regardless of size. PROT may be modified via the SCDSET program to make the ICPT a function of size.

Several other items are important to the Swapper. The table JBTSWP tells the size of each swapped out core image and the amount of core required when the job is to be swapped back in. (Note that 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 size and location of the job data area for each segment which has any physical core assignment. A job's JBTADR entry is still set up while it is being swapped either direction. The physical core assignment must be made before we can begin to swap a job in and cannot be canceled until it is completely swapped out.

The Swapper Cycle

The Swapper operates on an overall cycle which it repeats as often as possible. This cycle typically will require many jiffies to complete. Hence, the Swapper must operate as an asyncronous process under the control of the Monitor, rather than as a simple closed subroutine. Each clock tick the Scheduler dispatches to the Swapper's single entry point. 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 will attempt to continue on the next clock tick.

Although we know that the entire Swapper cycle normally requires many clock ticks, we shall initially look at the cycle as a continuous process. Then we shall look at a detailed flow chart which includes all the exit points. In both cases, we shall restrict our attention to the swapping of jobs consisting only of low segments. The general approach to swapping high segments and some of the special

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

problems involved, will be discussed in the final section of this chapter.

A macro flow chart of the entire Swapper cycle appears further into this chapter. The first step of the cycle is to choose the job to swap in. The job number will be stored in FIT, and remembered from this point on. All the following actions are directed toward the goal of getting this job into core. If there is not a job which needs to be swapped in, we check for jobs which must be swapped out in order to expand. If there is such a job, we proceed to swap it out. If there is not, the Swapper has nothing to do.

Once a job has been chosen to be swapped in, all further actions have the objective of creating enough room for the size specified by its incore image size in JBTSWP. The first step is to ensure that the total amount of available core exceeds the amount required. If CORTAL is less than the amount required, we choose a job to swap out, and force it out of core. This is a somewhat involved task and will be described in detail later. If CORTAL is greater than or equal to the amount of core required, then it is possible to make the required assignment without swapping out any more jobs.

If there are enough free pages, core will be assigned. If not, dormant and idle segments will be 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 setup to read in the core image. When the transfer has been completed, the necessary housekeeping is performed and the Swapper cycle begins again.

Choosing the Job to Swap Out

The relative priorities of jobs for being swapped out are specified by the scan table OSCAN. When we must swap out a job we scan the in-core queues according to OSCAN looking for jobs eligible to be swapped. We reject any job whose in-core protect time has not expired or which is locked in core. We keep a tally of the amount of core which we have checked. When this tally reaches the amount we need, we stop scanning and choose the first of the jobs which we found available as the one to swap. Note that if

we swapped the jobs strictly according to priority, all of the jobs we examined would have to be swapped.

Example: Suppose we need 10K and the jobs in the order specified OSCAN are set up as follows:

Job	Physical Core
14	2
20	LOCKED
6	6
12	4
10	12

We would stop scanning when we reached Job 12, and Job 14 would be chosen to be swapped out. If the queues were unchanged after we completed swapping Job 14, Job 6 would be chosen on the next scan.

Swapping I/O

All I/O for the Swapper is performed by the Disk Service Routine according to requests set up by the Swapper. Since we have not yet studied the Disk Service in detail, let us assume that we have a black box routine which will write 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 has been completed.

Swapping space is reserved on a per unit basis when the disk storage facility is initialized. This space is marked as in use so 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--drum, fixed-head disks, or disk pack. 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 the fastest device is assigned the lowest numbered class. When the Swapper needs to find space to write out a segment, it will start with the lowest class. It will scan through its SAT for each unit in the lowest class, and if it finds a large enough hole, it will use the corresponding area for the transfer. If it cannot find a large enough single hole, it will search for several holes which altogether have the required amount of space. If it finds enough space, it will write the core image as several fragments within that class. If there is not enough space altogether within a class, it will try the next class. If no single class has enough space, it will fragment the image across classes.

Shareable segments are normally left on the swapping device if there is a copy in core or even if the segment is dormant. This prevents our having to write the same segment out to the swapping device at a later time. If there is not enough free swapping space, we will delete one of these unnecessary segments, and rewrite it when we have to. However, there is normally no reason not to have enough swapping space for all dormant segments.

Examples

Before going into further details of the Swapper, let us look at some examples of how jobs will be swapped under some highly simplified loading conditions. In the examples we shall use specific values for CPU time requirements, but average or "expected" values for delays. The average delay time will more closely approximate the effect of the delay when actual CPU times vary randomly about the specified average value. Also, in these examples, we shall not consider the additional complications introduced by user I/O, competition for shareable resources, or any transient conditions. It is quite possible that in actual operations, these might have significant, or even dominant effects.

Example 1. Small, Interactive Jobs

A small, single disk pack system runs 20 jobs. The available user core is 100K. Each job runs in 10K and requires an average of one-tenth second (100ms) of CPU time for each interaction. Each job does only a small amont of I/O to disk and TTY. Hence, swapping and CPU time are the primary considerations in computing expected performance.

Normally, the Swapper will choose a job to swap in on the next clock tick after a user inquiry. Thus, there will be an average 8 ms (1/2 clock tick) delay between the user action and the Swapper initial actions. We expect to have core filled with inactive jobs. Hence, we will have to swap a job out to make room for the job to be swapped in, but there will be swappable jobs available. We shall assume that the job to be swapped out has no I/O in progress and that the Swapper can immediately submit its output request to the Disk Service. Assuming that the disk access arms are randomly positioned, and that the swapping space is in the center of the pack, the seek will take an average of 28ms (407 cylinders). The transfer can be initiated immediately upon completion of the seek. There will be a 8ms (1/2 rotation) average latency, or rotational delay time, followed by a 60ms transfer (5.6 micro-sec per word X10K). The Swapper will not be called again until the next clock tick, giving an average 8 ms delay. On the next call the Swapper can start the input transfer. We assume another 28ms seek for this transfer. There will be another 8ms latency, 60ms transfer, and 8 ms delay until housekeeping for swapping in the active job. The job will then run for 6 consecutive clock cycles (100ms). Assuming ideal conditions, where the user requests are regularly spaced, with one arriving every 1/2 second, each user would get his job completed within 288ms. This would appear to be instantaneous response. There will then be a 212ms (500-288) period while the system is idle and waiting for the next user request.

In the worst case, where all 20 users made a response at the same time, the last request finished would require 5.7 seconds (if we assume the same sequence of actions for swapping each job).

Operating as described, the system would repeat a cycle which takes an average of 500 ms. Of this 500 ms, 100 ms of CPU time would be used for the user job. Of the Null job time, 188ms would be counted as lost time and 212ms as idle

time. Hence, the system would show 20% user program CPU time plus 38% lost plus 42% idle.

We might ask how many such jobs the system could support. At saturation, the running of one job would be completely overlapped by the swapping of another. Assuming the same sequence of actions for swapping, we could have a maximum of one inquiry every 188ms, or 53 users. (Note, however, that any disk I/O done by user programs would increase swapping time, and decrease the maximum request rate accordingly.)

Example 2. Large, CPU Bound Jobs

Suppose ten users start compute bound jobs which require 40K each and 18 seconds CPU time. The system has 50K available user core. Hence one job will be in core while the other nine are swapped out, therefore the jobs will be processed and swapped on a round robin basis. Assume that all jobs circulate entirely within PQ2 and the ICPT in each job is greater than 6 seconds.

The scheduler will choose the job in core to run. Since this job is the only job in core it is the only runnable job hence it will be run in the PQ2 quantum run time, 6 jiffies, and then requeed to the back of PQ2. During this time the swapper will have choosen a swapped out PQ2 job to swap in but found no eligible jobs, i.e. with expiled MCU, to swap out. Since the MCU is assured to be greater than 6 seconds this NOFIT condition will persist until the swapper becomes frustrated. During this time the job in core will be run continuously regardless of position within PQ2. After the Frustration timer has goone off the job in core will be eligible for swap out after which the original job choosen for swap in will be brought in.

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

The system will report a cycle in which a job gets 6 seconds CPU time and then 536 msec are spent swapping. Each job will run for one such cycle then be swapped at for nine cycles.

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

Detailed Flow Chart

A detailed flow chart of the Swapper appears in the supplement. This flow chart indicates the various exits from the Swapper and the decisions necessary to continue the process on the next call. In studying the flow charts you must keep in mind that the Swapper is an asynchronous process relative to the overall Monitor cycle. In between calls to the Swapper, life goes on in the rest of the system. New jobs are created, and existing jobs are logged off. Core assignments are changed. Between calls to the Swapper, core might be either taken or freed as a result of commands and UUO's. The hole which we were preparing over the last eighteen clock ticks could suddenly disappear. Hence, the Swapper must recheck the core situation on each call.

Note that a job cannot be swapped while it has active I/O. Once a job is chosen to be swapped out, the Swapper can go no further until the job's I/O has stopped. It does set bits to prevent the job from being scheduled to run and thereby start more I/O.

In the actual flow, we first of all attempt to determine where we left off on the last call. There are several words which are used as flags in this process. FIT contains the number of the job chosen to be swapped in. If a job has been chosen to be swapped out, its number is in FORCE. If we are waiting to shuffle a job, its number is in CHKSHF. If there is a swapping transfer pending, SQREQ will be nonzero. And on the next call after completion of the transfer, the number of the job just swapped will be in FINISH. In determining where we are within the cycle, we

start with the innermost levels of the overall logical cycle. We check flags, working our way back toward the beginnings of the cycle until we determine what needs to be done next.

Notes which follow the flow chart are referenced by the numbers within parentheses inside the blocks.

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 is 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. It also contains a linked list of pages not in use.

MEMTAB is also one word per page of core. It is used during swap requests to keep track of where pages end up on the swapping area and which page to transmit next. The format is:

```
Bit Ø      - on for the last page in a fragment
Bits 5-17  - next virtual page in this fragment
Bits 21-23 - unit number in active swapping list
Bits 24-35 - page number on unit
```

Note: Bits 21-35 are the format of all pointers to the swapping space.

JBTSWP is used the same way it used to be while allocating swapping space. After allocation, an entry for a segment has the following format:

```
Bit Ø      - on indicates a fragmented segment
Bits 1-17  - swapping pointer (as in the RH of
             MEMTAB) if not fragmented, the
             address of the fragment table if
             fragmented).
Bits 27-35 - segment size in pages
```

after allocation, and before the swap is queued, if the seg is a low seg, LH of JBTSWP becomes the swapping pointer for the UPMP.

Fragment table entries are put into the four word space and are the same format as JBTSWP entries terminated by a zero word. If there is not enough room in the current four word entry, there will be a fragment pointer (bit Ø on) to the next four word block.

There are 3 parallel tables used by the swapper to keep track of the jobs currently under its control (swapping and paging). They are SWPLST, SW2LST, and SW3LST, each SLECNT long. If more than SLECNT jobs are given to the swapper, you will get a STOPCD.

SWPLST is used to keep track of the progress of the I/O on the job it is assigned to. Its format is:

Bit Ø	- on if fragmented swap
Bit 1	- direction of I/O (on if out)
Bit 2	- swapping or paging (on if swapping)
Bit 3	- I/O in progress
Bit 4	- I/O is done
Bit 5	- on if an IPCF page
Bit 12	- I/O error (IODERR, IOTERR, or IOIMPM)
Bit 13	- channel error (IOCHMP or IOCHNX) if not a fragmented entry
Bits 14-26	- starting physical page number
Bits 27-35	- number of pages if fragmented
Bits 18-35	- address of fragmented table

The fragment table is linked the same way the JBTSWP fragment table is, but the entries are as above.

SW2LST is used to save the original SWPLST entry during the swap because it is needed for cleanup, but the SWPLST entry is modified while I/O is progressing.

SW3LST contains the job number in the right half, the contents of SWPOUT as of when the entry was created in the left half.

Swapping High Segments

High segments are never chosen directly to be swapped in or out. We only consider job numbers (i.e., low segment numbers) 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 nonshareable high segments are always swapped at the same time as their low segments. A shareable high segment is never swapped if there are any jobs still in core which are using it. Also, shareable high segments normally have to be 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 shareable high segments will be 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 we need the space which it occupies. However, we will always be sure to have a copy on the swapping device before deleting the core image. A high segment is swapped in whenever a job which is 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 which swap jobs in and out look at the numbers in FIT and FORCE as segment numbers and swap the specified segment in or out. We alwways choose a job to swap in. We swap the low segments in first and then check if it has a high segment which needs to be swapped in. If it does, we put the high segment number into FIT and go back through practically the entire routine to swap in the high segment.

When we choose a job to swap out, we check before swapping it out to determine if it has a high segment which should be written out. If it does, we store the low segment number and swap out the high segment first. Upon completion of swapping out the high segment, we find that it has a corresponding low segment to be swapped out. We then put the low segment number into FORCE and repeat the swapping-out routine. The reason for this is that we cannot write out the low segment until all its I/O stops. A

shareable 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 we wait for I/O to stop in the low segment.

Complications and High Segments

A number of complicatons can arise in swapping jobs with shareable high segments. Let us identify several cases and consider them individually.

1. Low Segment in Core, High Segment Swapped Out

This case does not normally occur because the high segment is normally not removed from core until the last job is swapped out and is brought back with the first. However, it is possible for a job to do a RUN for a swapped out high segment. The low segment could possibly be set up in core while the high segment is swapped out. In this case, the low segment will be marked as swapped even though it is in core. When we choose it to swap in, we see that the low segment is already in core and proceed to swap in the high segment. This problem can also occur on a GETSEG.

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 nonzero 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 will eventually choose the job to swap out. Upon finding that 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 user could run another program or the program could do a GETSEG, detaching from the high segment and leaving it with no attached low segments in core. When an idle segment is chosen to be deleted, we check if it is in this situation. If so, before deleting the core image, we force the segment to be written out.

EXERCISES

1. Under what conditions will a job be swapped out?

*2. In what order does the swapper consider jobs for possibly swapping in? What determines this order?

3. What conditions must be met before a high segment will be swapped out?

4. Why can the swapper be sure of a successful return from CORGET when it tries to assign core for a job to bring in?

5. If both segments of a two segment job are to be swapped out which segment goes first? Why?

6. If two segments must be swapped in, which segment is swapped in first?

*7. What determines the order in which jobs are considered for swapping out? What is the order in which they are considered?

8. What happens if the swapper can't find enough room to swap in the job it has chosen?

9. Why could CORTAL be greater than BIGHOL?

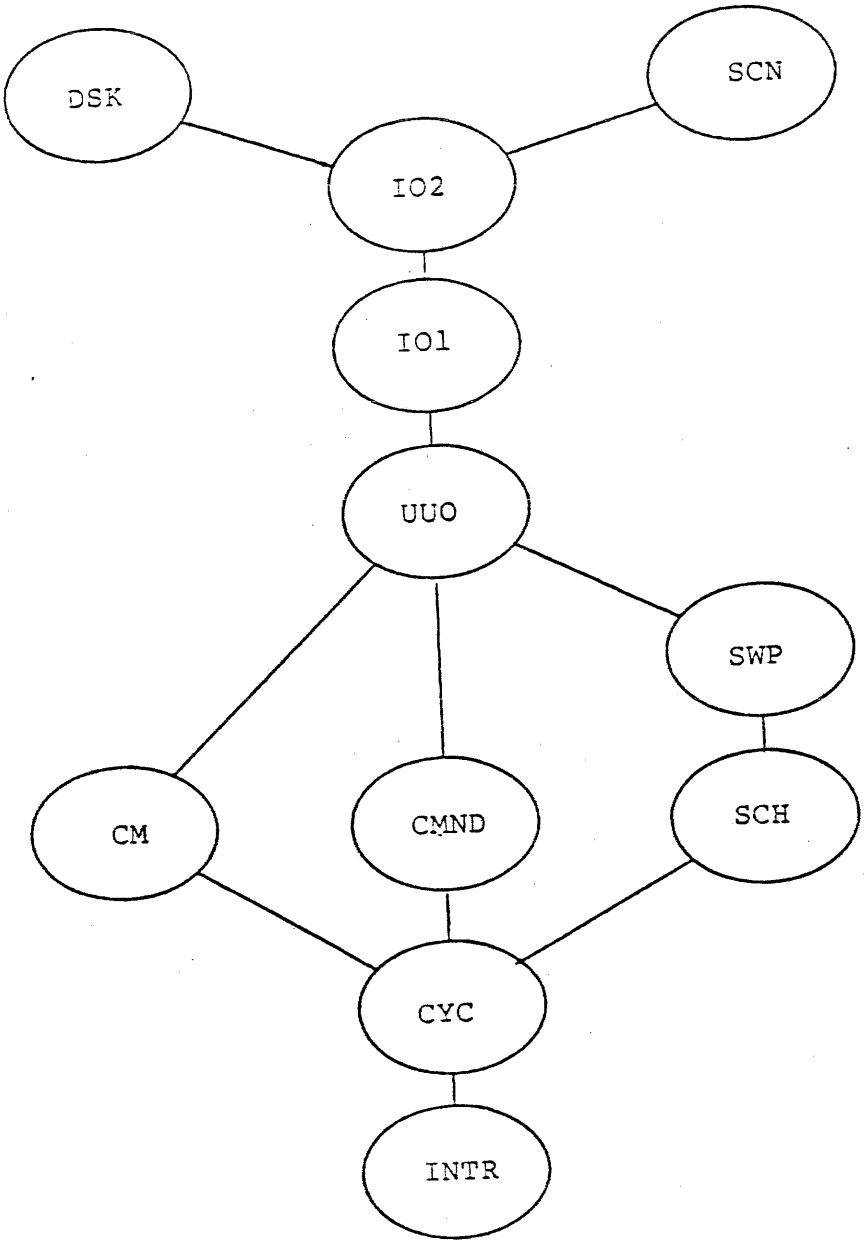
10. Suppose the swapper has selected a job to swap in and has been making room for it by swapping out jobs over a number of clock ticks. If, when there is enough room, a higher priority job has now become eligible to swap in, which job will actually be swapped in? What justification do you see for this?

11. Why would a job be rejected by the swapper when it is looking for jobs to swap out?

12. Under what conditions will a high segment be swapped in?

TOPS-1Ø MONITOR INTERNALS

UUO Processing



COURSE MAP

This page is for notes

UUO Processing

INTRODUCTION

UUOCON is the monitor module that executes programmed operators (UUOs) for the user. It must perform three functions: UUO preprocessing, dispatch to the correct service routine, and exit. This module will explain how each operation functions and how to add new UUOs.

In order to introduce the topic of crash analysis, this module will explain how illegal UUOs are handled.

RESOURCES

TOPS-10 Microfiche

TOPS-10 Monitor Internals Course Supplement

Course Materials - Chapter 7
Supplement - Graphics Section 7
Crash Analysis Guide - Chapters 1-3
Chapters 4 Section 4-1 thru 4-4
Chapters 7 and 8

LECTURE OUTLINE

I. Introduction
   A. Why Have UUOs
   B. What Are They
   C. Methods of Passing Arguments

II. Trapping

III. Data Base
   A. UUOTAB
   B. UCLTAB-UCLJMP
   C. CHCKTAB-CHKTBC
   D. CNAMES and NAMES MACRO

IV. Flow Charts

V. Adding a UUO

VI. GETTABS

VII. UUO Error Handling
   A. User Mode
   B. UIL Interrupt Level
   C. EUE

PROGRAMMED OPERATOR SERVICE (UUOCON)

DESCRIPTION

The function of UUOCON is to service in some manner those codes which are trapped to locations 424 and 425 of the current job's UPMP by the processor hardware. These are op codes 000, 040 through 077, and (in user mode) 7xx (input/output, HALT (JRST 4,), JEN (JRST 10,), and the unassigned operation codes.

The operations of UUOCON might, for the purpose of discussion, be divided into three sections.

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

Preprocessing includes switching to the exec AC block, setting up a push down list in the job's UPMP (for use by the monitor during UUO execution), saving the return PC on the stack, loading of accumulators with information to be used by the operator service routines and dispatching to the proper service routine.

Operator service routines perform the algorithm designed for the particular UUO, allowing the user to receive information about the system, to alter the operation of the system concerning his job, and to communicate with the input/output devices. A few specific examples are included in this module to demonstrate the information flow between the three sections of UUOCON and the user's job. Input/output UUO's are dealt with in the chapter on Input/Output Service.

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 exit routine calls the Scheduler if the interlock flag was set sometime during the UUO processing.

OPERATOR PREPROCESSING AND DISPATCH

SPECIAL REGISTERS

A rather important function of this section is to place information about this user's job (i.e., the job that issued the UUO) into certain accumulators and index registers before dispatching. Therefore, these registers and their contents are described briefly before going into the operations of this section.

P — A pushdown pointer to a 133 (octal) location list in the user's page map area. The first item placed in this list (.JBPDl) is the user's return; i.e., a copy of the PC word in location 425 (UPMP).

R — Contains a copy of the contents of JBTADR: XWD highest (KA) relative address, relocation for this job. Used as an index register by the system to relocate references to the user's program area on KA systems.

On KI and KL processors the first page of the users virtual address space, which contains the JOBDAT, is accessed via executive virtual page 341. Hence unless a job is locked in core, R will contain the contents of JBTADR which is job size in the LH and 341000 in the RH.

M — A copy of the programmed operator as trapped into location 424. The address R is set into the X field so that operator service can refer to (E) indirectly through M.

Pl — A copy of the AC address field of M. Pl could be for a User I/O Channel, which it is in the case of

input/output operators.

F* A copy of USRJDA (protected .JBJDA) for this software channel. This register contains Ø if this channel is unassigned. If the channel is in use, the left half of this word has status bits indicating what UUO's have been performed for the device so far; the right half contains the base 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 this word contains the address of the next DDB in a chain of all such blocks; the right half contains the base address of the dispatch table for this device's service routine.

W Contains the address of the job's PDB.

---

* These registers are pertinent only to input/output programmed operators, but will be loaded, in any case, when an AC address (P1) happens to correspond to an assigned I/O channel.

FUNCTIONAL DESCRIPTION

The following is a narrative of the operator-independent preprocessing and dispatch section of UUOCON.

MUUO (COMMON) After selecting the EXEC AC block the user mode flag bit of the trapped PC word is used to detect whether the call is from the Monitor (as in a GET command) or from the user. If from the Monitor, certain AC's have been set up and a portion of the UUOCON coding can be skipped; control goes to UUOSY1. If in user mode the user's AC's are saved

in the .JBAC part of his job data area (KA only) and 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 dual processor systems when one processor stops running a runable job while the other processor is running the null job. This event forces the CPU running the null job to stop and select the runnable job to run.

UUOSY1 This routine in UUOCON loads register M with the UUO itself and J with the job number.

If the UUO is in the op-code range of 0-37 control is transfered to UUOERR in ERRCON.

The return PC is taken from lcoation 425 in the UPMP and placed on the stack to ensure that it is not overwritten in the event a UUO is done by the monitor itself.

The op code 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 op code 72 or greater (long dispatcch I/O UUO) then routine DISP1 is entered.

DISP0 is entered directly for non-I/O UUO's or I/O UUO's between codes 55 and 71 if the channel is found to be assigned.

DISPO This coding obtains an address from a 2-address-per-word dispatch table using the op code as an index. Prior to dispatch routine UUOCHK, in module VMSER, is called to verify that the UUO arguments are incore; if not control will be transfered to the users PFH which will page in the page or pages containing the UUO arguments. This approach is taken so as to prevent a page fault from occuring as a result of a memory

reference made by the monitor. If this UUO was 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 it was 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 DISPØ if the UUO was 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" one and the device service routine does not have a long dispatch table (this is an important concept in making user programs "device independent"; e.g., it enables a LOOKUP to a physical paper tape reader to be "successful"). If the device service routine is capable of performing long UUO's, the dispatch routine DISPØ is called.

OPERATOR SERVICE

Before discussing a particular operator, let us first see how communication between the user's program and the monitor takes place. Information is passed to the monitor thru the user AC block or thru argument lists somewhere in the users address space. These lists as well as the actual arguments themselves may be in a page of address space that is incore or paged out.

The primary method of referencing UUO arguments, given a user virtual address, is by the use of the executive execute instruction (PXCT). However, before making memory references to a user virtual address two conditions must be verified. One that the address is a valid virtual address for that users address space and secondly that the page containing that address is incore. There are also some locations in the JOBDAT that need to be protected as well as 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.

UADCK1 - Takes successful return if the address being checked is an AC, otherwise falls into UADRCK.

UADRCK - Called only from UUO level, however the address being checked may be referenced from interrupt level sometime in the future hence AC references are illegal. References to locations in the protected part of JOBDAT and to pages that don't exist (> USRREZ) are also rejected. If the reference is to a paged out page the job's page fault handler will be invoked to get the page incore before proceeding or 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 will be stopped and the message "ADDRESS CHECK ....." will be typed on the user's terminal.

IADRCK - This routine is called from interrupt level primarily for I/O buffer address verification. Hence references to ACs, 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 proceeds to make the memory reference through the use of the PXCT instruction. In some instances the PXCT instruction will appear inline with the code that called the routines to verify the addreses. The EXCTUX, EXCTXU and EXCTUU macros will be used to generate the appropriate PXCT instruction. In other instances calls will be 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 via 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, STOTC1, that accomplishes the same result as STOTAC but takes a skip return.

In returning to the user, one may wish to skip one or more arguments that followed the UUO, or to give a skip or non-skip return to signify 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, when at the level equal to that following the dispatch, a POPJ P, is used to exit, the user will receive a non-skip return. If the sequence

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

is used, a skip return occurs. This could be used to bypass one argument 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 take care of adding the correct quantity to the correct entry on the pushdown list (recall that, if the original UUO was issued by the Monitor, the preprocessor dispatch was not a PUSHJ). If, for example, two arguments are to be skipped in return to a user mode call, this sequence could be used.

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

To give the same return to a call from the Monitor,

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

Example

All operators that do not deal with some phase of input/output are invoked through the use of the CALLI UUO. To keep this example reasonably simple, we will choose one of these:

```
CALLI ac, 27
    or
RUNTIME ac
```

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

The preprocessor routine of UUOCON sets up the standard accumulators and, using the UUO op code (CALLI - Ø47), dispatches 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, in this case RUNTIM. This argument is used to effect another dispatch to the routine JOBTIM, which gets the appropriate run time and stores it in the user accumulator.

When entered, the JOBTIM routine checks the contents of T1 for a valid job number and then uses it as an argument to the FNDPDB routine to find the process data block for the job. The desired time is extracted from the PDB, cconverted to msec and placed into T1. A JRST STOTAC causes this result to be stored in the user's accumulator, now addressed by M, and return to the UUOCON exit routine.

EXIT ROUTINES

ERROR EXITS

Error exits, which do not allow a return to the user, occur when a UUO op code is illegal or an address supplied by the user is illegal. A nonimplemented UUO in the range 4Ø through 77, or a UUO of Ø will stop the job with the error bit on (cannot continue) and print "ILLEGAL UUO at USER loc." An illegal op code (e.g., a DATAI in user mode) causes the job to be stopped with the error bit set and the message "ILL. INST. AT ..." to be printed. The HALT instruction stops the job, types "HALT AT USER loc.", but does not set the error bit. Thus, the CONTINUE command does function after a HALT.

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 USER loc") 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 dev: EXEC CALLED FROM loc," and results in an error stop condition.

NORMAL EXITS

If the original UUO was issued by the Monitor, the preprocessor dispatch was by a JRST rather than by a PUSHJ. The service routine's last POPJ would bypass the user exit routine and go directly back to the Monitor coding following the call.

If the UUO was 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.

USRXIT This routine checks to see if the user has typed a CTRLC (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 (SCHED) to take appropriate action. If the user's job continues in the future, control will come back here to restore the user's accumulators and continue the job.

II. ADDING A PROGRAMMED OPERATOR

There are two ways to add a new UUO function to the Monitor. One is to use a previously unused op code (42 through 46). The other is to add an additional CALLI. Adding customer defined CALLIs with negative arguments is the prefered technique. Before adding anything to any section of the Monitor, it is, of course, desirable to understand what is already there. Assuming that one already has this understanding and has written a tightly coded new routine that obeys the rules of address protection and uses as much existing coding as possible, we can investigate the process of getting this routine included in a running Monitor.

ADDING A NEW OPERATOR

1. Edit the new coding 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 presently 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 the dummy entry for CALLI -2, in the CNAMES, definition would be changed. Conditional assembly could be used to set up the dispatch table entry if conditional assembly was used with the routine itself. For example:

```
Routine Coding                  CNAMES Entry
                        !
IFN FTDMPU, <UDUMP:     !       IFN FTDMPU,<
                        !
        (coding)        !       X UDUMP, UDUMP
                        !       >
                        !       IFE FTDMPU,<
                        !
                        !       X CPOPJ, CPOPJ##
    >                   !       >
```

   In this example, the routine will be assembled and the address of UDUMP is added to the dispatch table if the feature switch FTDMPU is nonzero.

3. In preparation for assembling the new UUOCON, edit the correct feature test switch settings into the S (system parameter) source file, including any new ones you have established.

4. Assemble, naming as input first the S file, then the new UUOCON file.

5. Use FUDGE2 to or 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 Monitor Installation Guide.

EXERCISES

1. When a UUO is executed, what will be the contents of location 424 (UPMP)?

2. What is the range of op codes which are legal monitor UUO's?

*3. How is the address of the routine for a specific UUO determined?

4. Where are the user's AC's saved while a UUO is being processed?

*5. Suppose a UUO cannot run to completion (IO problems, etc.). When context switching occurs, USRPC (.CØPC) will contain the address for restarting the job somewhere in UUOCON. How does UUOCON then know the return address in the user's program, considering that UUOCON may have been called by any number of other jobs in the meantime?

*6. What action does UUOCON take if a UUO requiring a long dispatch table is executed, and the specified device service routine has a short dispatch table?

*7. How does UUOCON respond to a UUO without a currently defined function; i.e., Ø42?

*8. How does UUOCON respond to a CALLI UUO with an undefined function; i.e., CALLI, 400?

*9. Which AC's are loaded with what values by UUOCON, before it dispatches to the routine for a specific function?

10. Write the routine and specify all necessary monitor modifications to implement the following new CALLI:

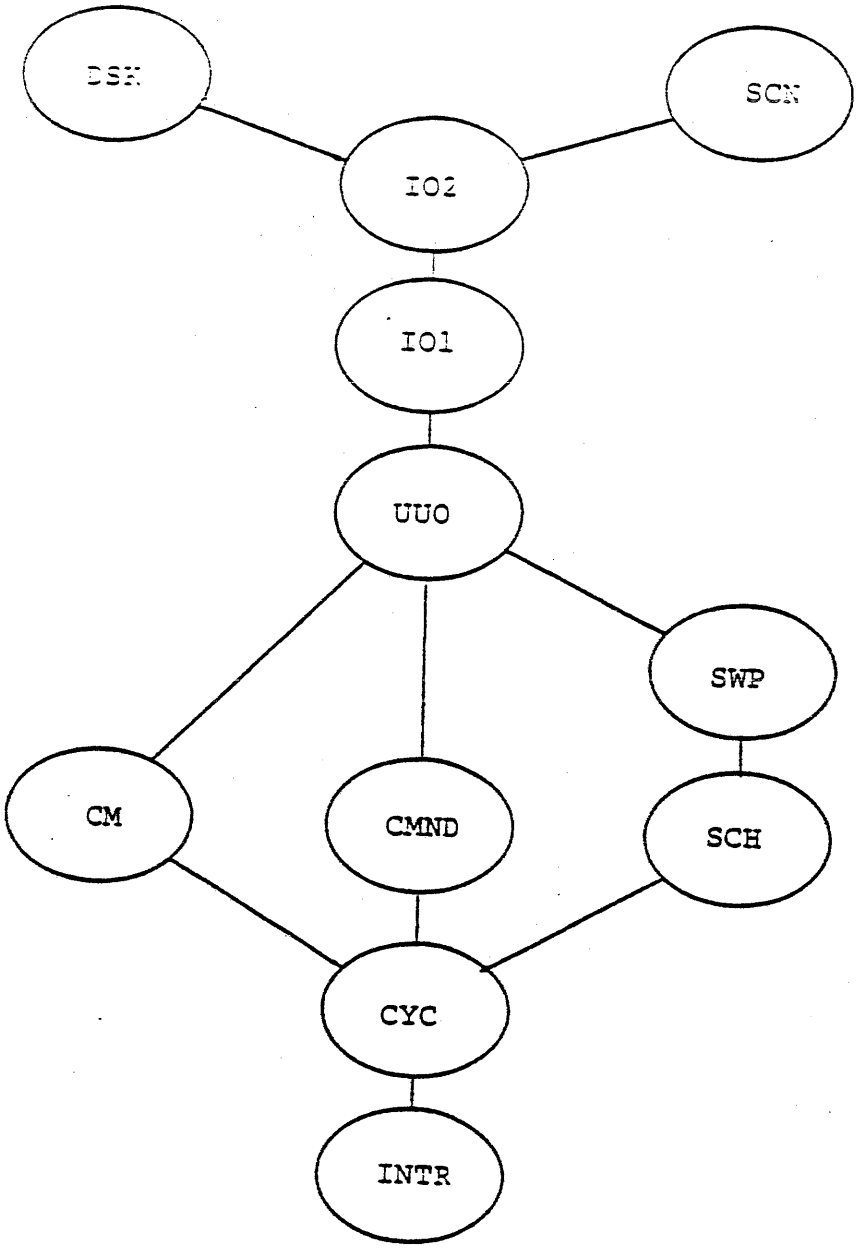
OPDEF CHAN [CALLI -4]

The CHAN routine will put into the user's specified AC the number of the first unused software channel for his job.

This page is for notes

TOPS-10 MONITOR INTERNALS

I/O Introduction and UUO Level Routines



COURSE MAP

This page is for notes

I/O INTRODUCTION AND UUO LEVEL ROUTINES

INTRODUCTION

User programs perform I/O using a select group of UUOs (OPEN, INIT, IN, CLOSE, FILOP, etc.). Each UUO requires two phases: device independent processing and device dependent routines. This module will discuss the first of those two phases; i.e., what each UUO does at the device independent level.

RESOURCES

TOPS-10 Microfiche

TOPS-10 Monitor Internals Course Supplement

Course Materials - Chapters 8 and 9
Supplement - Graphics Section 8 and 9

LECTURE OUTLINE

I. Introduction

II. Hardware Principles
  A. I/O Bus
  B. Data Channel

III. I/O Routines Organization
  A. UUOCON
  B. Device Service Routines

IV. Device Independent UUO Level Flows
  A. INIT
  B. INBUF
  C. INPUT
  D. CLOSE input
  E. OUTPUT
  F. CLOSE output
  G. RELEAS

V. Device Service Introduction
  A. DDB
  B. Dispatch Table
  C. Interrupt Level

VI. DDB
  A. Purpose
  B. Table Description
  C. PTP Example

VII. UUO Level Code
  A. Dispatch Table
  B. PTP Flow

This page is for notes

I/O INTRODUCTION AND UUO LEVEL ROUTINES

Input-output handling in the DECsystem-10 Monitor is based on the objectives of device independence and modularity of code. Any user program should be able to operate using 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. Modularity of code plays an important role in meeting this objective. 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 IO modules are organized makes it possible for an installation to add code to handle a special device without changing the existing code in any way. 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 which apply to standard devices.

In this module we briefly discuss the hardware principles which apply to IO processing. Then we look at the organization of the IO processing code and the functions performed by various modules. Next we examine the device independent functions performed within the UUO processor.

Next we discuss device service routines which perform the device dependent functions, and the macros used to generate configuration dependent code. Finally, we examine timing problems which must be taken into consideration by device service routines and some techniques by which these problems are solved.

Hardware Principles

All IO transfers are done by DATAO or DATAI instructions. Each such instruction addresses a specific device (or controller) by a device code in bits 3-9. 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 will continue execution of the program. The device, however, will not be ready to accept another instruction for a relatively long time. When it is ready, the device will request a priority interrupt.

There are two basicaly different types of IO devices, IO bus devices and data channel devices. IO bus devices require use of the IO bus for each word transferred. Data channel devices require use of the IO bus only for the initiation of the transfer.

IO bus devices will cause an interrupt for each word (or character) to be transferred for the slower devices, an entire interrupt routine will be executed on each interrupt. This interrupt routine will check for reaching the end of the buffer, check the device status for error conditions, and normally will request the next transfer. The faster devices, DECtape and non-DMA magnetic tape (TM1ØA) 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 next 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 will be executed. This instruction will call an interrupt routine which will do the necessary housekeeping and set 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 TM1ØB magnetic tape controller, and all disk controllers, make use of a data channel to access memory directly without interrupting the CPU or using the IO bus. A single instruction is used to initiate 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. They are normally assigned a low priority interrupt channel. The hardware principles of data channel transfers will be discussed in more detail in the chapter on disk.

Organization of IO Routines

All device independent routines are contained in UUOCON. This includes outer level routines to handle all IO UUO's. The UUO decoder dispatches to these routines, with various "global" AC's set up, and the IO 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.

All device dependent code for one device is normally included in a device service routine for that device. There will be only one service routine for a given type device, regardless of the number of such devices or units in the configuration. These service routines are written to work for any possible number of units. Therefore, any configuration dependent code is included in COMMON (or COMMOD for disk). The Device Data Block for a device will normally be included in the service routine. For devices having multiple units on a single controller--such as DECtape--additional copies of the DDB are set up at system initialization time, according to information in a table in COMMON. Where there are separate controllers for several devices of the same type, a small amount of code is dependent on which controller is being used. (The line printer is an example of such a device.) The code which depends on the specific controller is incorporated into the DDB and put into COMMON. In COMMON, a DDB and the controller dependent code are assembled for each controller in the configuration. The bulk of the code, which is independent of the specific controller, is included in the single device service routine.

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 Device Data Block. 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

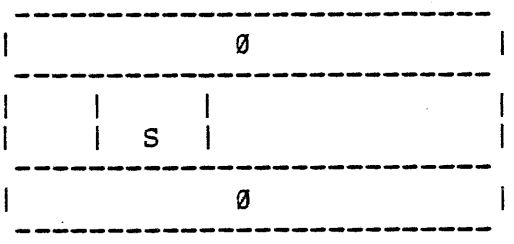
interfering with the interrupted process.

Independent Functions

In this section we examine the functions performed by the device independent routines in processing buffered IO. We begin with a general discussion of actions taken by each UUO routine. First we follow the steps taken by a program reading a file from an arbitrary device. Then we follow the steps taken by a program writing a file. Finally there is a set of annotated flow charts for the same routines. Throughout this section we concentrate on presenting a major concept. Neither the descriptions nor the flow charts are complete in every detail. Complete details would probably obscure the concepts more than they would clarify them. However, once you are thoroughly familiar with the material in this section, you should be well prepared to go into the listings for the ultimate detail.

INIT

The INIT (or OPEN) is the means by which the user program specifies the device which it wants to use. The main function of the 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 which matches the argument of the INIT. If this fails, it looks for a DDB having a physical device name which matches the argument. Finally, if a generic device name (e.g., DTA) is given, it looks for a DDB which is appropriate for that generic specification. If a DDB is found the Assigned-by-program bit, ASSPRG, is set in the DEVJOB word, and the job number is put into the DEVCHR word. The user bits in DEVIOS, including the data mode, are initialized according to the program's specifications. According to the data mode, the byte size field in the buffer ring header is initialized. The first and third words of the ring header are cleared. Hence, after INIT the ring header appears as follows:



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

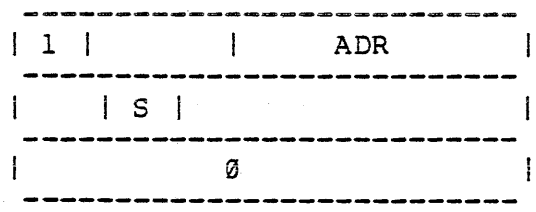
INBUF

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 will be called on the first INPUT UUO to set up a ring of two buffers.

The buffer ring is set up beginning at the job's first free location, according to the .JBFF word in the Job Data Area. If necessary, more core will be obtained. .JBFF will be updated to point to the first location after the buffer ring.

The length for each buffer is obtained by calling a device dependent routine. Normally, the length, including buffer header, will be two greater than the value obtained from DEVCHR, in the DDB.

The buffer linkages are set up, and the use bit in each buffer header is cleared. 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 is in an initial state, the ring buffer control block has the following form:



There is no call to the device service routine.

INPUT

With each INPUT UUO the user program asks that a 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 will start the device.

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

On all INPUT UUO's except the first, we clear the use bit on the buffer previously available to the user. Clearing the use bit indicates that there is no "good" data in the buffer. This informs the interrupt routine that it may continue reading into this buffer as soon as the previous buffer is full.

One very important function of the INPUT routine is to start the device. Whenever an INPUT UUO is executed and there is only one buffer remaining full, and the device is not running (IOACT=0), we call the device service routine to start the device. Since starting the device requires an actual IO instruction, it is always a device dependent function. It is, in fact, the only 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.

Before returning control to the user, we must ensure that the next buffer is full. If the use bit on the next buffer of the ring is set, it is already full. Hence, we can return control to the user immediately. If the next use bit is not set, we must not allow the job to continue running until the buffer has been filled. This is the function of a device dependent routine WSYNC, which is called with a PUSHJ.

WSYNC informs the scheduler that this job is to go into IO Wait. (It sets the wait state code IOWQ). It sets up its own return address, from the push down list, as the restart address for the job and exits to the monitor's outer loop. This job will be put into an IO Wait state and

another job will be chosen to run. It is the responsibility of the interrupt routine to get the job out of the IO Wait state when the next buffer has been filled. When the job is continued, it will be started at the next instruction after the PUSHJ to WSYNC. At this time, the next buffer should be full and control can be returned to the user program.

Note that--to the calling routine--WSYNC presents the appearance of a subroutine which can be called to fill the next buffer. From the monitor's point of view it is simply a way of terminating the job's time slice because it can not immediately continue. Note also that the UUO processor must be reentrant at this point. Before this job continues running, any number of other jobs may execute this same code. Hence, when we call WSYNC, all variables must be in job-dependent storage locations. Specifically, all stored variables will be either on the push down list in this job's UPMP 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 which the user sees, however. The purpose of IOEND is to prevent our trying 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 we reach end of file on the device. After the last buffer has been given to the user and he executes another INPUT UUO, we call the routine, CALIN, which normally dispatches to the device service routine to start the device. But CALIN 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 we call WYSNC, it does not put the job into IO Wait because IOACT is not set. At this time we set the user's end of file bit, IODEND, in DEVIOS. We give an error return 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.

CLOSE Input

The CLOSE UUO restores conditions to the initial state, ready to begin reading another file. The use bit is cleared in each buffer of the ring. The ring header use bit is set, indicating that the ring has been set up but never referenced. Both end of file bits are cleared. The Close

routine in the device service routine is called to perform whatever actions might be necessary. Most devices other than disk do not require any special actions on input close.

OUTPUT

In writing a file, the INIT and OUTBUF UUO's play roles directly analogous to INIT and INBUF in reading. One OUTPUT UUO is then issued for each buffer of data to be written.

The first OUTPUT normally does not write any data. The first OUTPUT is a dummy UUO by which the user program asks the monitor to set up his 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. We make the full buffer available to the interrupt routine by setting the use bit. If the device is not running, we call the device dependent routine to start it. Then we check the use bit of the next buffer to see if we can allow the user to put more data into it. If the use bit of the next buffer is not set, the buffer is empty, and we can allow the job to continue running immediately. If the use bit is set, there is still data in the buffer which must be written out. We call WSYNC to put this job in IO Wait until the interrupt routine restarts it. When the next buffer is free, we clear it to zeroes, and set up the buffer control block to allow the user to start filling the buffer.

CLOSE Output

On the CLOSE for an output file we ensure that any remaining buffers are written out, keeping the job in IO Wait until all buffers' use bits have been cleared. We call the device dependent routine for any device dependent functions. Then we restore the buffer ring to its initial state. The buffer control block is also reinitialized with its use bit being set to indicate an unused ring.

RELEASE

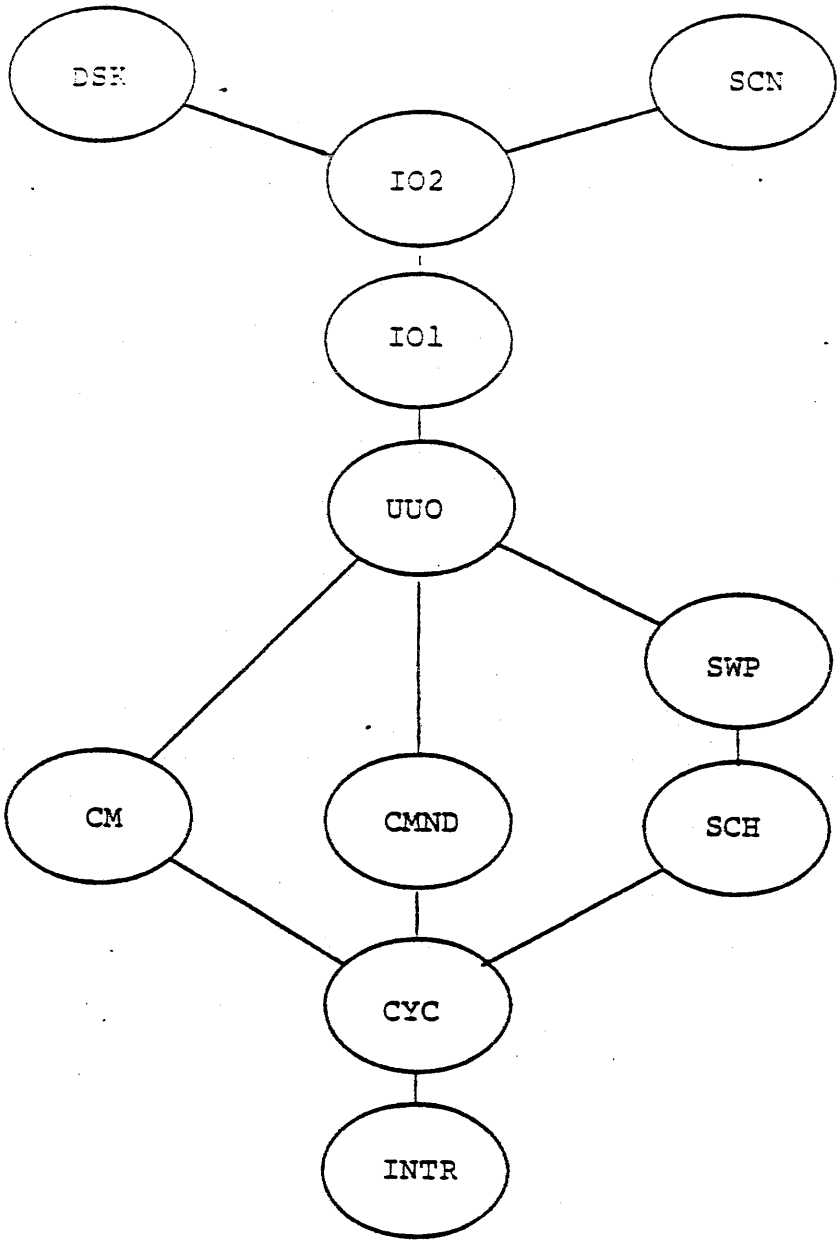
The RELEASE UUO countermands the INIT. It first does a CLOSE for both input and output and puts the job into IO Wait until the device is inactive. The device service routine is called for any device dependent actions. If the channel on which the RELEASE is being done is the highest channel in use by this job, we update the word where we remember the highest channel in use, JOBHCU. We clear the ASSPRG bit in DEVIOS, and unless the ASSCON bit is set, clear the job number 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.

Annotated Flow Charts of IO Routines

On the following pages are flow charts of each of the device independent routines for IO UUO's. The small numbers below and to the right of various blocks refer to notes at the end of the flow charts for that routine. The page numbers listed beside some connectors and subroutine calls refer to pages within that flow. These page numbers are at the top off the page.

TOPS-10 MONITOR INTERNALS

I/O DEVICE SERVICE ROUTINES AND INTERRUPT PROCESSING



COURSE MAP

This page is for notes

I/O DEVICE SERVICE ROUTINES AND INTERRUPT PROCESSING

INTRODUCTION

This module continues the description of I/O processing that was begun in the previous module. It focuses on the three components of device service routines: the device data block, UUO level routines and the interrupt routines.

RESOURCES

TOPS-10 Microfiche

TOPS-10 Monitor Internals Course Supplement

LECTURE OUTLINE

I. Interrupt Level
   A. Fielding of Interrupts
   B. Functions of Device Service Routines
   C. PTP Example

II. UUO Level vs Interrupt Level Interface
   A. Data Base
   B. Routines

III. I/O Macros
   A. Purpose
   B. INTTAB
   C. MONGEN Interface

I/O DEVICE SERVICE ROUTINES AND INTERRUPT PROCESSING

All device dependent code necessary to perform IO operations on a given device is usually included in a device service routine for that device. The only device dependent code that is put into COMMON or COMMOD is if it must be conditionally assembled according to configuration parameters. The service routine itself is an independently assembled module which is included in the monitor for configurations which have the corresponding device, and left out otherwise. It does not contain any code which must be assembled with configuration dependent parameters such as how many devices of some type are present, or to which interrupt level it is to be assigned.

Device service routines normally include three components:

1. A device data block

2. A dispatch table, and the corresponding routines to perform UUO level operations.

3. An interrupt routine.

The following section contains a brief discussion of each of these components.

1. The device data block must contain the standard entries described in the monitor table description. These entries are used by device independent code, and therefore must be present for any device which the device independent code serves. In addition to the standard entries, the DDB may have as many additional entries as needed by the device dependent code.

   Since the service routine is to be assembled independently, we do not know what other devices will be present in the system. We therefore can not fill in the linkage to the "next" DDB. This linkage is supplied by the system initialization code according to information from INTTAB. Also, we may not know how many units will be present in the system. Hence we can not write a DDB for each unit. Rather we write a single DDB, and the additional DDB's are set up during system

initialization by copying this one and using sequential device numbers.

For devices which might have multiple controllers, and therefore controller dependent code, the DDB is coded in COMMON, rather than in the service routine. The controller dependent code is incorporated into the DDB, and DDB's are conditionally assembled for as many devices as specified by the configuration parameters.

2. The device dispatch table provides a standardized set of entry points for all UUO Level functions for this device. There are two possible formats, short dispatch table and long. The short table contains only the basic entries required of all service routines. If the device requires any additional functions, it must have the long dispatch table. The DVLNG bit in the DEVMOD word of the DDB specifies which format the corresponding dispatch table has. The base address of the dispatch table is contained in the DEVSER word of the DDB. A detailed description of the dispatch table is contained in the Monitor Table Desciptions. Most of the UUO level routines depend so much on the nature of the device which 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 conditions bits for the device will be initialized, and its priority interrupt level assignment will be cleared with a CONO instruction. The RELEASE routine might perform the same functions. The CLOSE routine does whatever is appropriate for 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 which is not the device dependent part of some 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 we decrement

the value of the timer for each active device, and if the expected interrupt occurs the field will be cleared. If the field is decremented to zero, the interrupt has not occurred within a reasonable amount of time. We then assume that the device is hung, and dispatch to the hung routine in the device service routine. This routine can attempt to recover from the hung condition, or can attempt simply to 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 nonskip 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 device.

The function of the input and output routines is to start the device. Normally this is done by executing a DATAI or DATAO instruction. For disk, however we usually only add a request for a transfer to an appropriate queue, and the actual transfer will be started at a later time. The IOACT bit is always set before returning to device independent code. The IOACT bit indicates that we are expecting an interrupt from this device. As long as IOACT remains set, the device independent code will 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.

The Interrupt Routine

When an standard interrupt occurs on channel N, control goes to location 40 + 2*N. Location 40 + 2*N will contain a JSR to location CH'N.

Example: At 40 + 2*4 would be the instruction

```
JSR  CH4
```

This instruction saves the PC and processor flags for the interrupted routine in location CH'N and executes the instruction at CH'N +1.

Since several devices may be assigned to each PI channel, out first task is to determine which device caused the interrupt. This is the function of the "interrupt skip chain." The instruction at CH'N +1 will be a JRST to the interrupt routine for some device assigned to Channel N. This routine begins with a CONSO which will skip if its device has an interrupt pending. If the CONSO does not skip, we execute the next instruction which will be a JRST to the interrupt routine for another device assigned to this channel. This will also be a CONSO followed by a JRST to another interrupt routine on the channel. The chain continues through the interrupt routines for all devices assigned to the channel. The last CONSO is followed by a JEN which dismisses the interrupt, instead of the normal JRST. When an interrupt occurs, control is passed from one interrupt routine to another. Each interrupt routine checks its device and passes control to the next--until we find the device which caused the interrupt. When we reach the routine for that device, the CONSO will skip, and begin execution of the interrupt routine. In case of a spurious interrupt--when, no device is found, with an interrupt pending--the interrupt is dismissed by the JEN at the end of the chain.

The interrupt skip chain can not be coded directly in the device service routine. This is because when writing a given service routine, we don't know what other devices will be on the same PI channel. The instruction following the CONSO is normally something like

JRST .

At system initialization these instructions are replaced by the correct ones, according to information found in INTTAB.

The skip chain is not necessary for the newer peripherals on the KI-10 and KL-10. These devices are able to supply their own interrupt addresses, thus giving control directly to the correct routine, this is called a "Vectored Interrupt".

There are several functions which must be performed by every interrupt routine. If the interrupt routine is to use any accumulators, it must save their contents and later restore them. A routine is provided in COMMON to perform this function for each interrupt routine which requires it. Since an interrupt routine can be interrupted only by a higher priority interrupt, we only need one save routine and save area per PI channel. The routine for channel N will be assembled, if needed, with the label SAV'N. Since we don't know which channel a given device will use, we can not use this label directly in the service routine. Instead we use the label XYZ'SAV for each device XYZ and its PI channel N. In addition to saving the accumulators, the save routine sets up a push down list with the address of another routine which restores the accumulators, and dismisses the interrupt. This allows the interrupt routine to execute an unmatched POPJ to restore accumulators and dismiss the interrupt.

Most devices can cause interrupts due to error conditions as well as successful completion of a transfer. The interrupt routine must therefore check the conditions bits from the device to determine what caused the interrupt. If a hardware error has occured, the routine must take some appropriate action, possibly attempting to recover from the error.

Interrupt routines must always check for reaching the end of the current buffer. If we have reached the end of a buffer, a device independent routine in UUOCON must be called to advance the pointer to the next buffer. ADVBFE advances the output pointer, DEVOAD, and clears the use bit in the buffer just finished. ADVBFF advances the input pointer DEVIAD and sets the use bit in the buffer just filled. Both routines check if the next buffer is available to the interrupt routine. They also check if IO is to be stopped so that this job may be swapped or shuffled, if the user wants only one buffer at a time, if the user has typed ^C, or if the job is being locked in core. If, for any of these reasons, the interrupt routine must not continue processing out the next buffer, the buffer advance routine gives it a nonskip return. If the interrupt routine may continue processing, it is given a skip return.

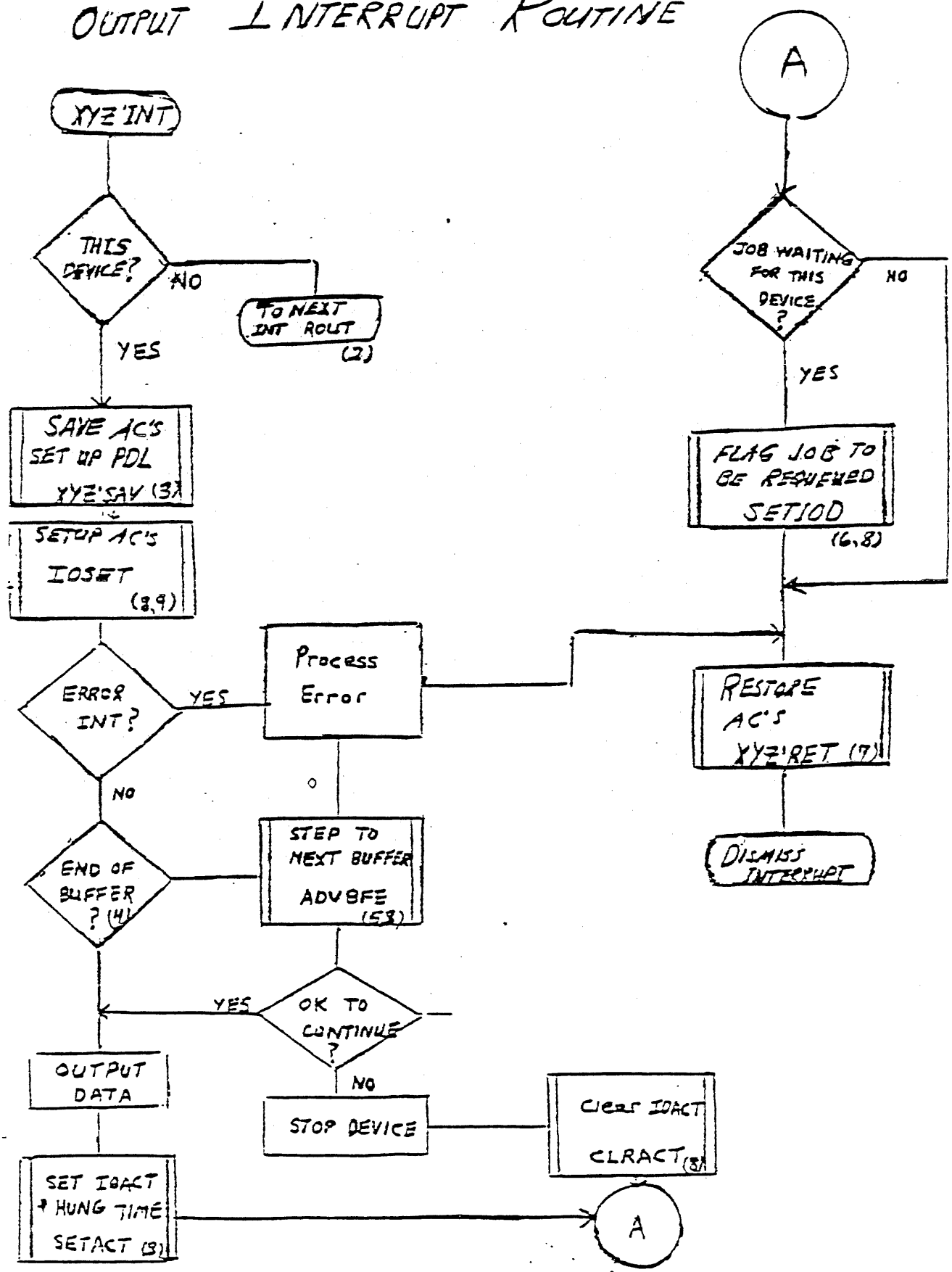
If we may continue into the next buffer, we issue the next IO instruction. Otherwise we "turn off" the device. This

normally means clearing its PI assignment. If we turn off the device, we clear the IOACT bit in DEVIOS to indicate that no more data will be transferred unless the device dependent routine is called again at UUO level to start the device.

Another function which every interrupt routine must perform upon reaching the end of a buffer is checking if the job is in IO Wait. If, the IOW bit is set when we finish a buffer, we call the device independent routine SETIOD to get the job out of IO Wait. If the job is in TIWQ, SETIOD changes the job's wait state code to TSQ, indicating to the scheduler that the job should be removed from IO Wait. It sets the JRQ bit as a flag that this job needs to be requeued by the scheduler. Then, unless the null jub was running when the interrupt occured, it returns to the calling routine. For all other I/O wait states, SETIOD changes the WSC to RNQ (Ø) directly. If the null job was running, we do not want to return to it upon dismissing the interrupt, because there is now a user job to run. In this case, SETIOD calls a routine to stop the current job. This routine requests a scheduler cycle (PI level 7 interrupt). Hence as soon as the IO interrupt which we are handling is dismissed, there will be a partical scheduler cycle. The scheduler will be called and will choose the job which just came out of IO Wait to be run.

On the next page is a flow chart showing the general functions performed by any interrupt routine.

OUTPUT INTERRUPT ROUTINE



Notes on Output Interrupt Routine

1. This is normally a CONSO instruction.

2. This exit, to next interrupt routine, is set up by the system initialization routine.

3. Routine is in COMMON. The label XYZ'SAV will be equated to the correct address.

4. Some devices will not have this step because they only get interrupts at the end of a buffer.

5. ADVBFE will advance DEVOAD, the current buffer for the interrupt routine, to the next buffer of the ring. The use bit will be cleared on the buffer just finished, making it available to the program. ADVBFE will give the "OK-TO-CONTINUE" return if all the following conditions are true:

   1. The next buffer is full. (Use bit set.)

   2. Scheduler is not trying to stop job. (SHF, CMWB, CNTRLC, and LOK not set.)

   3. User did not ask for "Synchronous" IO (IOCON bit in DEVIOS not set.)

6. Set job's wait state code to Wait Satisfied and set JRQ bit.

7. Routine in COMMON. Its address is put on PDL by XYZ'SAV, allowing the service routine to get there with an unmatched POPJ.

8. Device independent routine in UUOCON.

9. This routine sets up AC's F, J, and S according to the DDB whose address is given in AC F.

PI Channel Assignment

By convention all device service routines are written so that they may operate on any PI channel, without having to be reassembled. This convention is necessary if we are to avoid assembling service routines with configuration parameters, since a given device may operate on different channels in different systems. All requirements for data dependent on the PI channel or the hardware configuration are met by macros defined in COMMON. Since COMMON is assembled for each installation, with configuration parameters, it is a convenient place for the configuration dependent code needed by service routines. The need for configuration dependent code is handled in several ways, including INTERAL symbols for reference by the service routines and the INTTAB table used at system initialization time. Internal symbols in COMMON allow a service routine to set up external symbols for its own PI channel number, the locaton where the PC is stored on the interrupt, and a routine to save and restore AC's. The INTTAB entry for each device is used to link the interrupt routine for that device into the interrupt skip chain, to link its DDB into the system's DDB list, and, if necessary, to set up a copy of the DDB for each unit of that device. Defining these symbols and setting up the INTTAB entry for each device is the function of the "channel assignment" macros in COMMON.

The channel assignment macros are nested several deep, and combined in different combinations depending on the requirements of the device. One of the simplest of these is the ASGINT, which is used for a device having an interrupt routine, but no DDB and needing no AC save routine. The software clock interrupt is an example of such a routine.

Example: ASGINT CKØ, 7

This macro sets up the definitions

```
CKØCHN == 7
CKØCHL == CH7
```

and generates the INTTAB entry

```
XWD  7, CKØINT
XWD  Ø, Ø
```

Devices which have DDB's and need AC Save Routines require a more complicated sequence of assignment macros. One such sequence is ASGSAV, ASGSV1, and ASGSV2. This sequence is used for single devices (controllers) which have DDB's and save routines. We shall examine the functions of these macros, beginning with the innermost and working out.

ASGSV2 is used to cause an AC save routine to be set up for the specified channel. Also it equates the labels used (by convention) in the service routine to the labels which will be set up in the save routine.

Example ASGSV2 LPØ, 4

This generates the definition

USED4 == 1

which will later cause a CHAN macro to be assembled, generating an AC save routine for PI channel 4.

In addition it defines the following symbols:

LPØSAV == SAV4	Address of save routine
LPØRET == RET4	Address of AC restore routine
LPØCHL == CH4	Place where PC is stored on channel 4 interrupt
LPØSAV == SVAC4	Location where AC's are saved
LPØPDP == C4PDP	Word used to initialize push down pointer
LPØJEN == C4JEN	Address of instruction which dismisses the channel 4 interrupt.

All symbols are declared as INTERNAL to COMMON. Many service routines will only use the first of these symbols. The restore routine is normally reached by an unmatched POPJ from the device interrupt routine.

ASGSV1 sets up the INTTAB entry and defines a symbol by which the service routine can refer to its own PI channel number. For all standard devices ASGSV1 also calls ASGSV2 to generate a channel AC save routine.

```
Example  ASGSV1  PTR, 4

This generates:  PTRCHN == 4
                 XWD  1004, PTRINT
                 XWD   0,   PTRDDB
                 ASGSV2  PTR, 4
```

ASGSAV calls ASGSV1 to set up an INTTAB entry and generate a channel AC save routine. Its only other function is to declare the DDB and interrupt addresses EXTERNAL on Pass 2, if they are not defined in COMMON.

For multiple devices, such as the line printer, there must be a separate INTTAB entry for each device. These entries are set up by the MULASG macro, which repeatedly calls the DEVASG macro.

DEVASG is written with four arguments.

```
DEVASG DE,X,PI,DSKFL
```

DE will be a two letter device mnemonic. Sequential values for X are used for the devices. Hence if there are two line printers, they will be LP0 and LP1. PI is an argument specifying the PI channel. DSKFL is a "disk flag." It is set to a non zero value for disk in Level D disk service monitors (i.e. 5.xx and later monitors.)

The DEVASG macro defines one symbol, and then does an ASGSV1 for the device specified by its first two arguments.

```
Example:  DEVASG  LP, 0, 4, 0

This will generate:

          LP0N == 1
          ASGSV1  LP0, 4
```

The MULASG macro simply repeats the DEVASG macro for each of the devices of the given type present in the configuration, incrementing the value of the second argument.

```
Example  MULASG  LPT, LP, 4, 0
```

If LPTN has the value 2, this will have the effect of generating:

```
DEVASG  LP, 0, 4, 0
DEVASG  LP, 1, 4, 0
```

Specification of PI Level

Devices are divided into groups for the purpose of assigning PI channels. All devices in a group will be assigned the same channel number. If any devices of a group are present in a given configuration, we step to the next channel for the next group. If it turns out that no devices of a group are present, the channel number is not advanced for the next group. Advancing the channel number is the function of the NEXTCQ macro.

If any of the devices of the preceeding group were present in the configuration, the symbol .CHAS will be non zero. This symbol determines whether or not NEXTCQ will increment the value of .CH, the symbol used for all PI arguments. The process of advancing .CH is complicated by the fact that we might want to reserve one or more channels for use by special devices. If the symbol UNIQ'N, where N=1-7, is defined and equal to a non zero value, no standard devices will be assigned to channel N.

The NEXTCQ macro does nothing if the symbol .CHAS is equal to zero, indicating that no devices in the previous group were present in the given configuration. If .CHAS is non zero, NEXTCQ calls the NEXTCH macro. NEXTCH then calls the NEXTCU macro, which has the function of skipping over a channel if the corresponding symbol, UNIQ'N is non zero. If UNIQ'N is non zero, NEXTCU calls the NEXTCH macro, which increments the value of .CH and calls NEXTCU again for the new value. The sequence is repeated until a channel is reached for which UNIQ'N is not defined or equal to zero.

Channel AC Save Routines

For each service routine which needs to save AC's there will be an ASGSV2 macro. An AC save routine will be generated for each channel having any service routines that need to save AC's. Note that we do not need a separate save routine for each device, since a device can not interrupt another device having the same PI level.

Channel AC save routines are generated by the CHAN macro, with an argument to designate the PI level. The CHAN macro

for channel N is conditionally assembled according to the value of the symbol USED'N. USED'N is set to 1 by an ASGSV2 macro with a second argument of N. Hence a save routine will be generated for each device for which ASGSV2 was called.

In addition to generating the save routine, the CHAN macro also sets up the location CH'N which begins the interrupt skip chain. For those channels with USED'N equal to zero, a NULL macro is called to generate only the CH'N location and no save routine.

Example: The following code would be generated if USED5 is non zero. HIGHAC is defined as the highest AC to be saved by this routine. PDL is a symbol which specifies the length of the push down list to set up.

```
CH5:    Ø                   Initial skip chain
        JEN@CH5             This word replaced during
                .           system initialization

SAV5:   Ø                   Called with JSR
        MOVEM   HIGHAC, SVAC5 + HIGHAC      Save highest AC
        MOVEI   HIGHAC, SVAC5               Set up BLT pointer
        BLT     HIGHAC, SVAC5 + HIGHAC-1    Save other AC's
        MOVE    P, C5PDP                    Set up push down
                                            pointer
        JRST    @SAV5                       Return to caller

Return here on POPJ from calling routine

RET5:   MOVSI   HIGHAC, SVAC5               Set up BLT pointer
        BLT     HIGHAC, HIGHAC              Restore AC's
CSJEN:  JEN     @CH5                        Dismiss interrupt

SVAC5:  BLOCK   HIGHAC                      Place to save AC's
C5PDP:  XWD     -PDL+1, .+1                 Initial push down
                                            pointer
C5PDP1: EXP     RET5                        First word on push
                                            down list
                                            -Set so that POPJ
                                            returns to RET5
        BLOCK PDL-1                         Rest of push down
                                            list
```

Timing and Interlock Comsiderations

Whenever there is a possibility of interactions between separate asynchronous processes, we must take special care to ensure that the interaction takes place correctly and that the process do not interfere with each other. This problem can occur with separate user jobs executing a UUO which can not immediately run to completion. It can also occur when there is a device active at the same time a UUO is being processed for a job. In this section we examine several of the problems that occur, and techniques for overcoming them.

Starting a Device

In order to guard against unsolicited interrupts from a device, we set up in core the bits to test in checking if that device caused the interrupt. When we do not have the device "turned on" those bits are cleared, making it impossible to recognize an interrupt from that device. When the device is started, the bits are set up. In turning on the device, we must be careful that we do not get the interrupt before we set the bits indicating that we were expecting an interrupt. Essentially "turning on the device" and "setting the bits which indicate that it is on" must appear to be simultaneous actions. We create this effect by turning off the PI system while we start the device and set the bits. The STARTDV macro may be used for this purpose.

Before calling the STARTDV macro, we set the bits to be tested on interrupts in the left half of AC T1. The condition bits to be sent to the device are put in the right half. The STARTDV macro is then written with the device mnemonic as an argument.

Example for PTP

```
        HRLI   T1, PTPDON          Bit to test an interrupt
        HRRI   T1, PTPCHN          PI Assignment
        TRO    T1, PTPDON          CONO bit to start device
        STARTDV PTP
```

The STARTDV generates:

```
        EXTERNAL PIOFF, PION
        CONO     PI, PIOFF         Disable interrupts
        CONO     PTP, (T1)         Start PTP
        HLRM     T1, PTPCON        Store bit to test
```

```
CONO    PI, PION        Enable interrupts
                        (Expect immediate interrupt)
```

The interrupt routine begins with:

```
CONSO   PTP, @PTPCON    Did PTP cause interrupt?
JRST    Next Interrupt Routine;  No, check next device
```

If you use the STARTDV macro, you must set up location XXX'CON in the service routine. Also you must ensure that the device does not cause a normal interrupt while you have XXXCON set to zero. If a condition which causes an interrupt occurs while XXX'CON is cleared, the interrupt will be dismissed as a spurious interrupt, by the JEN at the end of the skip chain. However, since the condition has not been cleared, the device will immediately cause another interrupt, and the system will be hung in an infinite loop. Normally the device initialization routine should clear the PI assignment in the hardware, as well as clearing XXX'CON.

Race Conditions

Whenever the final result of two asynchronous processes depends on which of them finishes first, a race condition is said to exist. Since race conditions lead to unpredictable results, they must always be avoided. We are in jeopardy of a race condition any time there is a possible interaction between an interrupt routine and a lower priority routine. Race conditions are also possible when a process may be interrupted for any reason, and the same code then be executed for another process before being finished for the first. There are basically two ways of avoiding race conditions. One is to set up interlocks to control the order in which events occur in the separate processes. The other is to arrange things so that the results are the same regardless of the order in which the events occur.

As an example, consider the following problem: the use bit in buffers provides synchronization between UUO level routines and interrupt routines. On output, if the use bit is set it indicates that there is at least some data in the buffer, which still needs to be written at interrupt level. The IOACT bit and IOW bit are also significant. IOACT indicates that we are expecting an interrupt from the device, and hence the UUO level routine need not call the OUTPUT routine to start the device. The IOW bit indicates

that the job is in IO Wait, and the interrupt routine should call SETIOD upon completion of a buffer. There is a potential problem if we are about to put the job into IO Wait at the time we get the interrupt completing a buffer. We check the next buffer's use bit, and finding it set we set the IOW bit. The IOW bit is the flag to the interrupt routine that the job should be taken out of IO Wait. What if the interrupt which completes the buffer occurs after we find the use bit set, but before we set the IOW bit? We will put the job into IO Wait, and the event which should take it out of IO Wait has already occurred. It turns out that this is a case where we arrange to do the right things, even when the events occur in the "wrong" order.

There are two possible cases, depending on whether or not there is another full buffer. If there is another full buffer, the interrupt routing continues writing from it. Upon completion of that buffer it will detect the IOW bit and call SETIOD to get the job out of IO Wait. If there is not another full buffer, the problem is somewhat more complicated. The interrupt routine will turn off the device and clear IOACT, since it can not continue in the next buffer. Since IOW is not yet set, it will not call SETIOD. Back in the UUO routine, we will attempt to put the job into IO Wait by calling WSYNC. WYSNC normally sets the job's Wait State Code to IO Wait and exits to WSCHED for rescheduling. If it did this in the case we have just described, the job would be put into IO Wait, and would never be taken out of that state. To avoid this problem WSYNC will never put a job into IO Wait if the IOACT bit is not set. Rather, it will give an immediate return to its caller. Upon return from WSYNC, we check the next use bit again. This time we find the buffer available, and allow the job to continue. Hence, even if the interrupt occurs after we test the use bit but before we call WSYNC, the final result is the same as if the interrupt had occurred before we tested the use bit.

Suppose in the above example the interrupt had occurred during WSYNC--after we checked IOACT and found it set, but before we set the IOW bit. Here again, the interrupt routine would turn off the device, if it did not have another buffer, without getting the job ot of IO Wait. Again the job would be hung. To avoid this problem, WSYNC turns off the PI system before it checks IOACT and back on after setting IOW. Hence the two instructions appear

simultaneous to the interrupt routine. This is an example of preventing a race condition by means of an interlock to control the order in which certain events occur.

INTERRUPT ROUTINE CHAIN

```
40 + 2N:        JSR    CH'N
                JSR      PIERR


CH'N:           0
                JRST   DEV1'INT


DEV1'INT:       CONSO DEV1, Conditions
                JRST  DEV2'INT

                Process DEV1 Interrupt


DEV2'INT:       CONSO DEV2, Conditions
                JRST  DEV3'INT

                Process DEV2 Interrupt


DEV3'INT:       CONSO DEV3, Conditions
                JEN   @CH'N

                Process DEV3 Interrupt
```

This page is for notes

EXERCISES

1. What is the basic function of the OUTPUT routine in a device service routine?

*2. In general, where is an output device turned off if it has emptied the last buffer supplied by the user? Where is the specific instruction for the paper tape punch?

*3. What is the INTTAB entry for the paper tape reader? -- the software clock interrupt?

4. If an interrupt occurs, but no device assigned to the channel admits causing it, how is the interrupt dismissed?

5. Which interrupt assignment macros generate (directly) INTTAB entries? -- device save routine definitions?

*6. Which instruction results in CKØCHL being defined? How is it defined?

7. What would be the result of defining UNIQ3 = 1?

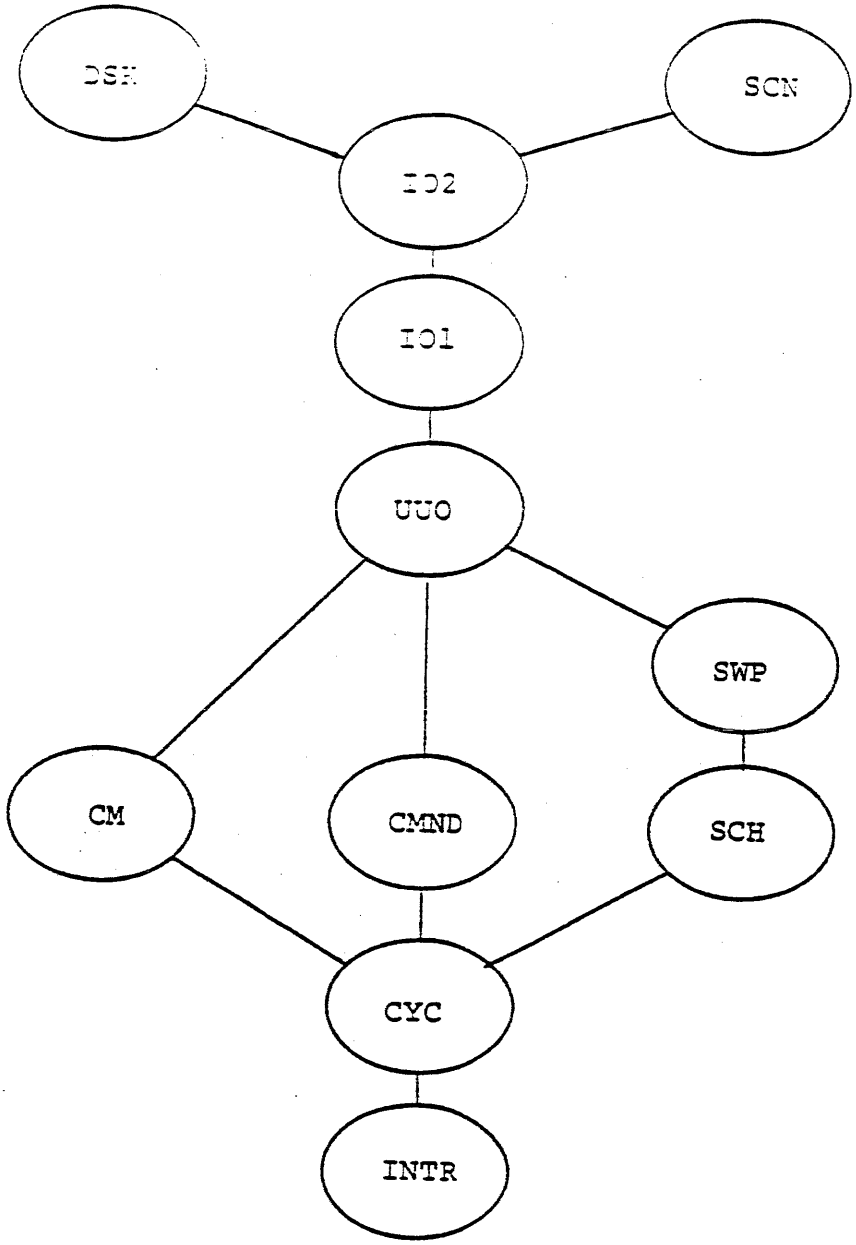
8. How could you ensure that a special device is assigned as the only device on a specific channel, without making any changes to COMMON.MAC?

9. List the actions taken by PTPINT if a buffer has been finished and the next buffer is not available to it.

*1Ø. Where do the CONSO skip chains get initialized?

TOPS-10 MONITOR INTERNALS

Disk Service



COURSE MAP

This page is for notes

DISK SERVICE

INTRODUCTION

The disk service routine is the most complicated of all the service routines because it is heavily used and it has a complicated incore data base. This module will explain how jobs compete for use of the disk, how that competition is resolved by the queue mechanisn and how I/O is done within the framework of the file structure.

This page is for notes

RESOURCES

TOPS-10 Microfiche

TOPS-10 Monitor Internals Course Supplement

Course Materials - Chapter 10
Supplement - Graphics Section 10-1 thru 10-16

LECTURE OUTLINE

I. Introduction
   A. FILSER
   B. COMMOD
   C. ???KON

II. Hardware Principles
   A. Fixed Head
   B. Movable Head
   C. Configuration
   D. Channel Command List
   E. I/O States

III. Disk Resident Data Base
   A. STR
   B. MFD
   C. UFD
   D. RIB
   E. FILE

IV. Files And Structures
   A. DSKDDB
   B. Physical Disk Block Usage
   C. Home Block
   D. TABSTR
   E. STR Data Block
   F. UDB

V. Space Management
   A. CLUSTER
   B. SAT
   C. SPT
   D. SAB

DISK SERVICE

All device dependent functions for disk files are performed by a group of modules collectively known as the disk service. The disk service performs two different and logically independent types of functions; IO operations and file operations. IO operations are the reading/writing of specific blocks on specific units. File operations involve the processing of directories, pointers, etc. The file processing software accepts requests stated in terms of file names, file structures, and relative blocks within a file. From such requests it sets up requests for operations on specific blocks which can then be handled by the IO software. The file processor itself must frequently call upon the IO processor to read and write various special disk blocks.

The disk software includes several modules which are assembled separately and included in the monitor, as needed, at load time. Most of the executable code is included in FILSER. FILSER performs all operations which are independent of the controller type, and will be present in any disk monitor. There are separate routines to handle controller dependent functions on each controller. These routines, RPXKON for RH10/20 disk controllers, DPXKON for RP10 disk controllers and FHXKON for RC10 fixed head controllers, are loaded only if needed. SWPSER creates IO requests for the swapper and interfaces with FILSER. The data base for the disk service is contained in a separate module named COMMOD. COMMOD is analogous to COMMON for the rest of the monitor.

Hardware Principles

Each disk unit is connected to a controller, and all communications to that unit must go through the controller. The controller is connected to the CPU by the IO bus. The controller is also connnected to a data channel, which allows the controller to access memory without going through the CPU or the IO bus. 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 (i.e. track number, etc.) and the core address of a list of core areas. The

transfer will be to or from consecutive locations on the disk, but may be scattered among an arbitrary number core areas. The length and address of each of these core areas is 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 may be started, the unit, the controller, and the data channel must all be idle. All will be busy until the transfer is complete. The CPU, however, is needed only for the time required to send the instruction to the controller. If can then go on processing while the transfer takes place. When the transfer is complete, the controller will cause a priority interrupt on its assigned channel.

Before a transfer can be started on a disk pack, the access arms must be positioned to the correct track. A positioning operation requires the unit and controller to be idle (RP10 only), but not the data channel. The unit will be busy until it reaches the designated track, but the controller is almost immediately available for other operations. When the unit reaches position, it will inform the controller. The controller will cause 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 will be no interrupt, but an attention bit for the unit will be 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 that RH10/20 controllers can initiate additional position requests to other idle units even-though the controller itself is transferring data).

The basic addressable unit of disk storage is a sector. The size of a sector is different for various types of disk, but there are always an even number of sectors per track. It is sometimes useful to know which sector (of each track currently accessable) 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 a DATAI instruction to the controller. The unit must have been previously selected by a DATAO or CONO.

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 the project-programmer number of its owner. The program may read or write the blocks of a file either sequentially or randomly. 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 or 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 RIB's 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. However, a group could possibly consist of only one block.

Directories

The locations of all files belonging to one user are found in a User File Directory for that user. The User File Directory, or UFD, is itself a file with a 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 a two word entry corresponding to each file belonging to that user. Each entry specifies a file name and extension, and contains a pointer to the RIB of that file.

All the UFD's belong to an "artificial user" with project-programmer number [1,1], and 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 or MFD. The collection of files consisting of an MFD, all the UFD's to which it points, and all the user files to which the UFD's point - - is called a file structure.

File Structures

As a collection of files, the file structure is logically independent of any hardware considerations, such as units, controllers, etc. 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 units, and a single file may be spread over several units of the structure. But different type units, those with different models of controllers, can not be combined in one structure. Hence RP04 and RP06 disk packs may be combined, and RM10B drums and RD10 fixed head disks may be combined. But the same structure will never include both disk packs and fixed head devices.

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 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 a structure is removed from the system, the entire structure becomes inaccessable. 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. Therefore, if an entire unit is to be used only for swapping, that unit need not be included in any structure.

Allocation of Disk Space

Before a block may 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 is a Storage Allocation Table, or SAT, block which has 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 being set will prevent 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 possibly be more clusters than can be accounted for with a single SAT block. In this case, there will be as many separate blocks of SAT's as necessary, and each SAT block will be physically near the blocks which it describes. All the SAT blocks for a file structure are combined into a file called SAT.SYS. SAT.SYS 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 a table called a Storage Allocation Pointer Table, or 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 which is needed.

When a SAT block is in core, it resides in a Storage Allocation Block, or SAB. All the SAB's for a unit are linked together and to the SPT. 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 during system initialization without the file structure being refreshed.

Allocation of disk space is done in two different ways. If a user is writing a file and reaches the end of the space previously allocated, additional space will be allocated at that time. If possible, the space will be 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 will be 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.

I/O Processing

The disk I/O processing software maintains information about each piece of disk hardware in tables found in COMMOD. These tables include the Unit Data Block (UDB), Controller Data Block (KON), and Channel Data Block (CHN). The IO 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 or DDB, has all the standard features of any DDB, plus a great deal of additional information unique to disk. Disk DDB's are set up dynamically as INIT UUO's, and ASSIGN commands which give a logical name to disk, are done. There is, therefore, a DDB for each user software channel which is doing disk I/O. In addition, on non-VM systems there is a DDB for use by the swapper. Every disk transfer which is done is the result of a request being set up in a disk DDB and presented to the IO processor. This includes reading and writing of user files, non-VM swapping transfers, and all transfers done by the monitor for its own purposes.

Request Queues

When an I/O request is presented to the I/O processor, the transfer or positioning will be started immediately if all the necessary devices are available. Usually, 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, or PW, Queue for that unit. If the request does not require positioning, it is added to the Transfer Wait, or TW, Queue for the Data Channel. The queues are formed simply by linking together the DDB's beginning with the Unit Data Block for a PW Queue or the Channel Data Block for a TW Queue.

Each time there is a disk interrupt, each unit which 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. The task of choosing which request to process next is the function of two optimization routines.

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 we mean by "best" is somewhat difficult, but there are two basic considerations. First of all we attempt to minimize the time that each unit is not doing data transfers. In addition, we try not to be grossly unfair to any individual request. We do not want to delay one request indefinately in favor of requests which can be processed more efficiently. Therefore, each optimization routine is written to choose the request which has been waiting the longest every so often. "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 which 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 which may be specified when the monitor is built.

This page is for notes

EXERCISES

Hardware Principles

1. To the hardware, what is the basic addressable unit of disk storage?

2. Under what conditions will a disk controller be "busy" (i.e. unable to accept a command)?

3. What events cause a disk controller to interrupt the CPU?

4. How can a program determine which sector of a given unit will be next available for access?

Structure of Files

5. Explain the relationships between blocks, clusters, groups, and superclusters.

6. How are blocks of disk storage linked together to form a file?

7. What is a "file structure?"

8. What is the purpose of directories?

9. What factors limit the length of a disk file?----the number of files which one user can own?

10. How does the structure of a file written by random access output differ from that of a file written sequentially?

I/O Processing

11. Under what conditions would a positioning operation be started immediately upon execution of a UUO?----a transfer operation?

12. What does the IOACT bit mean in a disk DDB?

13. How many requests can there be at any one time in the:

    a. TW Queue for a fixed head disk?

    b. TW Queue for a disk pack?

    c. PW Queue for a disk pack?

14. What determines the number of blocks read or written on a single transfer?

15. After a transfer has been completed for a user who is reading a file sequentially, when will the next request be set up?

16. State, in your own words, the rules used to select the next request to be processed, from each type of queue.

Allocation

17. What are the advantages of a file being written as a single group?

18. How can a user ensure that a file is written as a single group.

19. What are the relative advantages and disadvantages of a larger cluster size?

20. What happens to space which is allocated for file, but has not been used when the file is closed?

21. What factors should influence the choice of "minimum amount to allocate" for disk files?

File Operations

22. List three ways a STR may be put on a job's search list.

23. When a LOOKUP is done, what determines the STR(S) where the disk service will look for the file?

24. When are disk DDB's set up?-----deleted?

25. When are directories

    a. Created?

    b. Updated?

    c. Deleted?

26. Explain the differences between cr eating, updating and superseding a disk file.

Miscellaneous

27. What are the effects of a disk unit going off line?

28. What factors should be considered in deciding on the configuration of units into file structures?

29. If a program which is writing a disk file fails to run to completion, what happens to the file? What happens to the old version, if the program is superseding a file?

30. How can blocks be marked as in use in a SAT, but not be included in any file?

This page is for notes

TOPS-10 MONITOR INTERNALS

Terminal Scanner Service

MODULE OUTLINE

TERMINAL SCANNER SERVICE

I. Scope

II. Data Structures

- A. Line Information
  1. Line Data Blocks (LDBs)
  2. LINTAB
- B. Job Information
  1. TTY DDBs
  2. JDA
  3. TTYTAB
- C. Job vs. Line Information
- D. Chunks

III. Terminal I/O

- A. Output
- B. Input

IV. Data Structures Revisited

V. Control Character Handling

- A. CONTROL/O
- B. CONTROL/C
- C. CONTROL/D
- D. CONTROL/T
- E. CONTROL/R
- F. CONTROL/Q and CONTROL/S

VI. Terminal Image Modes

A. Image Mode
B. Packed Image Mode (PIM)
C. Half Duplex Terminals

VII. Other Terminal Monitor Calls

VIII. Pseudo-terminals (PTYs)

IX. Macro Interpreted Commands (MIC)

SCOPE

All device dependent functions for teletypes are performed by the scanner service comprising SCNSER and an additional module depending on the type of scanner. This module contains the actual I/O instructions, 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.

This module deals with terminal communications at the local level. Local terminals are connected to a PDP-10 via DC10s, DC76s, PDP-11s and DL10s (DN87), or PDP-11s and DTEs (DN87S or DN20). TOPS-10 must handle special characters and echoing for these terminals. The specifics of the device drivers will not be investigated in detail; the main focus is on the SCNSER module.

Remote terminals are those on a TOPS-10 network. The node will handle echoing and most special characters, not TOPS-10. They are discussed in detail in the TOPS-10 Data Communications course.

DATA STRUCTURES

Line Information

LINE DATA BLOCKS (LDBs)

LDBs contain information about a terminal line. There is one LDB for each terminal and it is built when the monitor is initialized. LDBs are not dynamically created; when they are setup they are around forever. The reason for this choice is simple. Even though a job may not be logged in on a terminal, users may still type on that line. To speed response, the LDBs already exist which allows the monitor to avoid having to spend the time to allocate an LDB. The code to allocate and initialize the LDBs is in COMMON where it is discarded when system initialization is complete.

For a complete description of an LDB, see the monitor tables. However, it is important to remember thaat the LDB contains

1. Pointers to input and output chunks
2. Line status bits
3. Line characteristic bits
4. Horizontal position counter
5. MIC information
6. Break characters for Packed Image Mode
7. Count of characters to echo

LINTAB

The LINTAB table is used to locate the LDB entry for a line. It contains one entry for each terminal in the system (including CTY and PTYs). Refer to the monitor tables for a full description of each entry including bit descriptions. The bits in LINTAB are used to initialize the LDBs and cannot be changed.

Job Information

TERMINAL DEVICE DATA BLOCKS

Terminal Device Data Blocks (DDBs) are created dynamically as jobs login and contain terminal information that relate to the job. This includes:

1. Pointers to user buffers
2. Device and logical names for the terminal
3. I/O status information (DEVSTA)
4. Device mode information (DEVMOD)
5. CPU number of the CPU that owns this terminal
6. A pointer to the LDB.

There are two tables that contain pointers to TTY DDBs: TTYTAB and JDA.

TTYTAB

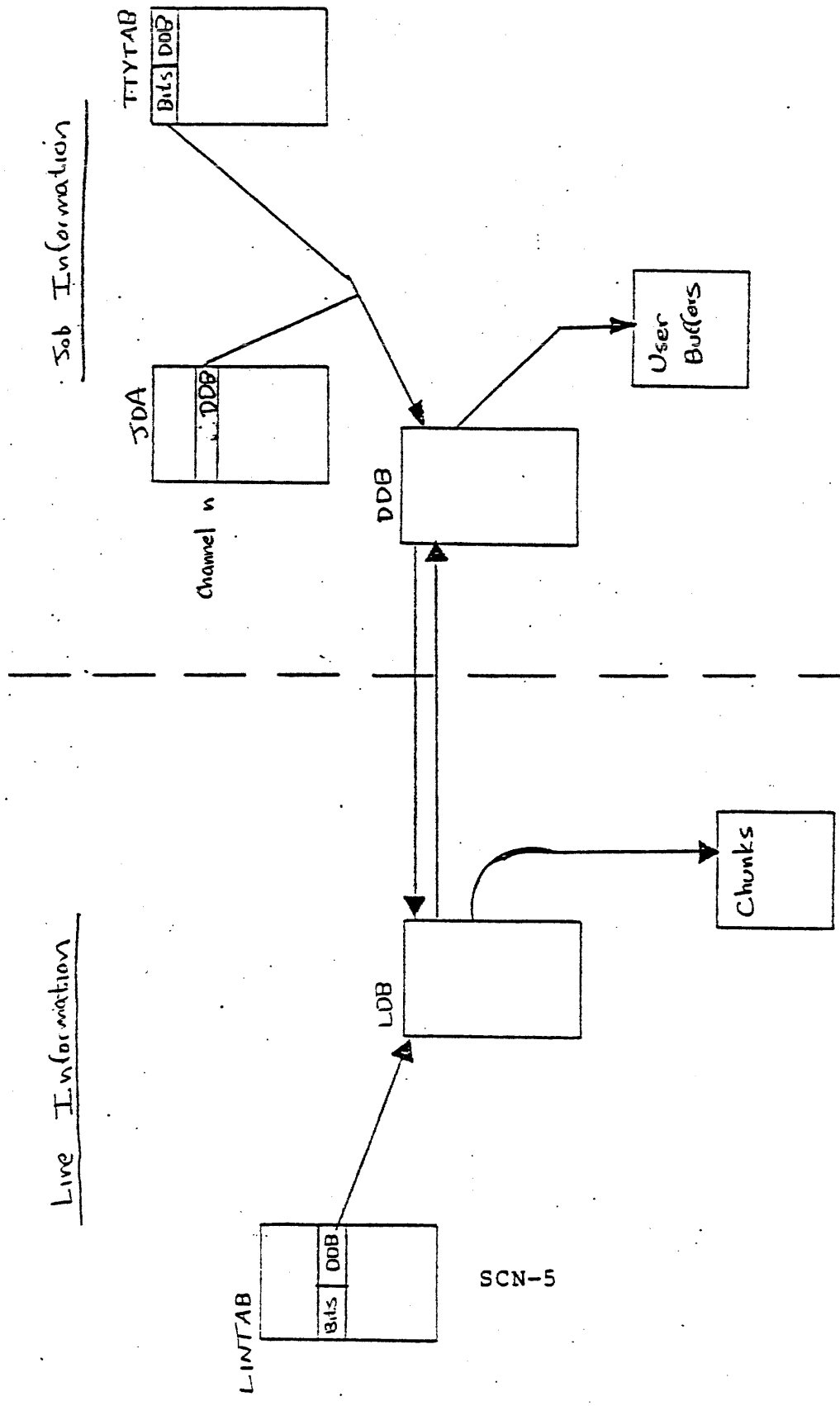
TTYTAB is a table in COMMON that has one entry per job and points to the DDB of the controlling (attached) terminal of the job. A zero entry indicates no attached terminal.

JDA

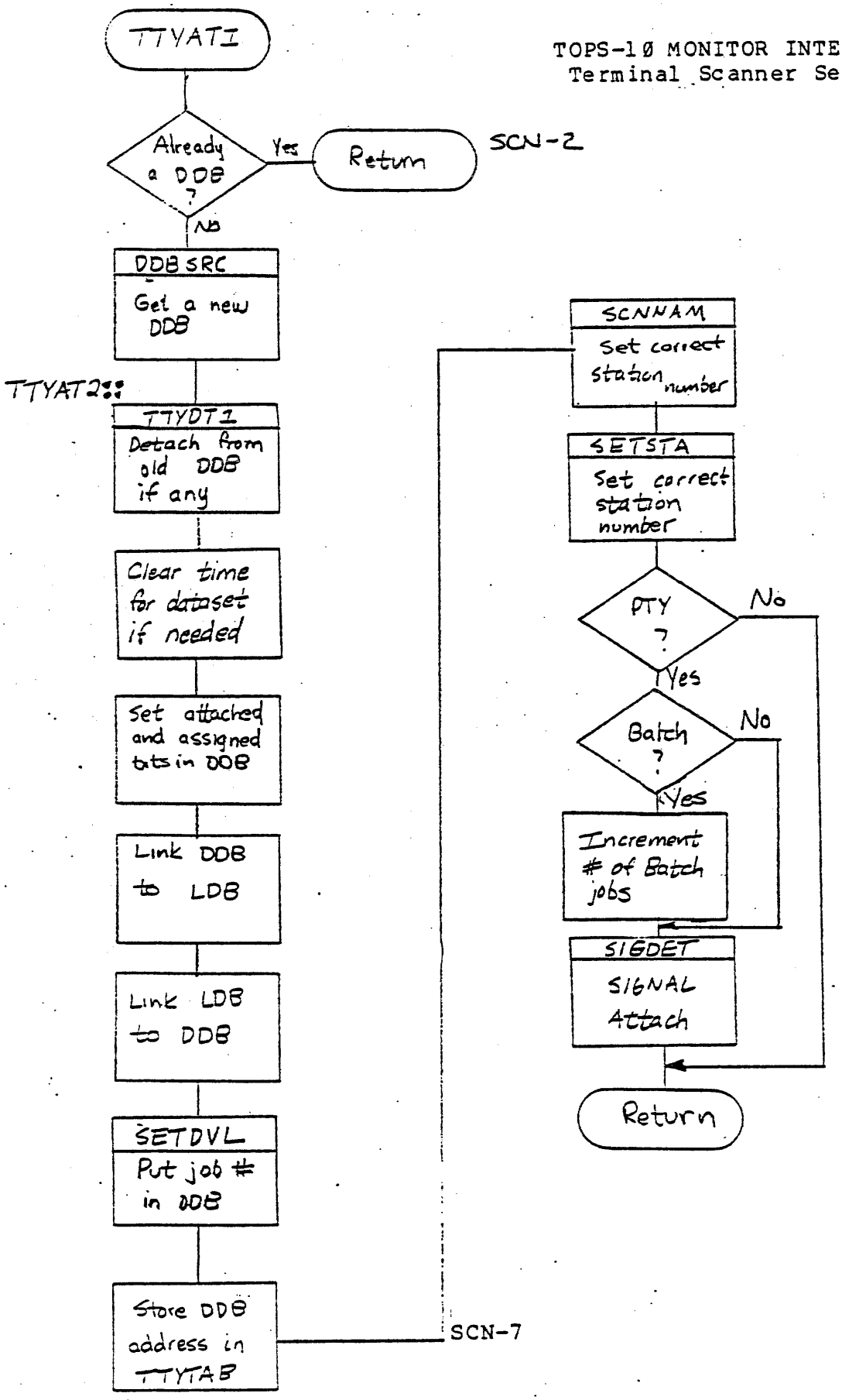
The Job Device Assignment (JDA) table is part of each job's UPMP. It has one entry per channel number and points to the DDB that is associated with a channel (RH). The LH describes which UUOs have been done on that channel.

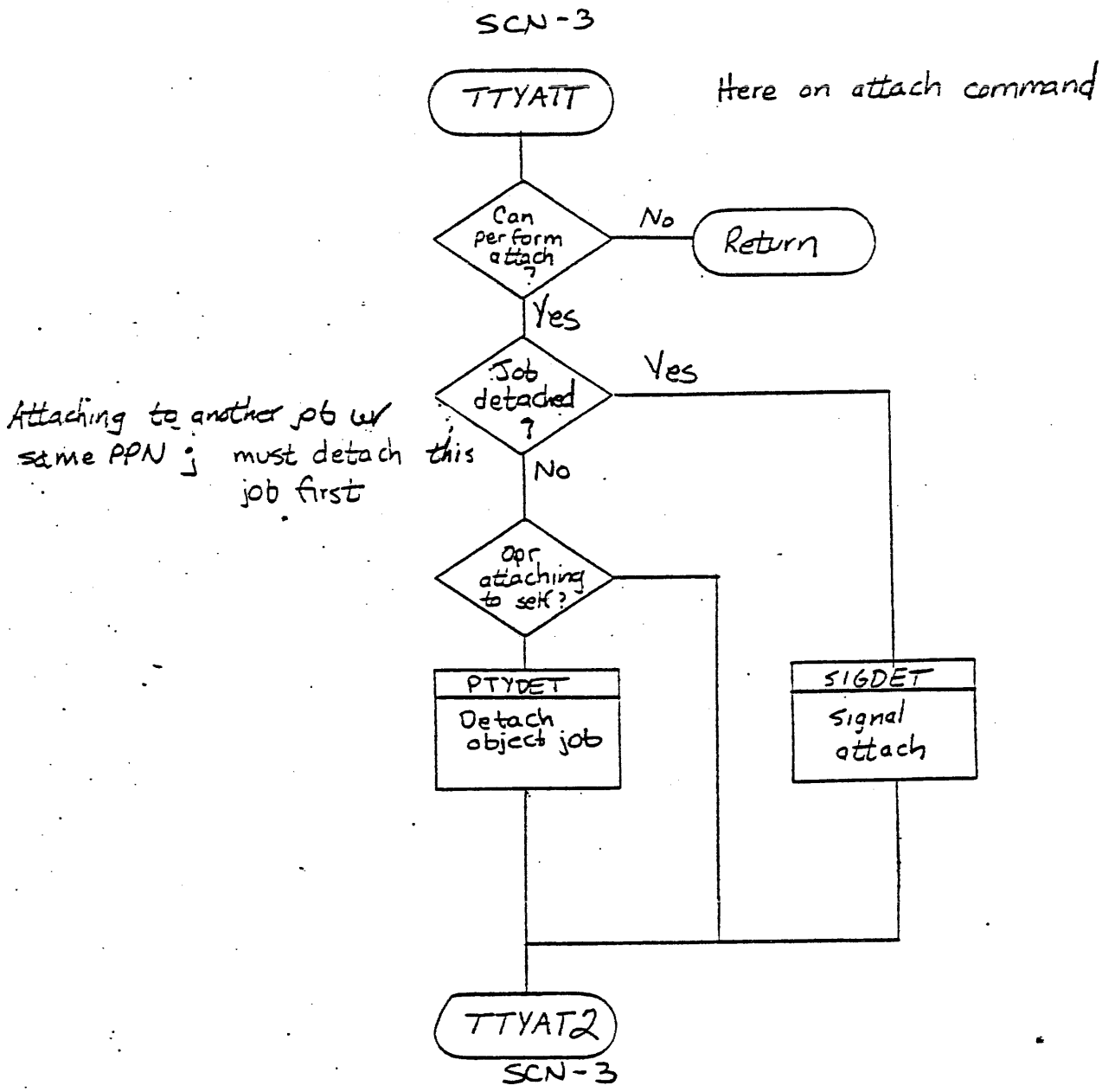
Job vs. Line Information

The reason that job information for terminals is kept separate from line information lies in the processes that use the information. TTY DDBs are used by the UUO processor to locate user buffers and are used only by jobs (only jobs can issue UUOs). Commands are issued by lines so there must be another source of information. COMCON looks at LDBs to find lines with commands typed on them. Refer to figure SCN-1 to see how job and line data structures are inter-related.


Sob Information
TTYTAB
Bits
DDB
JDA
DDB
Channel n
DDB
User Buffers
Chunks
LDB
Live Information
LINTAB
Bits
DDB

The connections between LDBs and DDBs are made when; 1) a job logs in, 2) a job ATTACHs or 3) when a user that is not logged in runs a program. The connection is broken via 1) LOGOUT, 2) DETACH or 3) when the program that a non-logged in user has run finishes. The routines to handle the connections are TTYATI (to attach to a job initially) and TTYATT (for the ATTACH command). When a non-logged-in user issues a request to run a program, the TTY DDB is created and entrance is made at TTYATI.

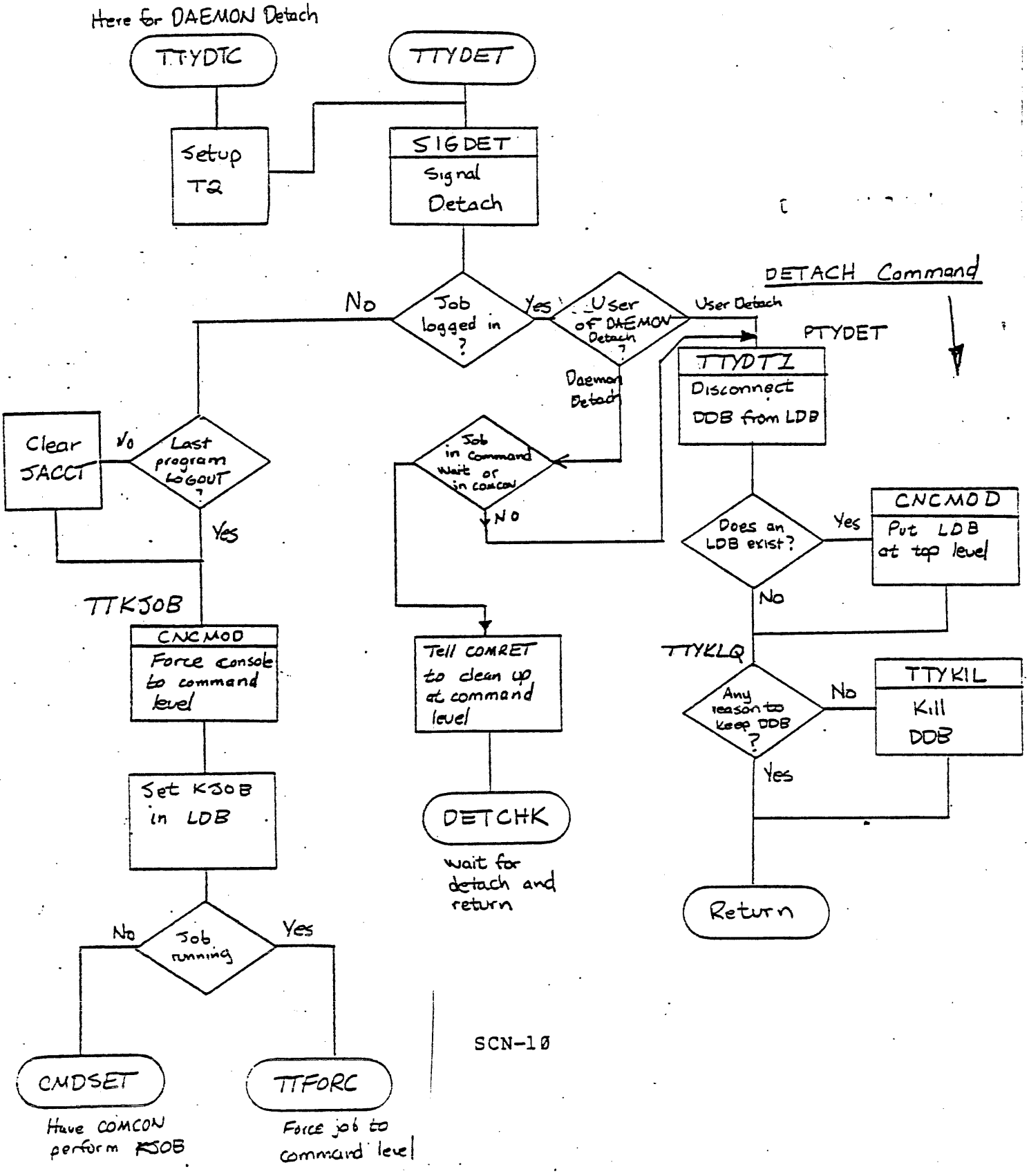

TTYATI
Already a DDB ?
Yes
Return
SCN-2
No
DDB SRC
Get a new DDB
TTYAT2::
TTYDT1
Detach from old DDB if any
Clear time for dataset if needed
Set attached and assigned bits in DDB
Link DDB to LDB
Link LDB to DDB
SETDVL
Put job # in DDB
Store DDB address in TTYTAB
SCN-7
SCNNAM
Set correct station number
SETSTA
Set correct station number
PTY ?
No
Yes
Batch ?
No
Yes
Increment # of Batch jobs
SIGDET
SIGNAL Attach
Return


SCN-3
TTYATT
Here on attach command
Can perform attach ?
No
Return
Yes
Job detached ?
Yes
No
Attaching to another job w/ same PPN ; must detach this job first
Opr attaching to self ?
PTYDET
Detach object job
SIGDET
Signal attach
TTYAT2
SCN-3

The routines to break the links are TTYDET (for all user detaches) and TTYDTC (for a DAEMON detach).

TTY DDBs are created when a job number is assigned (when a program is run). If the job is not logged in, the DDB is deleted when the program finishes. If the job is logged in, the DDB stays until the job KJOBs or DETACHs.

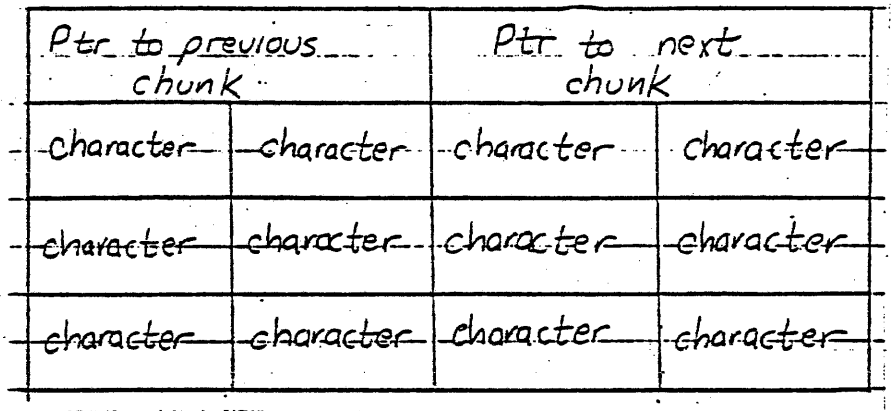
SCN-4


Here for DAEMON Detach
TTYDTC
TTYDET
Setup T2
SIGDET
Signal Detach
DETACH Command
No
Job logged in ?
Yes
User of DAEMON Detach ?
User Detach
PTYDET
TTYDTI
Disconnect DDB from LDB
Daemon Detach
Clear JACCT
No
Last program LOGOUT ?
Yes
Job in Command Wait or in COMCON
NO
Does an LDB exist?
Yes
CNCMOD
Put LDB at top level
No
TTKJOB
CNCMOD
Force console to command level
Tell COMRET to clean up at command level
TTYKLQ
Any reason to keep DDB ?
No
TTYKIL
Kill DDB
Yes
Set KJOB in LDB
DETCHK
wait for detach and return
Return
No
Job running
Yes
CMDSET
Have COMCON perform KJOB
TTFORC
Force job to command level

Chunks

Terminal data is stored in four-word buffers called chunks. Chunks are maintained as doubly linked lists. Each terminal line may potentially have two linked-lists of chunks; one for input and one for output. When chunks are no longer needed by a line they are returned to a free list of chunks. The pointers to the chunks are kept in the LDB.

Each chunk has the the following format:



SCN-5

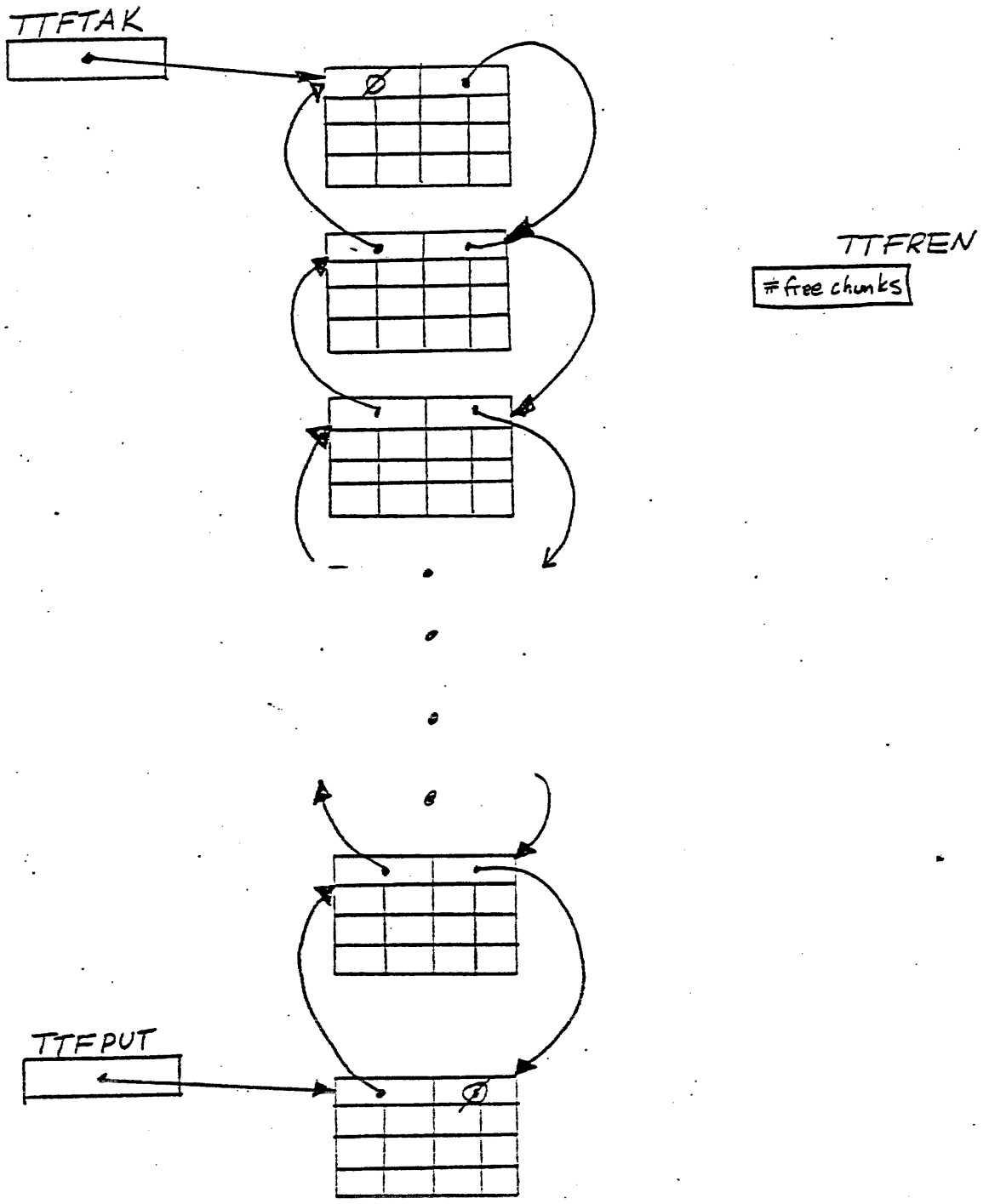
Character data is stored as nine-bit quantities which permits a maximum of 12 characters to be stored in a chunk (4 to a word).

All the chunks are kept in a pool in the monitor. The pool is initialized by the TTYINI routine in SYSINI. There, the space is allocated and all the initial links created.

The location TTFTAK points to the first free chunk in the pool. When a line needs a chunk, it gets the chunk pointed to by this location. TTFPUT points to the last free chunk in the list and returned chunks are stored after this chunk. TTFREN contains the number of free chunks in the system.

SCN-6

Chunk Free List



The total number of chunks allocated at MONGEN time is determined by the following formula:

SCN-7

chunks allocated at MONGEN

If TTPLEN < 10 , TTCHKN = TTPLEN * 11

If TTPLEN < 20 , TTCHKN = TTPLEN * 10

If TTPLEN < 40 , TTCHKN = TTPLEN * 7

Else , TTCHKN = TTPLEN * 6

where:

TTPLEN = TCONLN + PTYN + 1

and

PTYN = # of PTYs

TCONLN = M.TLTL + M.RMCR + M.CPU

and

M.TLTL = # TTY lines + #TTYs on CFE + #XTC lines + 1 (KLINIK

M.RMCR = #Network TTYs

M.CPU = # CPUS

Characters are placed in and removed from chunks using three macros: LDCHK, LDCHR and STCHK. Macros are used in place of subroutines to make handling faster (at a slight cost of size). They do the following things:

1. LDCHK - takes a character out of a chunk but does give back used chunks (useful when echoing input).

2. LDCHKR - takes a character out of a chunk and deallocates (returns) used chunks to the free list if necessary.

3. STCHK - puts a character in a chunk, allocating chunks from the free list if necessary.

The last two must only be called with SCNSER interrupts turned off or problems will result.

The SCNON macro turns on scanner interrupts and SCNOFF turns off scanner interrupts. Both are defined in S.MAC.

TERMINAL I/O

Terminal I/O can be a difficult subject to understand because of the many options and special characters involved. Terminals can be in character mode, line mode, image mode or packed image mode. I/O can be performed using INs, OUTs, TTCALLs or TRMOP.s. There are over ten special characters, each demanding unusual treatment. Sometimes COMCON must use the characters; other times the user (through UUOCON) wants to use the characters. Some terminals have special character sets or modems. Each terminal type has its own fill, length, width, etc. The SCNSER module and various device drivers must handle all these cases, and handle them efficiently.

To simplify the presentation, the next two sections will describe how terminal I/O is accomplished on one terminal for the following "simple" case:

1. DTE-20/PDP-11 based local communications

2. Terminal is in line mode

3. No special characters (only Carraige return/line feed)

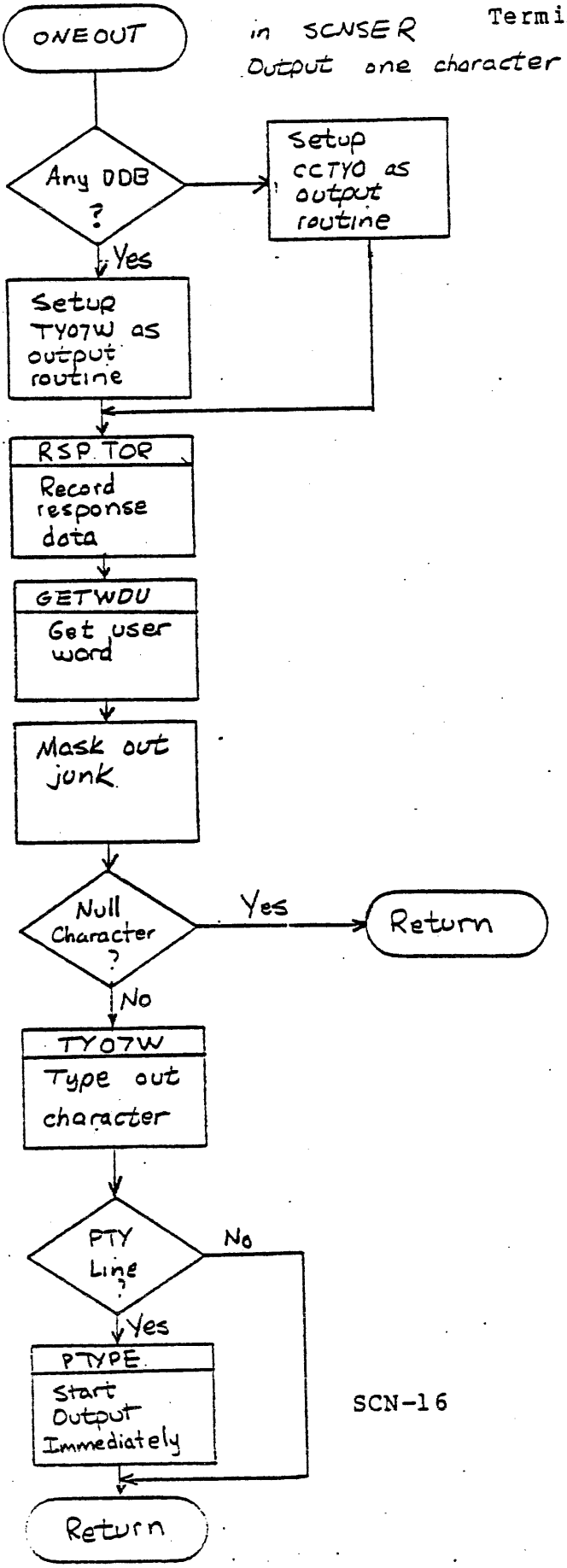
4. User programs will use TTCALLs for I/O

Variations can be easily understood in relation to this basic structure. In a later module, the specifics of DC10, DL10-based and DTE-based communications will be discussed.

Output

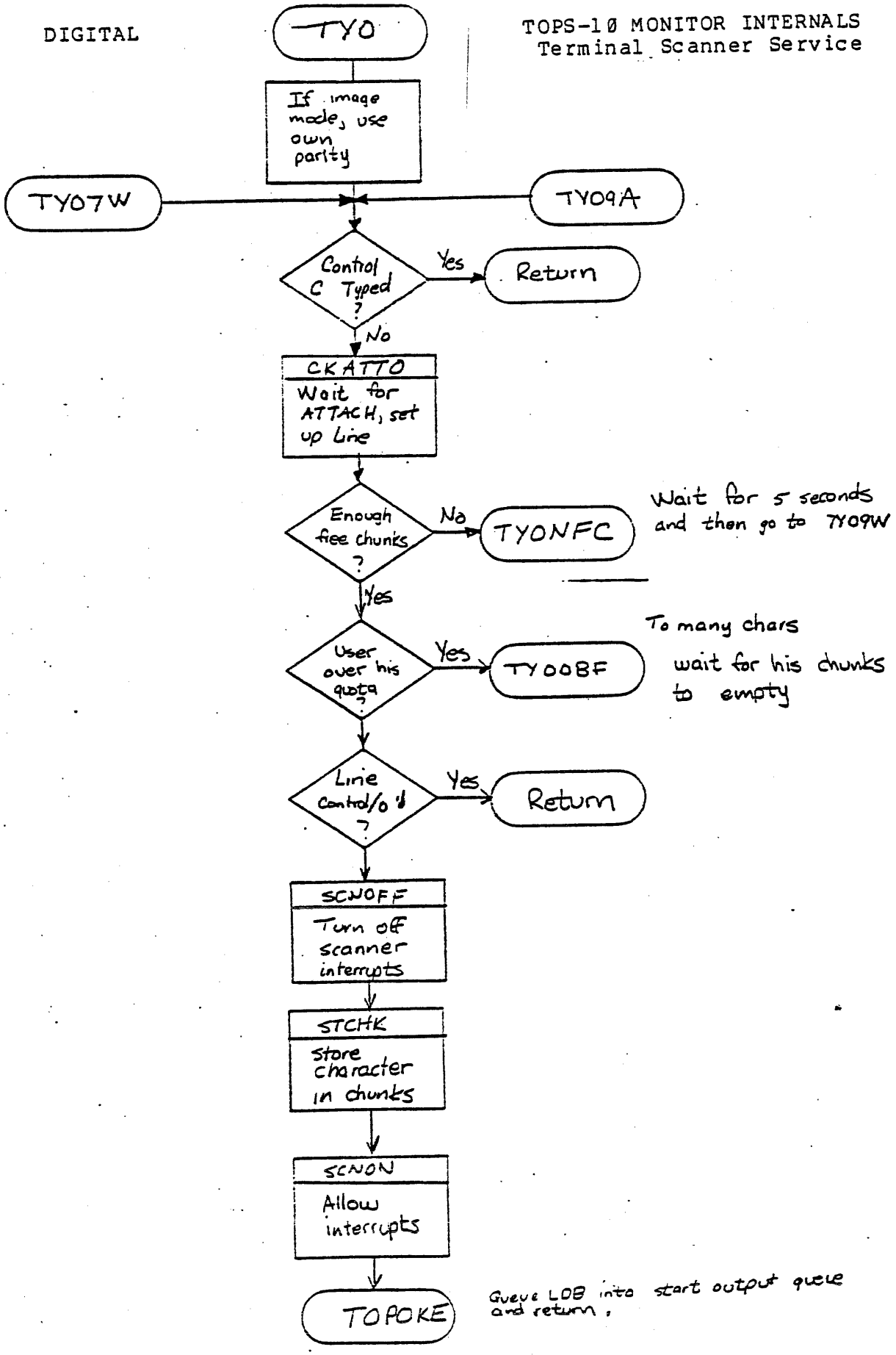
Output is the easiest case to consider. It can come from two sources: UUOCON (via a TTCALL) or COMCON (the monitor must print something). Terminal output is non-blocking: once the output has been queued up, the job or process may continue to run without having to wait for the output to finish.

Consider the case of TTCALL 1, (output one ASCIZ character). When the UUO is issued, control passes from UUOCON to ONEOUT in SCNSER. This routine (figure SCN-8) will get the user character, check to see that a null character is not being output and start the output. In addition, terminal response data is recorded.


ONEOUT
in SCNSER
Output one character
Any DDB ?
Setup CCTYO as output routine
Yes
Setup TY07W as output routine
RSP.TOR
Record response data
GETWDU
Get user word
Mask out junk
Null Character ?
Yes
Return
No
TY07W
Type out character
PTY Line ?
No
Yes
PTYPE
Start Output Immediately
Return

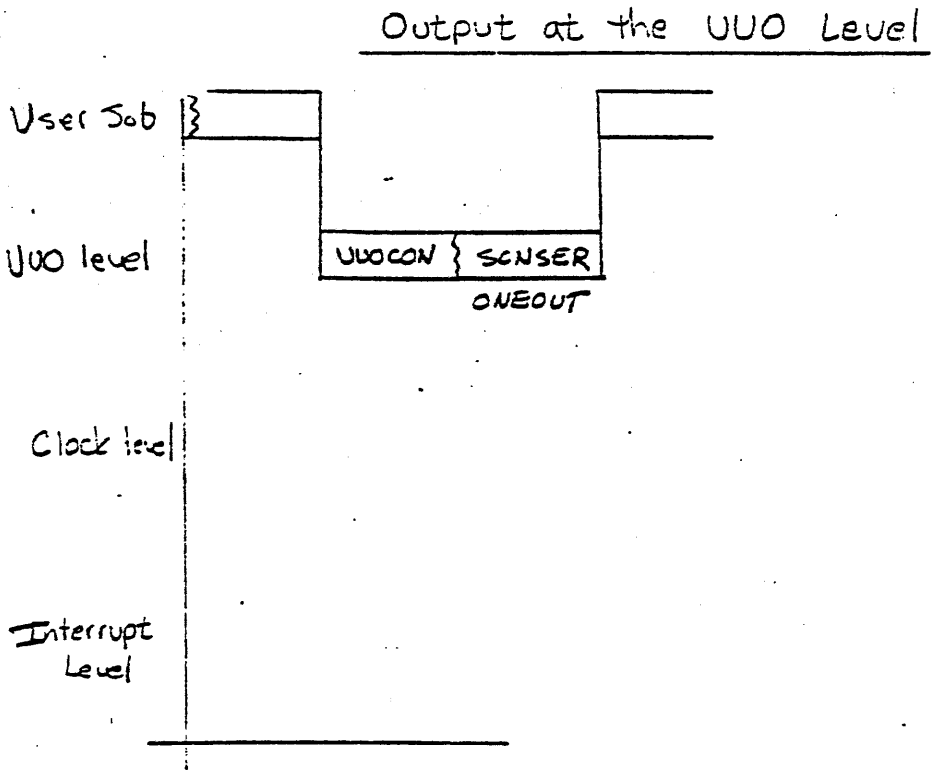
ONEOUT will call TYO7W (also in SCNSER) to perform the output. There, the character will be placed in a chunk in the output list. Checks are made to see if Control/C has been typed, if enough free chunks are left, if a line has been Control/O'd (throw away output), or if a user has too much output in the queue.

The output request is made by entering the LDB of the line into the start output queue. This is done by adjusting the LDBQUE word of the LDB.


TY0
If image mode, use own parity
TY07W
TY09A
Control C Typed ?
Yes
Return
No
CKATTO
Wait for ATTACH, set up Line
Enough free chunks ?
No
TY0NFC
Wait for 5 seconds and then go to TY09W
Yes
User over his quota ?
Yes
TY00BF
To many chars wait for his chunks to empty
Line Control/O 'd ?
Yes
Return
SCNOFF
Turn off scanner interrupts
STCHK
store character in chunks
SCNON
Allow interrupts
TOPOKE
Queue LOB into start output queue and return.

At this point, return is effected from the UUO and the user program continues to run. The character has not reached the terminal but the job can continue because terminal output is non-blocking. A time chart would look like this:

SCN-10



But when does the output get started? The clock cycle is responsible for calling the correct routines. In CLOCK1 at CIP5:, there is the following instruction

```
CIP5::  PUSHJ   P,@.CPSTO
```

.CPSTO is a location within the CPU data block that holds the terminal output routine, in this case .SCØTIC.

The function of .SCØTIC is to start output for a variety of types of lines. It will call several routines:

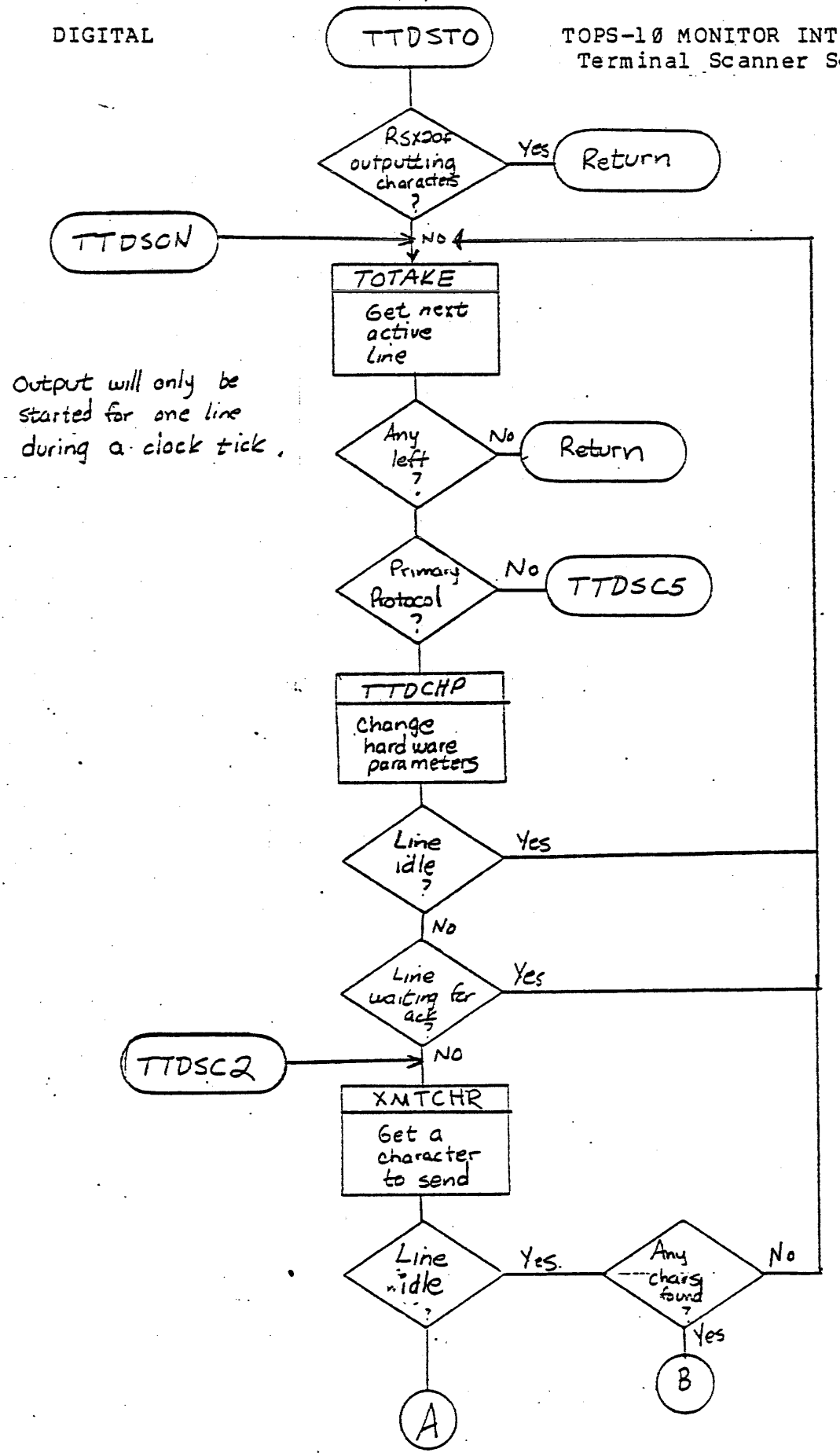
```
CTYSTO - CTY for KS10, KI10
TTDSTO - DTE-20 basedd terminals
NETSTO - Network terminals
D76STO - DC76 Terminals
DLØSTO - DC10 Terminals
DL1STO - DL10 Terminals
DZSTO  - DZ11 Terminals (KS10 only)
```

The routine for our example is TTDSTO in TTDINT (the device driver for DTE-20 terminals).

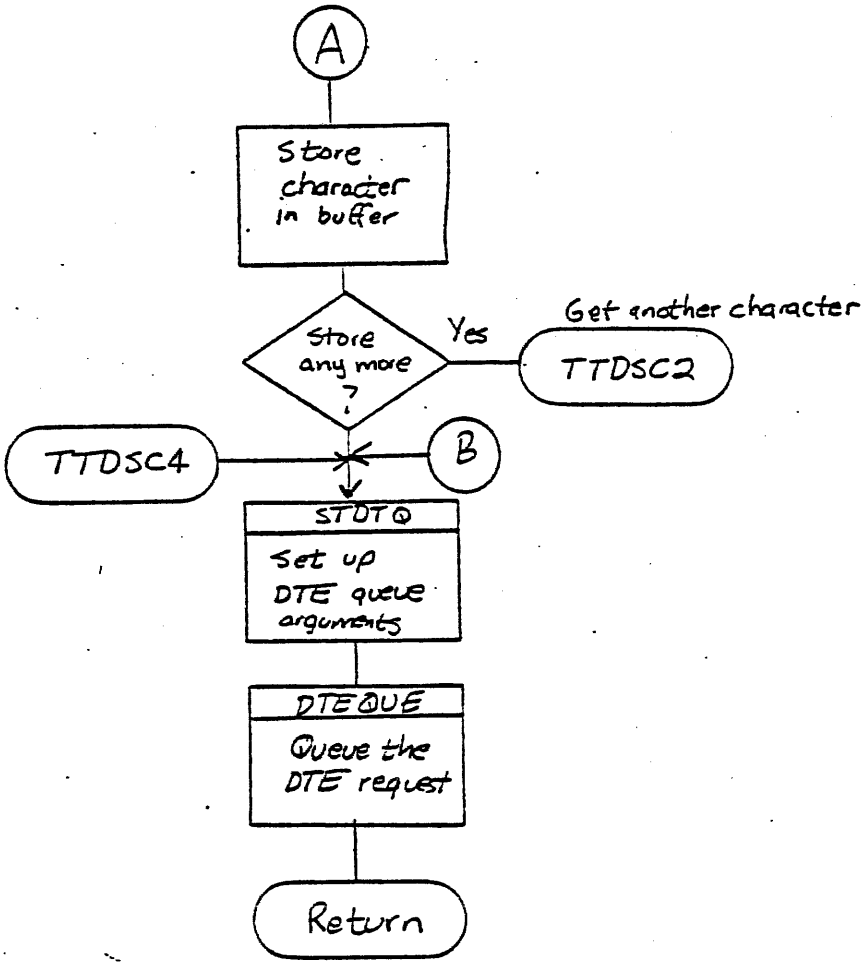
TTDSTO searches for an LDB in the queue, gets a character from the chunk, deallocates the chunk if necessary, puts the character in the DTE-20 queue and starts the DTE. Return is made is made from TTDSTO after a string or charcter has been sent. This means that only one DTE-based terminal line will be serviced every clock tick. This is true for every line type; only one line of that type will be started every clock tick.

DIGITAL

TOPS-10 MONITOR INTERNALS
Terminal Scanner Service

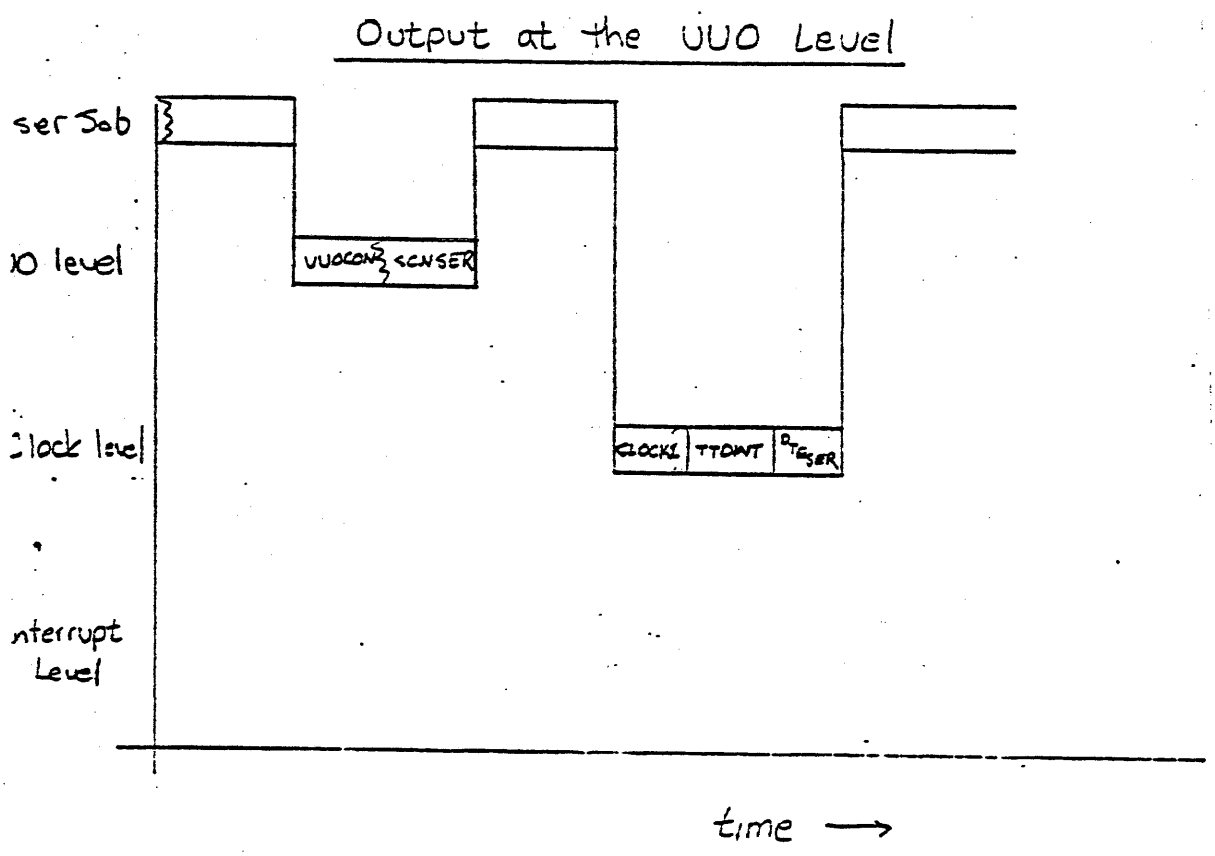


SCN-12

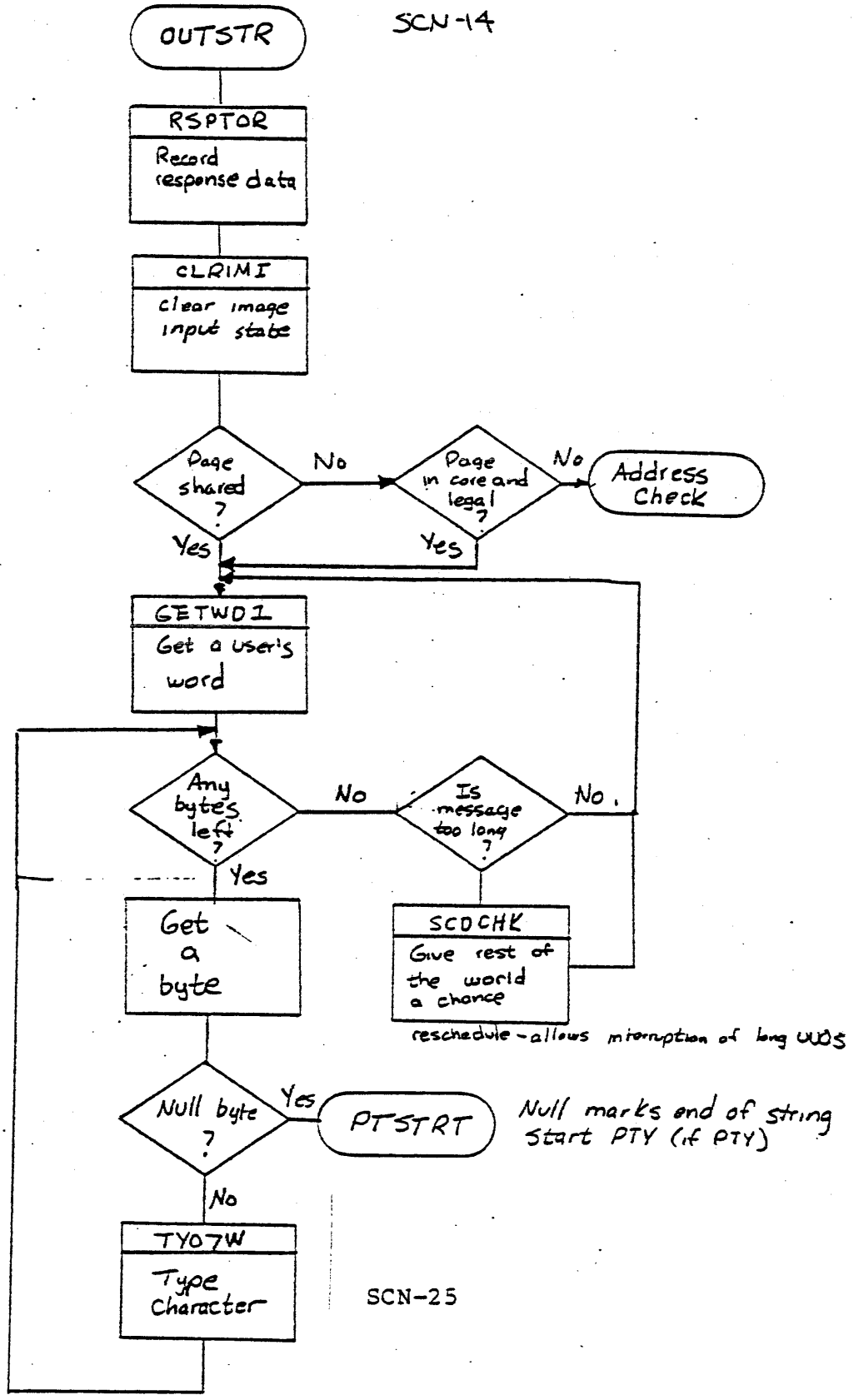

A
Store character in buffer
Store any more ?
Yes
Get another character
TTDSC2
TTDSC4
B
STDTQ
Set up DTE queue arguments
DTEQUE
Queue the DTE request
Return

The DTE driver will not be discussed in this module. Suffice to say that the character gets to the PDP-11 and is finally output. The sequence of actions is:

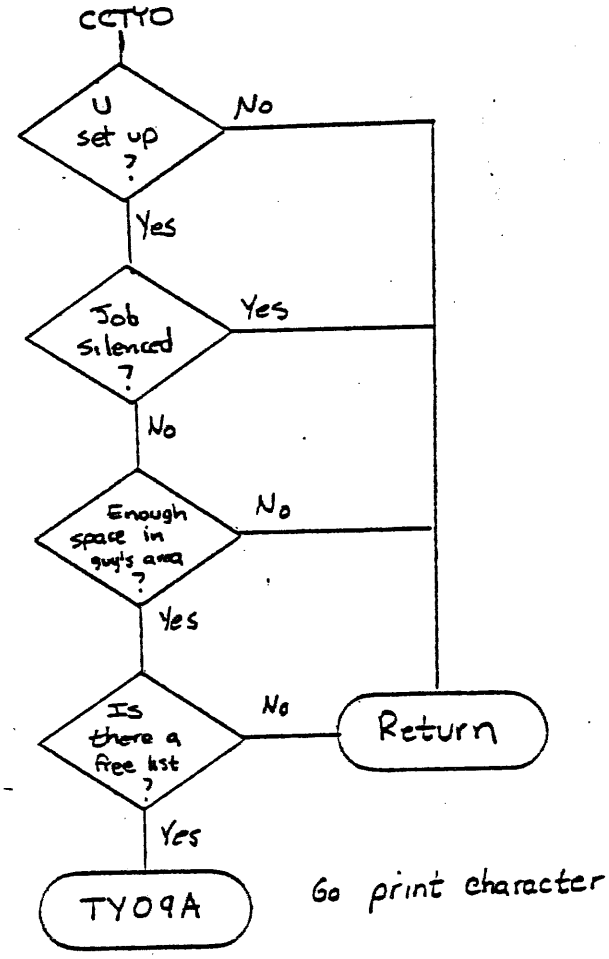
SCN-13



The process to output a string (TTCALL 3,) is only slightly more complicated. Essentially it calls the TYO7W routine multiple times. From that point on, the same routines as were previously discussed take over. The flow can be seen in Figure SCN-14.

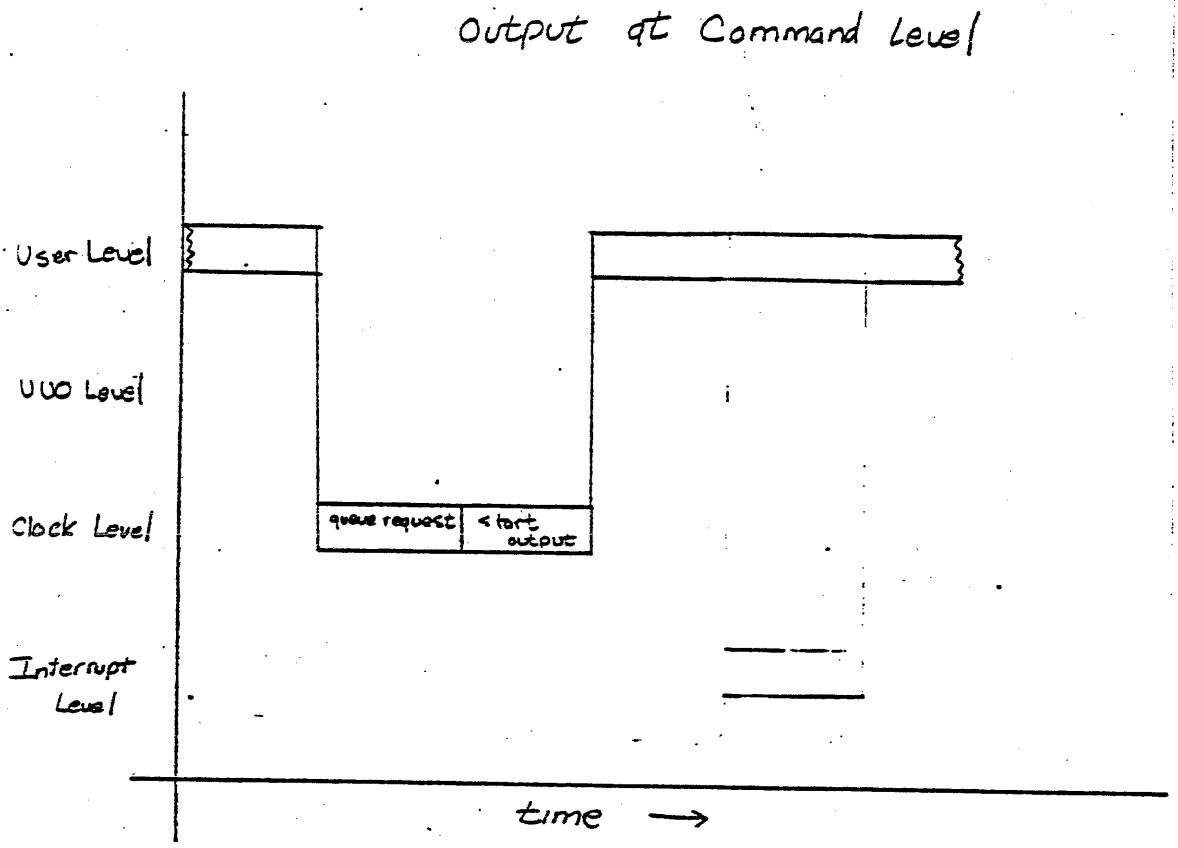

OUTSTR
SCN-14
RSPTOR
Record response data
CLRIMI
clear image input state
Page shared ?
No
Page in core and legal ?
No
Address Check
Yes
Yes
GETWDI
Get a user's word
Any bytes left ?
No
Is message too long ?
No
Yes
Get a byte
SCDCHK
Give rest of the world a chance
reschedule - allows interruption of long UUOs
Null byte ?
Yes
PTSTRT
Null marks end of string
Start PTY (if PTY)
No
TYO7W
Type Character
SCN-25

The command processor (COMCON) interface to these routines follows a slightly modified path. When output must be done by COMCON, CONMES is called. CONMES calls COMTYO for each character which in turn calls CCTYO (in SCNSER).



Observe that control reaches TY09W, part of the same routine (TY07W) that performs output for UUOs. Output follows the same path from that point on with one difference. Since COMCON is part of the clock cycle, the decision to queue the output request and start the output request may occur in the same clock cycle. If not started right away, output will begin at a later clock interrupt.

SCN-16


Output at Command Level
User Level
UUO Level
Clock Level
queue request
start output
Interrupt Level
time

Input

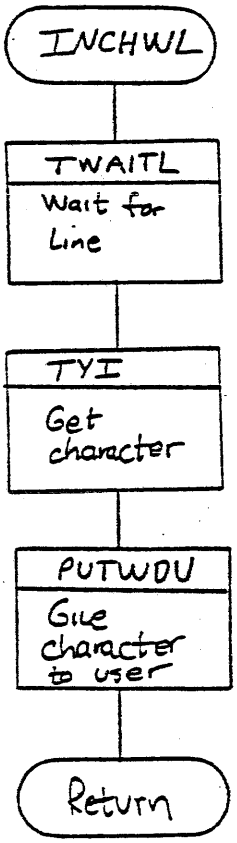
Terminal input is more difficult to understand because oc echoing and the fact that the job will change status and go into terminal wait state, requiring rescheduling. When a break character is received, the job can continue.

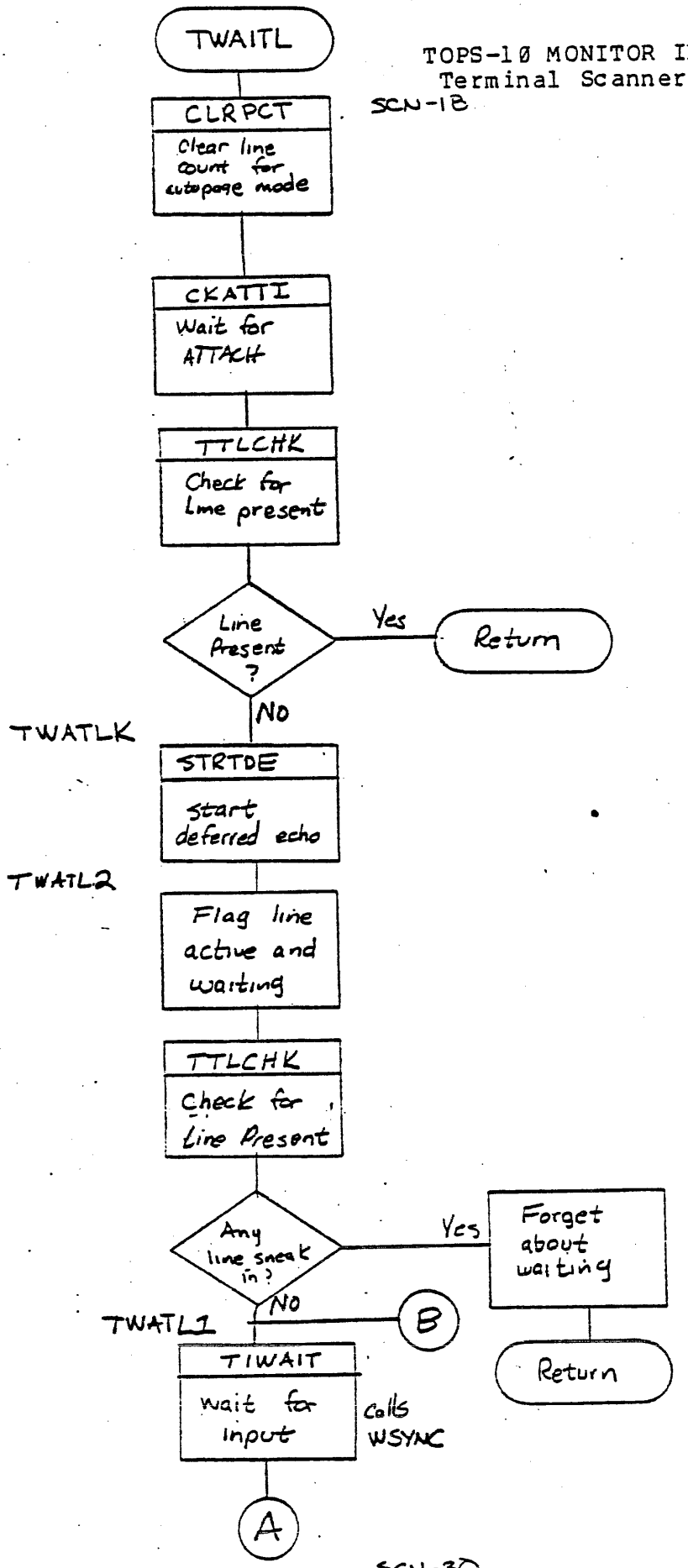
Assuming the same "simple" case as before, consider what happens when a job is waiting for a line of input from the attached terminal. The job will issue the INCHWL UUO (INput CHaracter and Wait, Line mode).

When this monitor call is issued, the terminal goes into TI wait until a break character is received. Only the first character is returned at that time. Successive UUOs return the reamining characters until none remain. At that time, the next INCHWL will put the terminal back into TI wait.

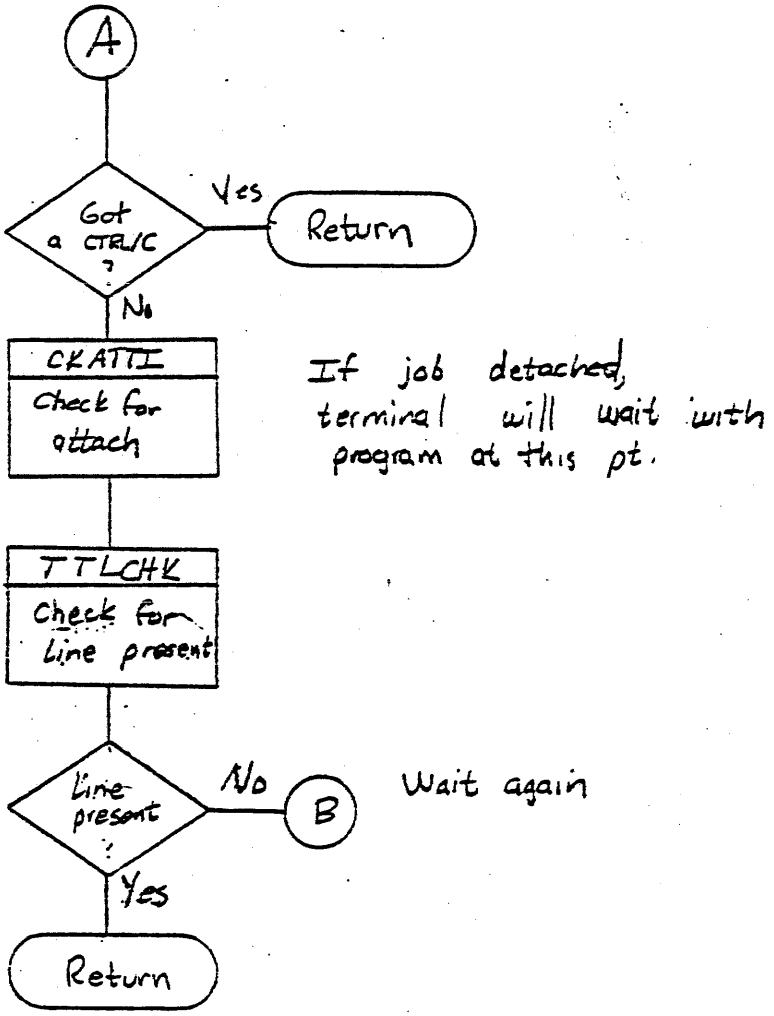
When INCHWL is issued, control passes from UUOCON to INCHWL in SCNSER. This routine calls in turn TWAITL to wait for a line, TYI to get a character and PUTWDU to give the character to the user. If a line has been input already, return from TWAITL is immediate, otherwise, the job goes into TI wait and will not return until a break character is received. The job blocks.

SCN-17


INCHWL
TWAITL
Wait for Line
TYI
Get character
PUTWDU
Give character to user
Return


TWAITL
CLRPCT
Clear line count for autopage mode
CKATTI
Wait for ATTACH
TTLCHK
Check for line present
Line Present ?
Yes
Return
NO
TWATLK
STRTDE
Start deferred echo
TWATL2
Flag line active and waiting
TTLCHK
Check for Line Present
Any line sneak in?
Yes
Forget about waiting
Return
NO
TWATL1
B
TIWAIT
wait for input
Calls WSYNC
A

SCN-19


A
Got a CTRL/C ?
Yes
Return
No
CKATTI
Check for attach
If job detached, terminal will wait with program at this pt.
TTLCHK
Check for Line present
Line present
No
B
Wait again
Yes
Return

After a character is typed, the particular device or front end to which the terminal is attached will eventually interrupt the KL10. After a certain amount of processing, the interrupt routine will call RECINT in SCNSER. This is true for all terminal device interrupt routines.

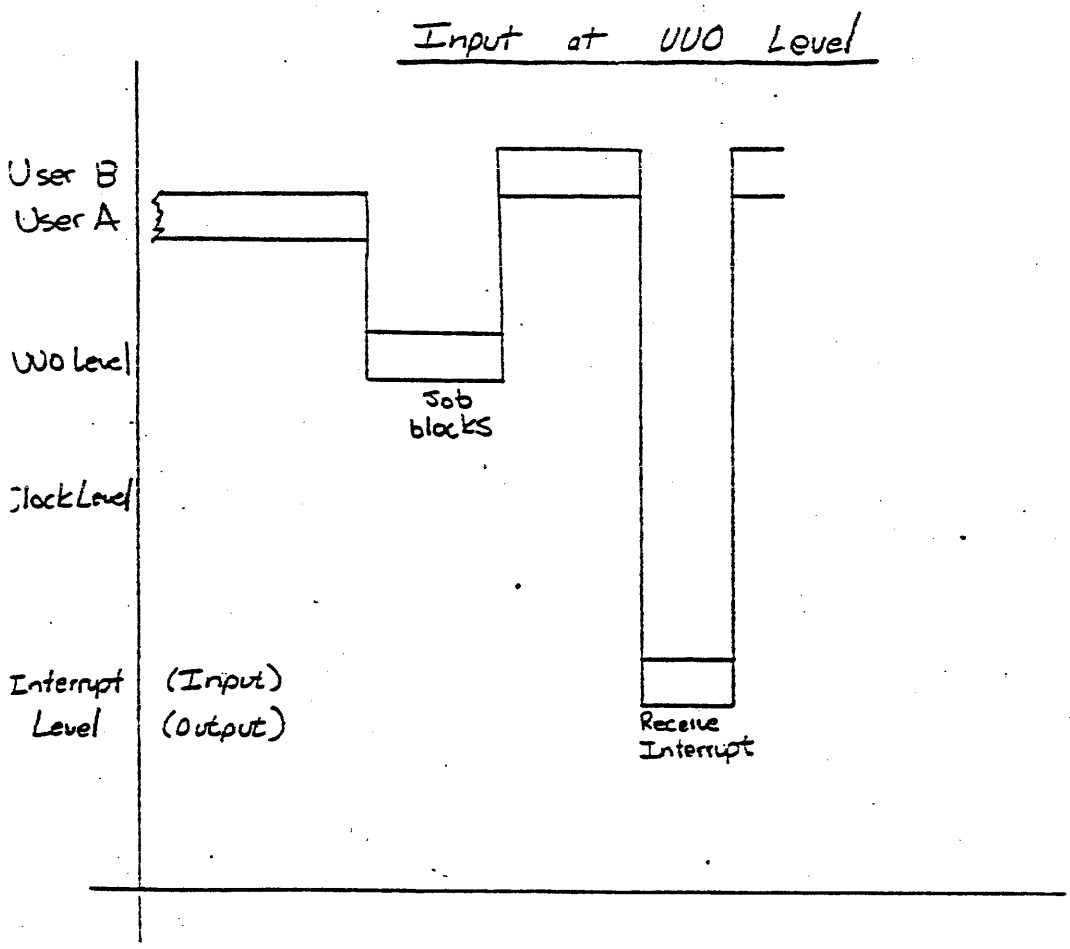
RECINT is the heart of terminal input. It must handle many special cases. Among the tests it makes are:

1. Network Virtual Terminals
2. Dataset interrupt
3. Half-duplex terminal
4. Packed image mode
5. Input image mode
6. Special characters
7. Break characters
8. Auto CRLF and upper case conversion
9. Positioning on a line

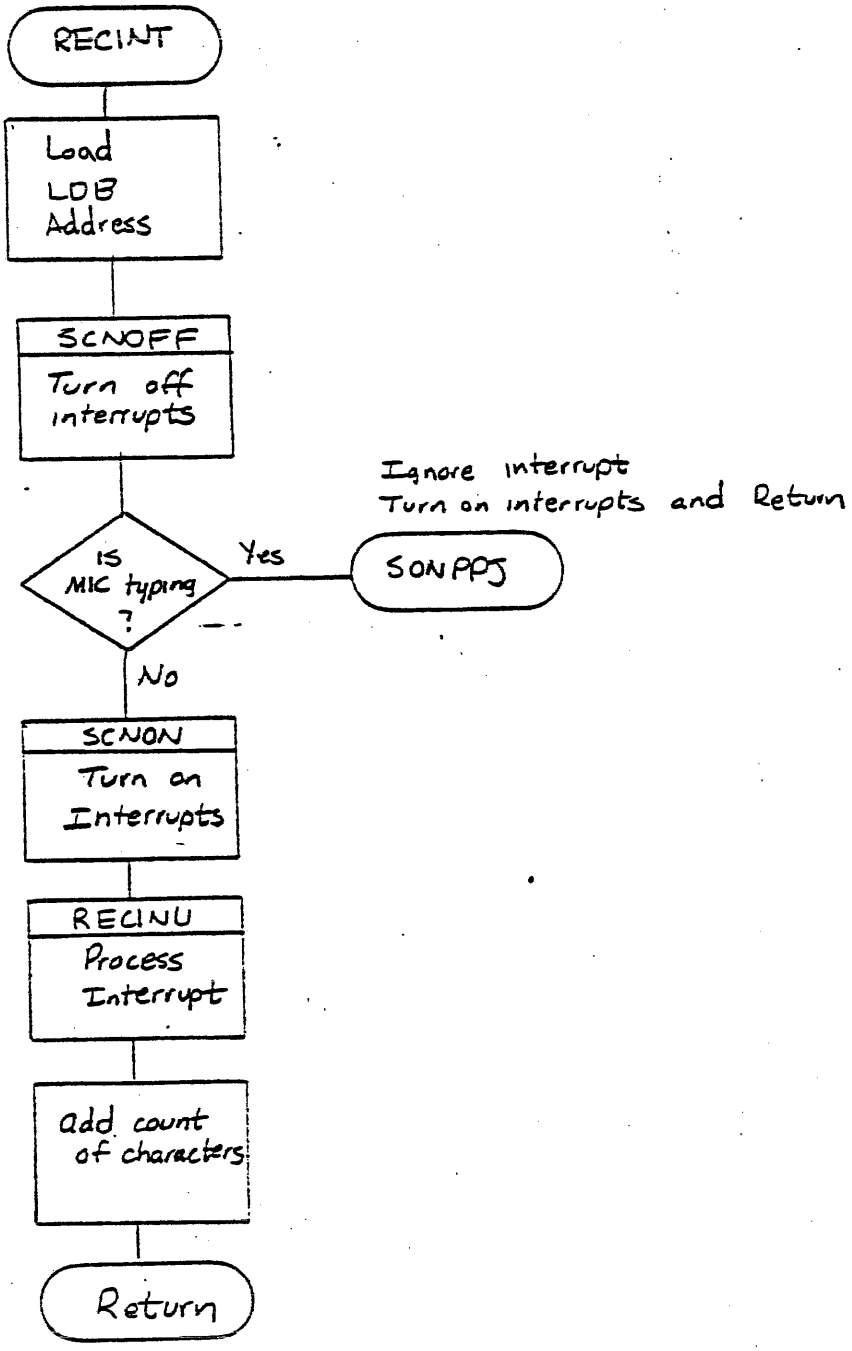
Once all the tests have been met, the character is placed in a chunk and the TOPOKE routine is called to terminate RECINT.

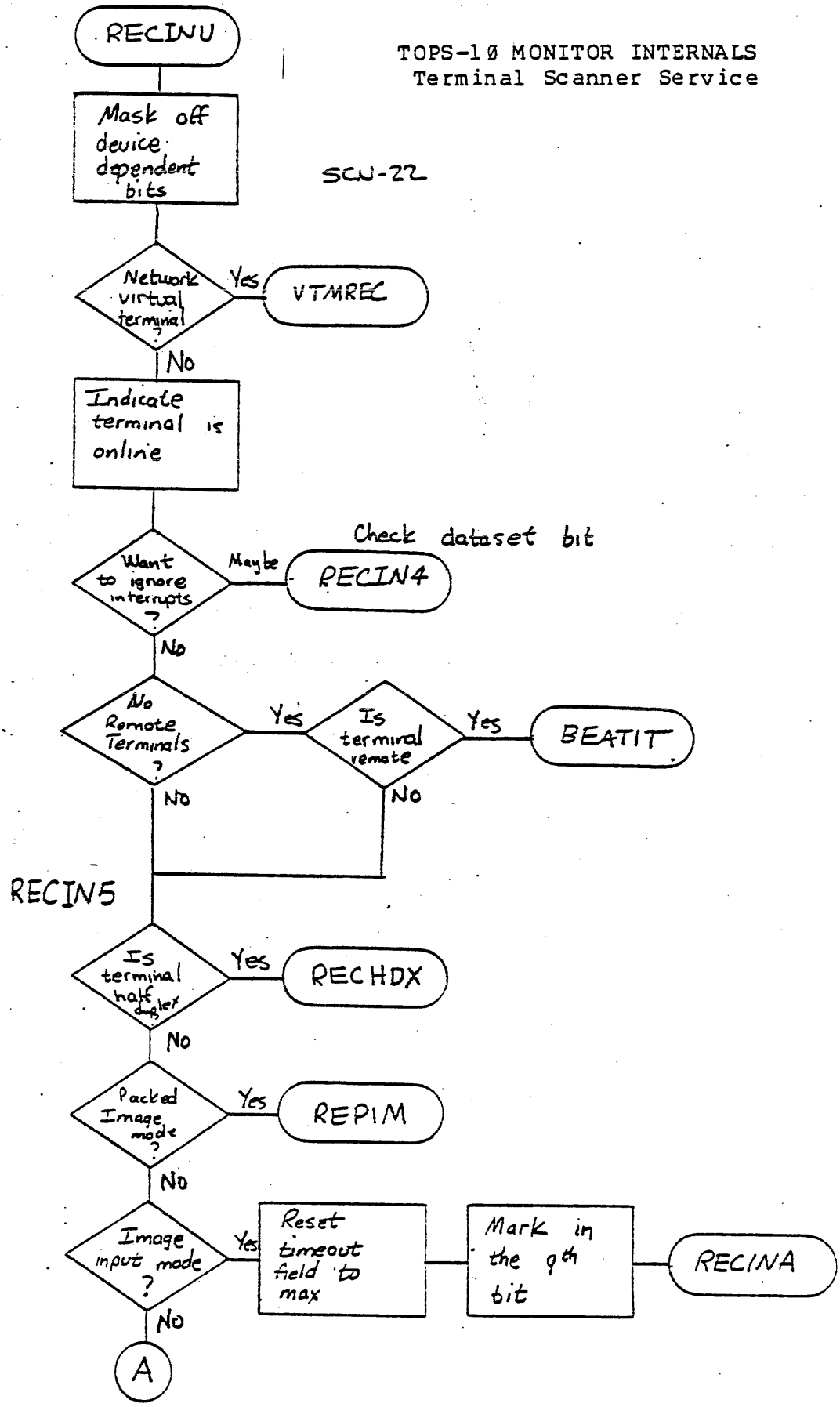
Calling TOPOKE is far more important than it appears. Remember from the output section that TOPOKE places the LDB into the output queue. But in this case, what is in the output queue? Nothing. Then why do it? The answer to that will be seen in a moment. First, review what has happened so far by looking at the following diagram:

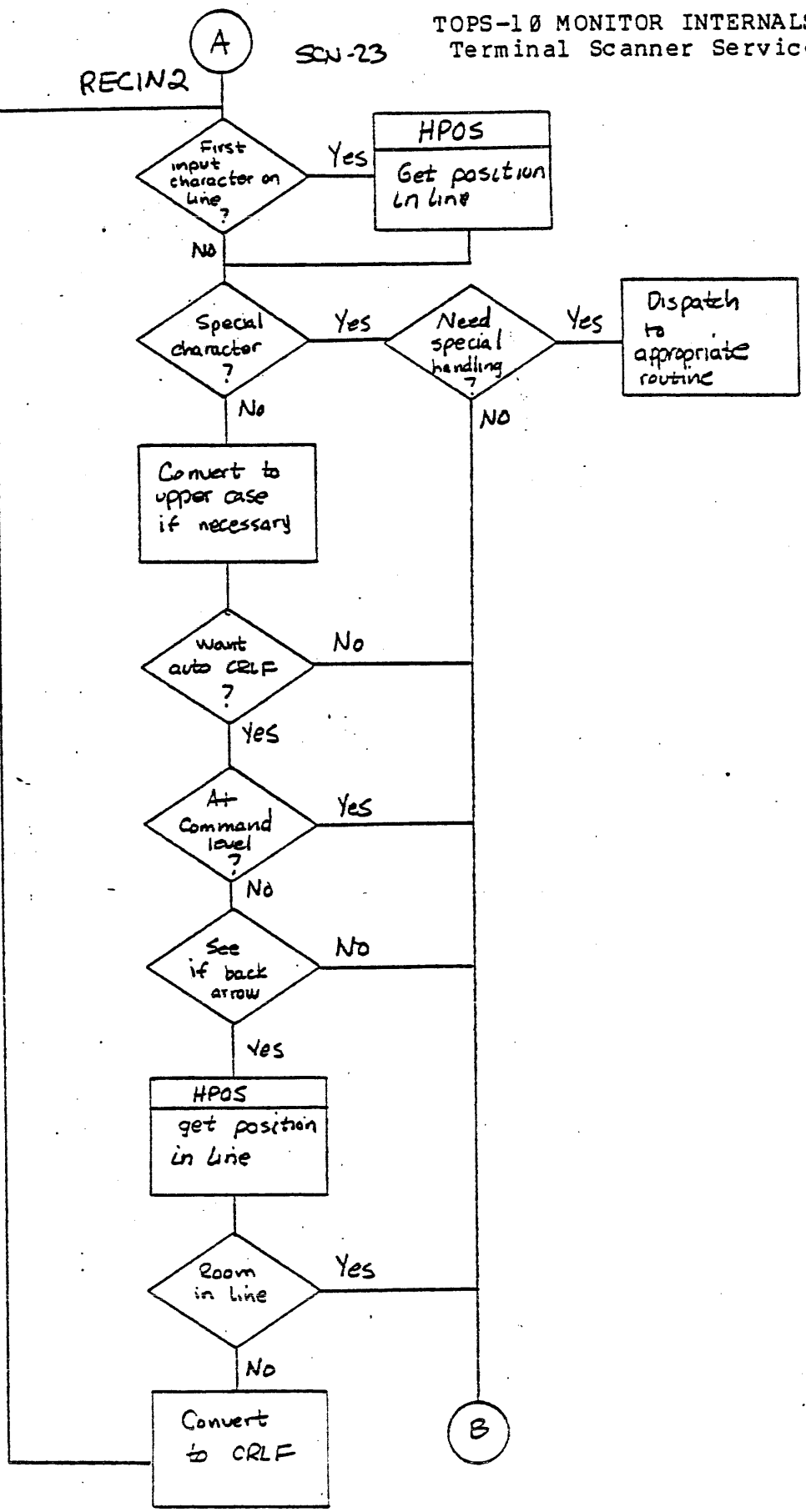
SCN-20

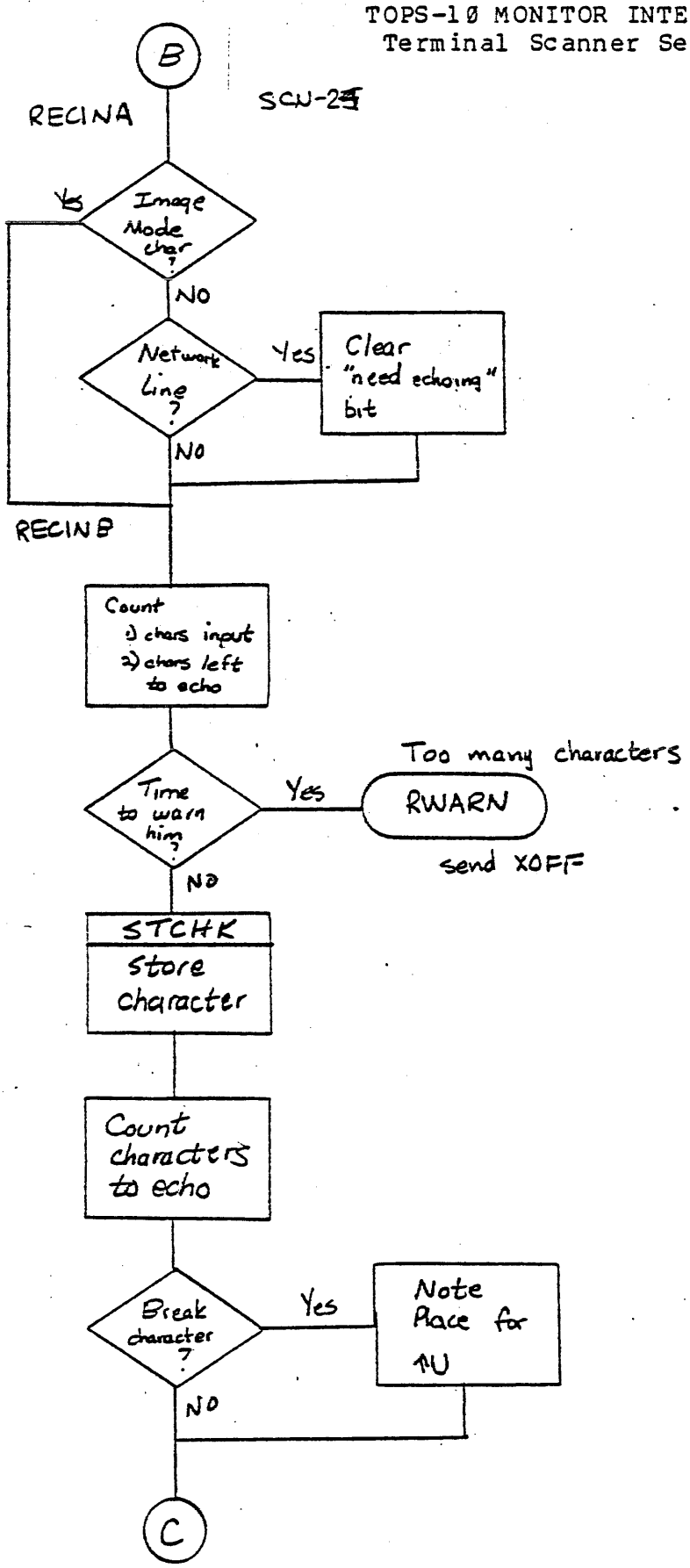

Input at UUO Level
User B
User A
UUO Level
Job blocks
Clock Level
Interrupt Level
(Input)
(Output)
Receive Interrupt

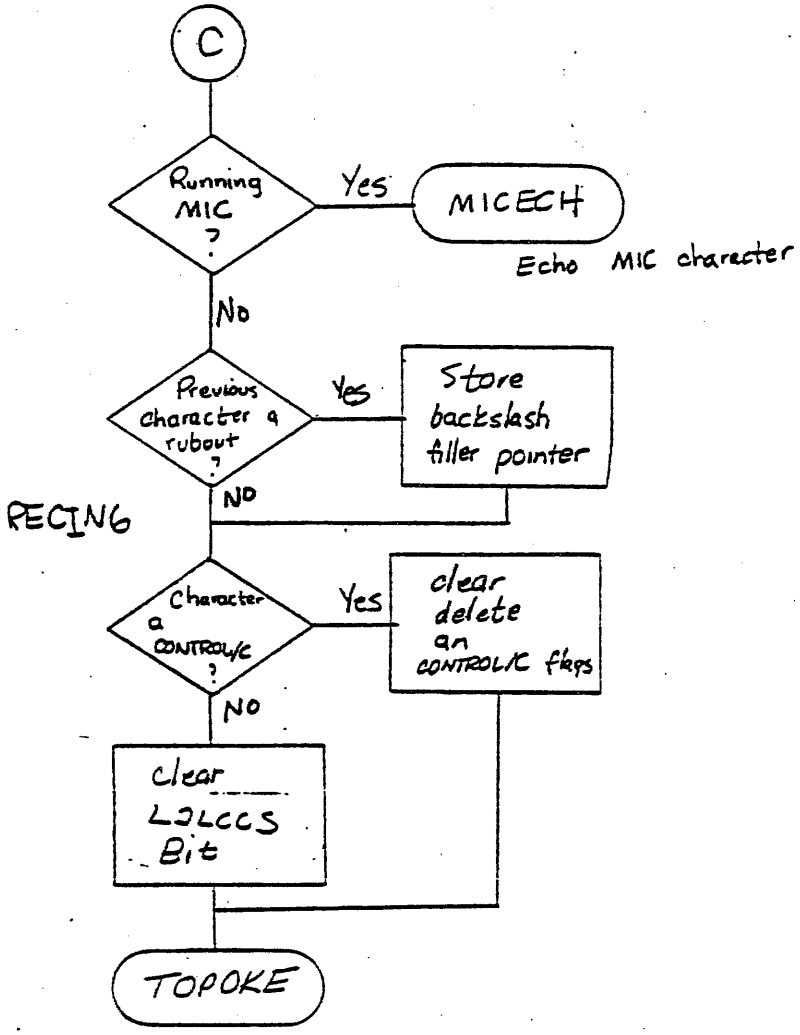
SCN-21


RECINT
Load LDB Address
SCNOFF
Turn off interrupts
Ignore interrupt
Turn on interrupts and Return
IS MIC typing ?
Yes
SONPPS
No
SCNON
Turn on Interrupts
RECNU
Process Interrupt
Add count of characters
Return


RECINU
Mask off device dependent bits
Network virtual terminal ?
Yes
VTMREC
No
Indicate terminal is online
Check dataset bit
Want to ignore interrupts ?
Maybe
RECIN4
No
No Remote Terminals ?
Yes
Is terminal remote
Yes
BEATIT
No
No
RECIN5
Is terminal half duplex
Yes
RECHDX
No
Packed Image mode ?
Yes
REPIM
No
Image input mode ?
Yes
Reset timeout field to max
Mark in the 9th bit
RECINA
No
A

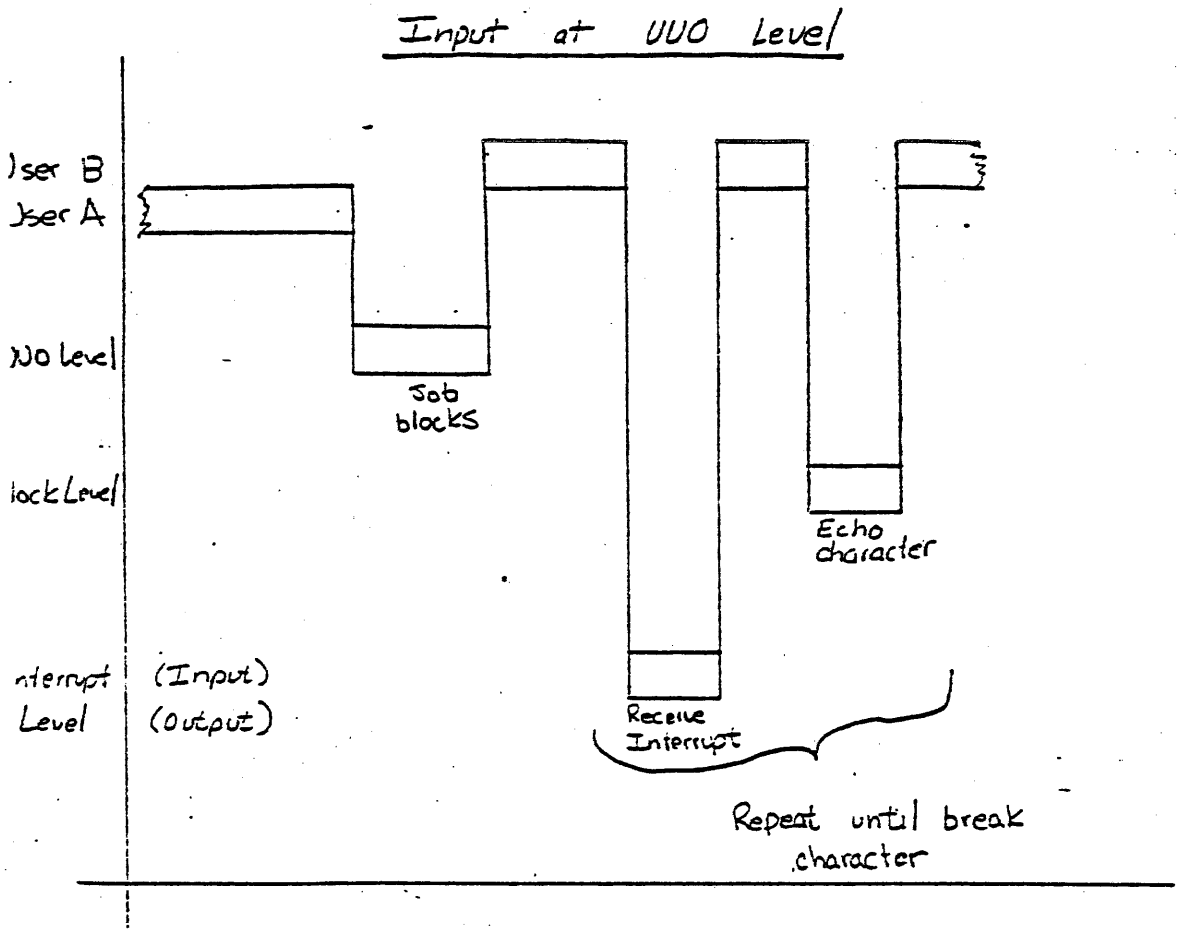

A
SCN-23
RECIN2
First input character on line ?
Yes
HPOS
Get position in line
No
Special character ?
Yes
Need special handling ?
Yes
Dispatch to appropriate routine
No
NO
Convert to upper case if necessary
Want auto CRLF ?
No
Yes
At Command level ?
Yes
No
See if back arrow
No
Yes
HPOS
get position in line
Room in line
Yes
No
Convert to CRLF
B


B
RECINA
SCN-25
Image Mode char ?
YES
NO
Network Line ?
Yes
Clear "need echoing" bit
NO
RECINB
Count
1) chars input
2) chars left to echo
Time to warn him ?
Yes
Too many characters
RWARN
send XOFF
NO
STCHK
Store character
Count characters to echo
Break character ?
Yes
Note Place for ↑U
NO
C


C
Running MIC ?
Yes
MICECH
Echo MIC character
No
Previous character a rubout ?
Yes
Store backslash filler pointer
No
RECING
Character a CONTROL/C ?
Yes
clear delete an CONTROL/C flags
No
clear LOLCCS Bit
TOPOKE

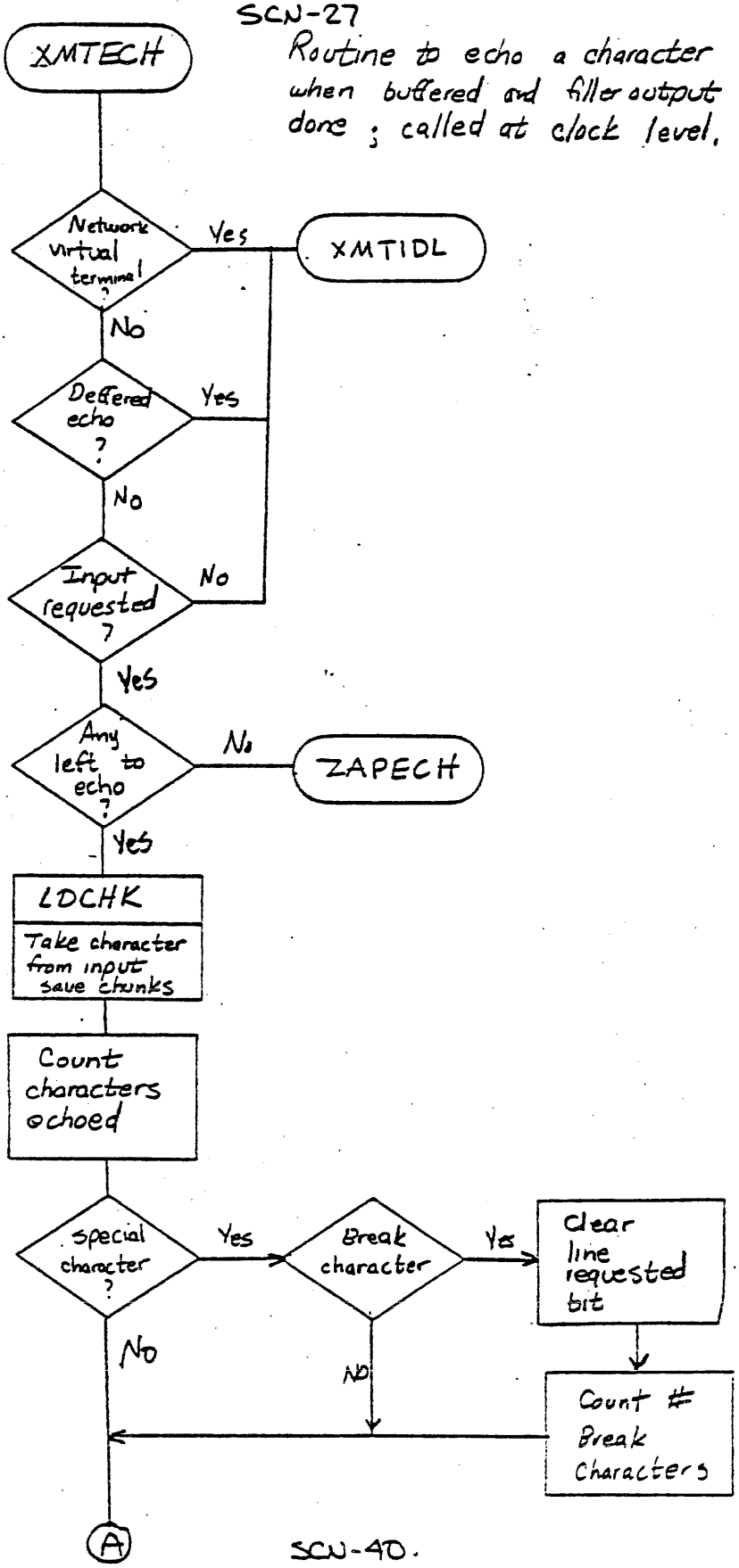
Once the interrupt has been dismissed, the character has been stored in an input chunk and the LDB is in the output queue (with no output in the queue). Nothing happens until the clock interrupt. There, the monitor, during its service of the terminal queues, sees the LDB in the queue and calls XMTCHR to start output for that line. Because there is no output, XMTCHR will call ZAPBUF to clean up the output chunks. ZAPBUF, besides deallocating chunks, is also responsible for starting character echoing via a call to XMTECH. RECINT placed the LDB into the output queue without any output so that ZAPBUF and eventually XMTECH will be called to produce the necessary echoing. The cycle of interrupt level (receive character)-clock level (echo character) will be repeated until a break character is received.

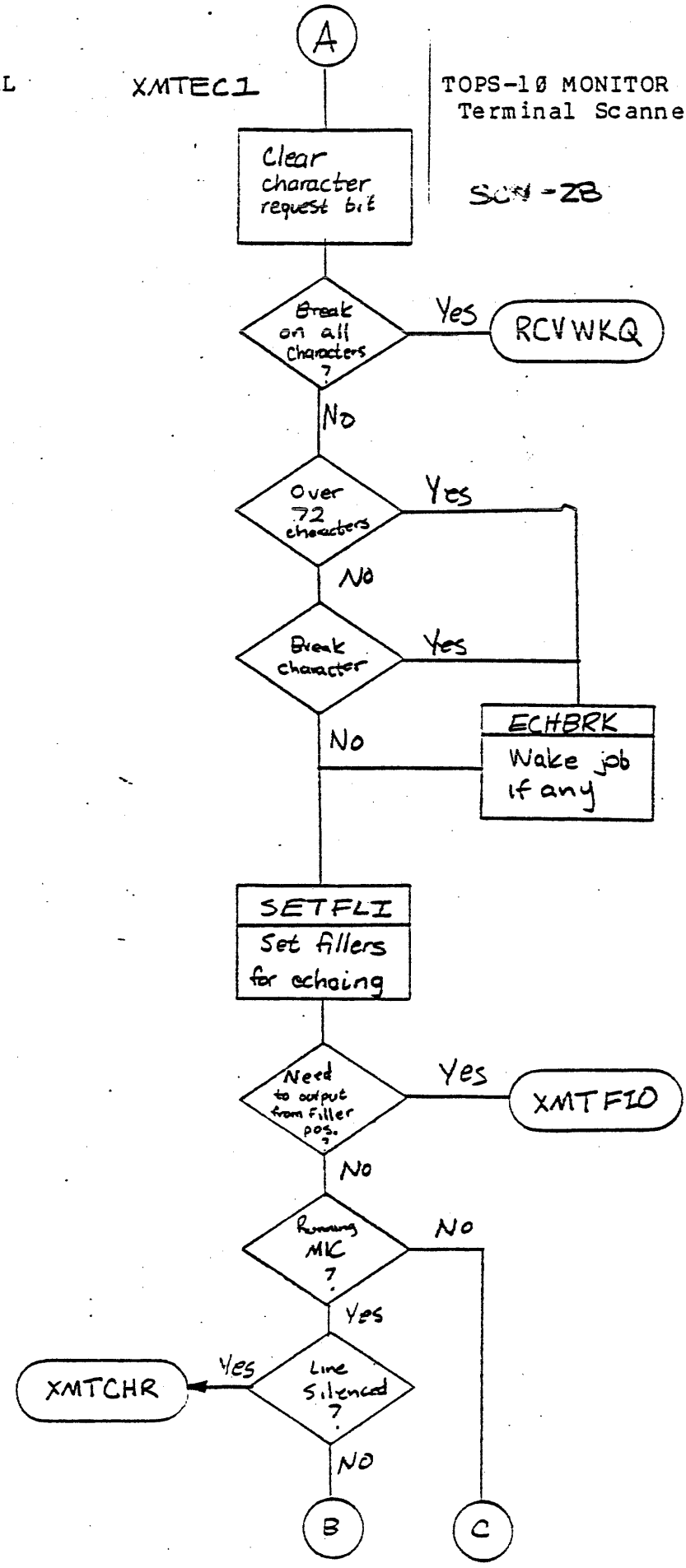
SCN-26

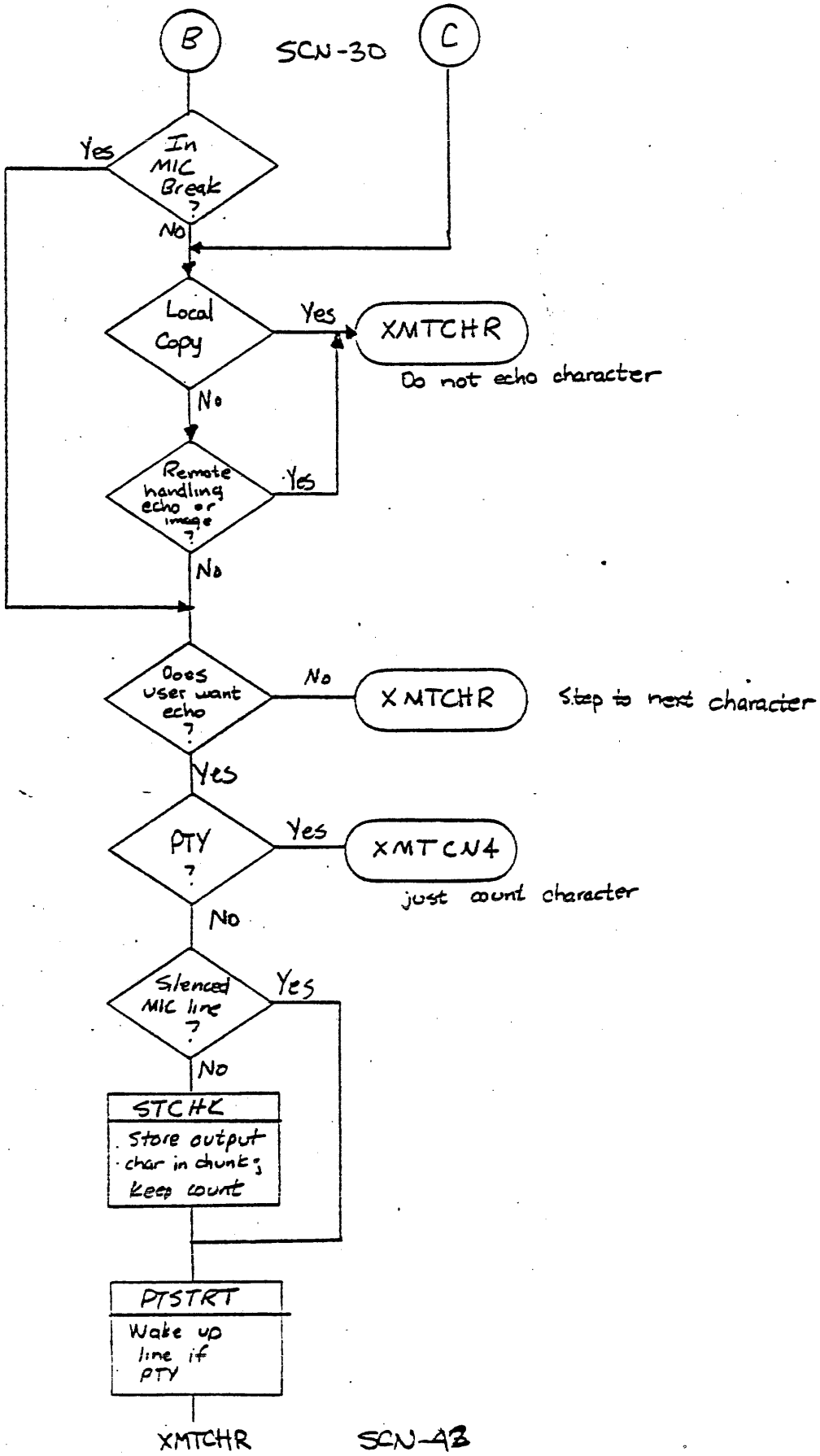


SCN-27

Routine to echo a character when buffered and filler output done ; called at clock level.



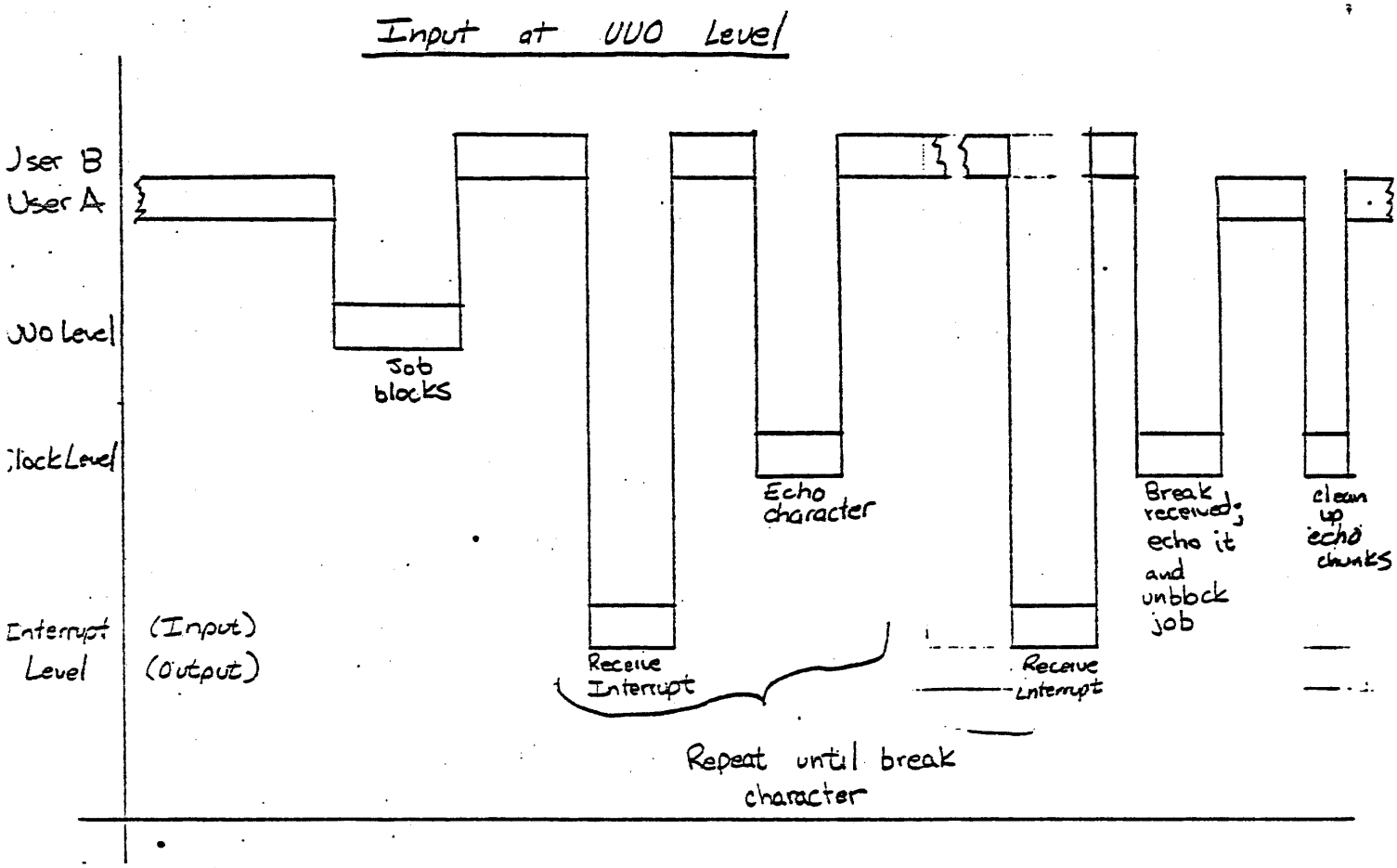

A
Clear character request bit
Break on all characters ?
Yes
RCVWKQ
No
Over 72 characters
Yes
No
Break character
Yes
No
ECHBRK
Wake job if any
SETFLI
Set fillers for echoing
Need to output from Filler pos. ?
Yes
XMTFIO
No
Running MIC ?
No
Yes
Line Silenced ?
Yes
XMTCHR
NO
B
C


B
SCN-30
C
In MIC Break ?
Yes
No
Local Copy
Yes
XMTCHR
Do not echo character
No
Remote handling echo or image ?
Yes
No
Does user want echo ?
No
XMTCHR
Step to next character
Yes
PTY ?
Yes
XMTCN4
just count character
No
Silenced MIC line ?
Yes
No
STCHC
Store output char in chunk; keep count
PTSTRT
Wake up line if PTY
XMTCHR

The next question is: where in this cycle will a break character be detected and the job brought out of TI wait so that it may run? The answer: in XMTECH, if a break character is received, a call is made to ECHBRK which will wake the job. ECHBRK eventually calls STTIOD (Start TTy I/O Done) in CLOCK1.

The complete timing of the process now looks like this:

SCN-30



There are two other cases that must be mentioned in this basic discussion. One is the case of input going to COMCON for command processing instead of to the user. The job will be in monitor mode and not in user mode. This is also handled in ECHBRK. If the command is waiting in command wait, the job will not be woken up but instead the command bit will be turned on for detection during a future clock tick.

The other case is when a user issues a INCHRW which only waits for one character. In this case, all characters are break characters. XMTECH will handle this condition at XMTEC1, calling RCVWAK to revive the job if necessary.

DATA STRUCTURES REVISITED

Now that the fundementals of terminal I/O have been presented, a closer look at the data structures is necessary. In particular, the LDB pointers to the chunks must be discussed. The are 10 words in the LDB that are important:

LDBTIP - where to put characters into the input buffer
LDBTIT - where to take characters from the input buffer
LDBTIC - the count of the echoed characters in the input buffer
LDBBKC - the count of the break characters in the input buffer
LDBTOP - where to put characters in the output buffer
LDBTOT - where to take characters from the output buffer
LDBTOC - the count of the characters in the output buffer
LDBECT - where to take characters from for echoing
LDBECC - count of characters to echo
LDBBKU - the lcoation of the last break character in the input buffer

These words handle the input buffers, output buffers and echoing. When the system is restarted, all the words are cleared. The use of these words can best be illustrated by showing various cases.

CASE I - No input

When the line is idle, all the words are zero.

Case I - No input or output.

No Chunks Allocated

LDB

Input	
LDBTIT	0
LDBTIP	0
LDBTIC	0
Break Characters	
LDBBKU	0
LDBBKC	0
Output	
LDBTOT	0
LDBTOP	0
LDBTOC	0
Echoing	
LDBECT	0
LDBECC	0

LDBTIT - where to take characters from input buffer

LDBTIP - where to put characters into input buffer

LDBTIC - count of echoed chars in input buffer

LDBBKU - pointer to last break character

LDBBKC - count of break characters in input buffer

LDBTOT - where to get characters from output buffer

LDBTOP - where to put characters into output buffer

LDBTOC - count of characters in output buffer

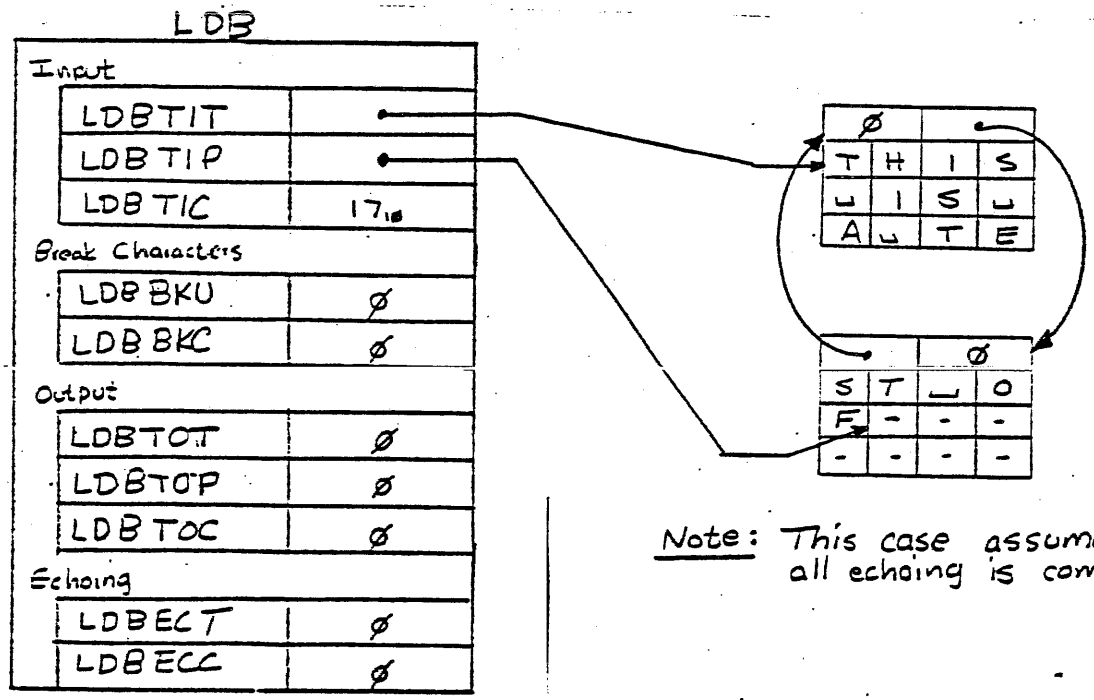
LDBECT - where to take characters from for echoing

LDBECC - count of characters to echo

CASE II - Less than one line typed

Here, the user has been typing without hitting a break character. The drawing shows how LDBTIT points to the first character and LDBTIP to the first free slot. Assuming that all characters have been echoed, LDBTIC will be 17, equal to the number of received characters. All other words are zero.

Case II - Less than one line typed (no break character).



Note: This case assumes all echoing is complete.

Text

THIS␣IS␣A␣TEST␣OF

LDBTIT - where to take characters from input buffer

LDBTIP - where to put characters into input buffer

LDBTIC - count of echoed chars in input buffer

LDBBKU - pointer to last break character

LDBBKC - count of break characters in input buffer

LDBTOT - where to get characters from output buffer

LDBTOP - where to put characters into output buffer

LDBTOC - count of characters in output buffer

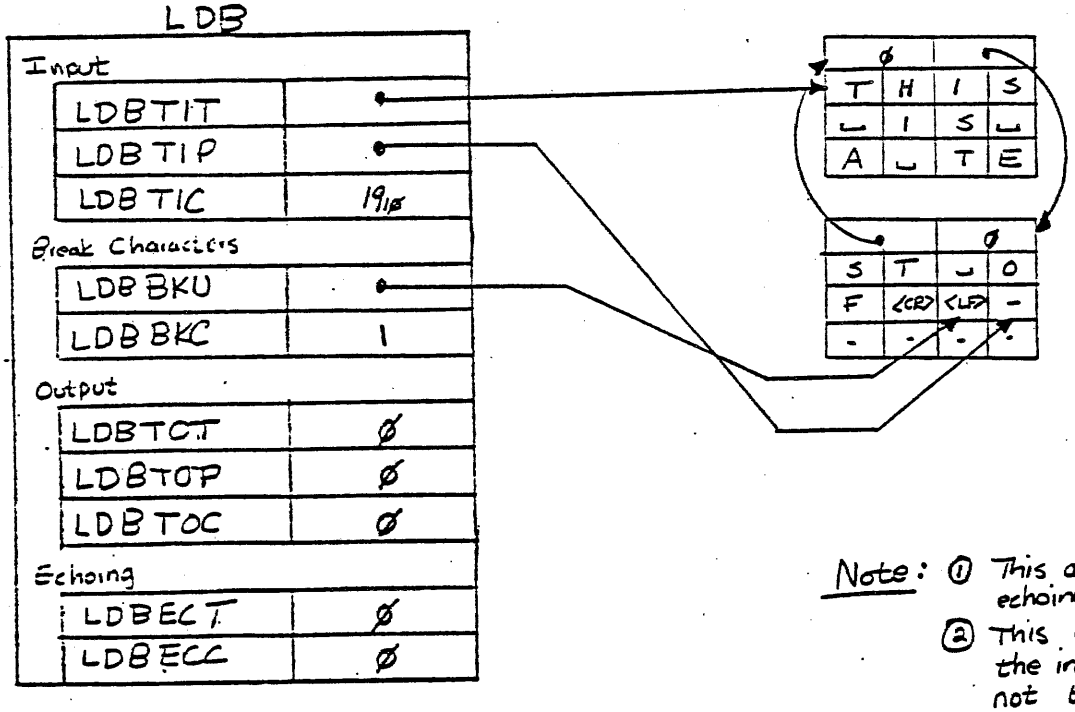
LDBECT - where to take characters from for echoing

LDBECC - count of characters to echo

CASE III - One Line Typed

This case is the same as case 2 except that a carraige return has been typed. The monitor will fill in a line feed for both echoing and in the input buffer. LDBBKU will point to the line feed. If any further typing follows, LDBBKU becomes especially important because CONTROL/U, CONTROL R, CONTROL/W and DELETE will not back up past the character pointed to by LDBBKU. Note that carraige return is not considered a break character, line feed is.

Case III - One complete line has been typed.



LDBTIT - where to take characters from input buffer

LDBTIP - where to put characters into input buffer

LDBTIC - count of echoed chars in input buffer

LDBBKU - pointer to last break character

LDBBKC - count of break characters in input buffer

LDBTOT - where to get characters from output buffer

Note: ① This assumes all echoing is complete
② This assumes that the input line has not been processed.
③ Carriage return is not considered a break character; line feed is.

Text:

THIS␣IS␣A␣TEST␣OF <CR>

LDBTOP - where to put characters into output buffer

LDBTOC - count of characters in output buffer

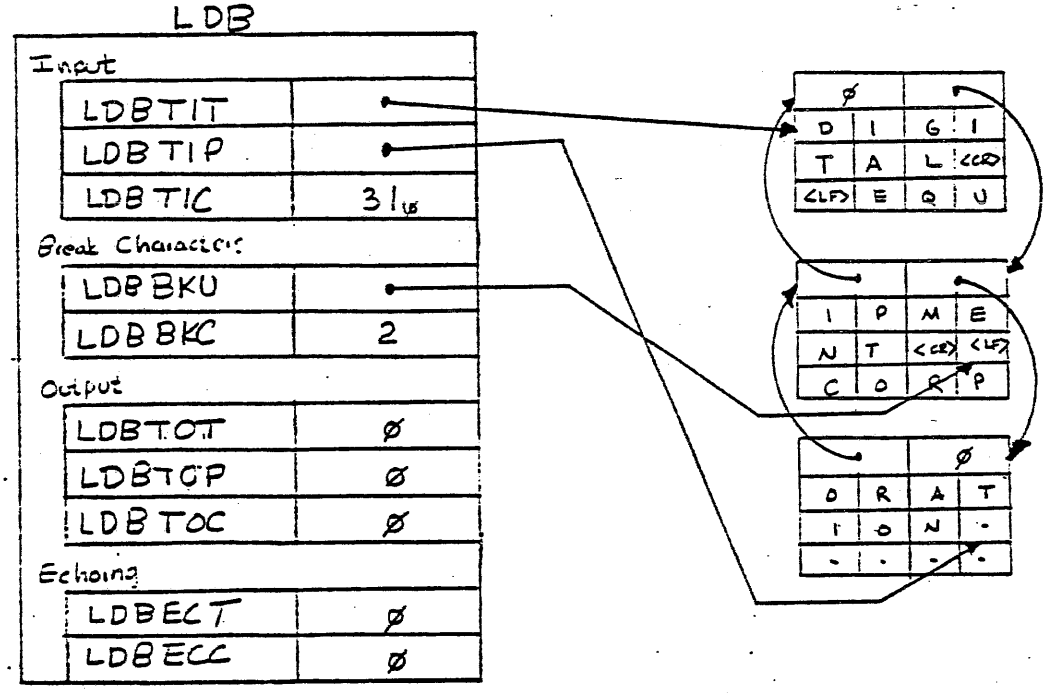
LDBECT - where to take characters from for echoing

LDBECC - count of characters to echo

CASE IV - Several lines of type ahead

This example is similar to the previous one except that there is just more input. The pointers and counts are adjusted accordingly.

Case II - More than one line has been typed and no processing has been performed.



Note : ① Echoing is complete
② This is a type ahead situation

LDBTIT - where to take characters from input buffer

LDBTIP - where to put characters into input buffer

LDBTIC - count of echoed chars in input buffer

LDBBKU - pointer to last break character

LDBBKC - count of break characters in input buffer

LDBTOT - where to get characters from output buffer

LDBTOP - where to put characters into output buffer

LDBTOC - count of characters in output buffer

LDBECT - where to take characters from for echoing

LDBECC - count of characters to echo

Text :

DIGITAL <CR>
EQUIPMENT<CR>
CORPORATION

CASE V - Incomplete Echoing for Several lines of input

This is a more complex case. The same text as case 4 has been typed but echoing is incomplete. The user has typed:

```
DIGITAL<CR>
EQUIPMENT<CR>
CORPORATION
```

but all that has been echoed is:

```
DIGITAL<CR><LF>
EQ
```

The characters

```
  UIPMENT<CR><LF>
CORPOR
```

are waiting in the output buffer and "ATION" must still be echoed (placed in the output buffer).

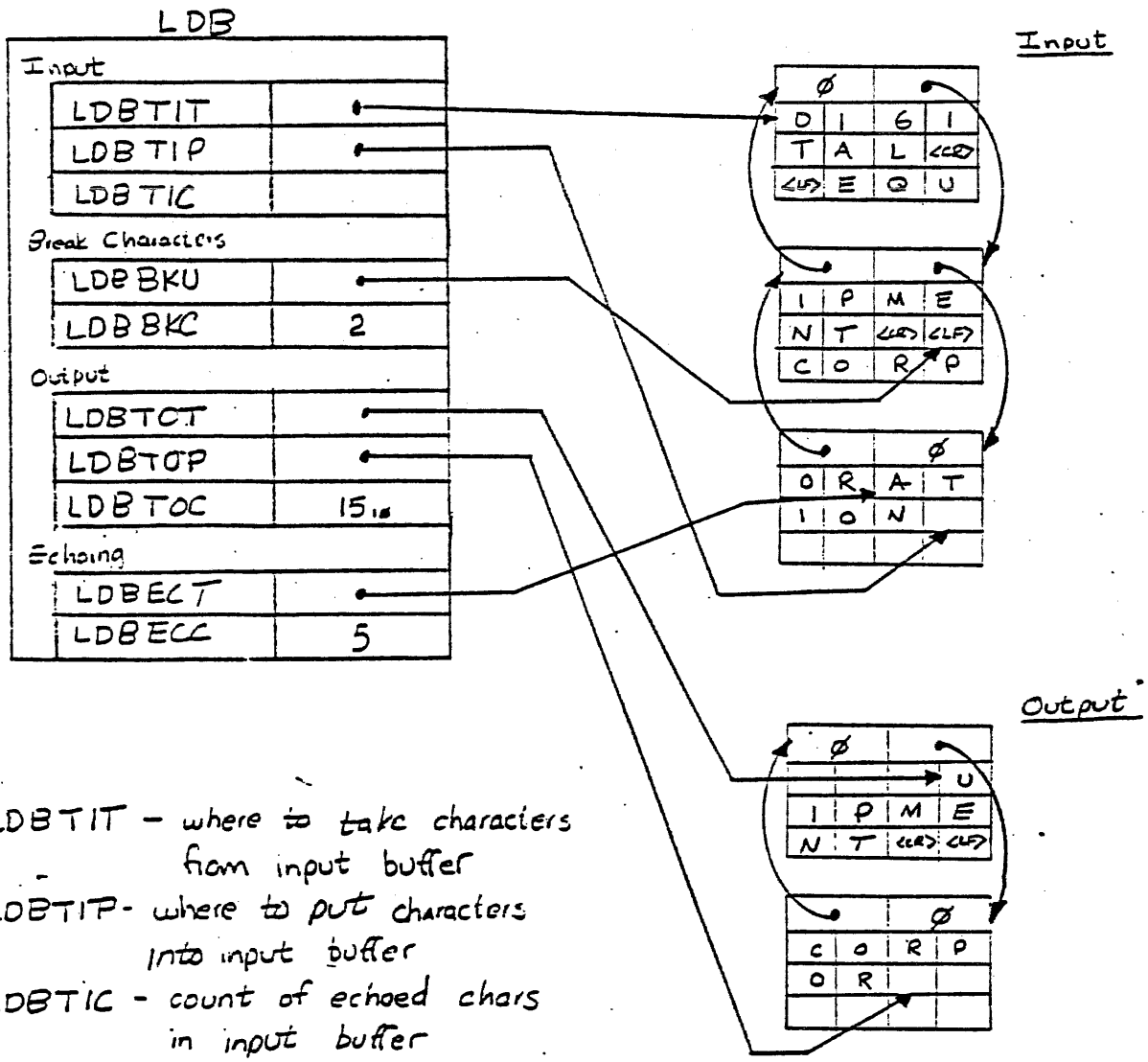
Note also:

1. LDBTIC reflects only those characters that have been echoed, not the total number of characters input. Once the copy of an input character is placed in the output buffer, it is considered as having been echoed.

2. LDBECT points to where echoing must continue.

3. LDBTOP and LDBTOT function similar to LDBTIP and LDBTIT.

Case V Type Ahead with echoing incomplete.



LDBTIT - where to take characters from input buffer

LDBTIP - where to put characters into input buffer

LDBTIC - count of echoed chars in input buffer

LDBBKU - pointer to last break character

LDBBKC - count of break characters in input buffer

LDBTOT - where to get characters from output buffer

LDBTOP - where to put characters into output buffer

LDBTOC - count of characters in output buffer

LDBECT - where to take characters from for echoing

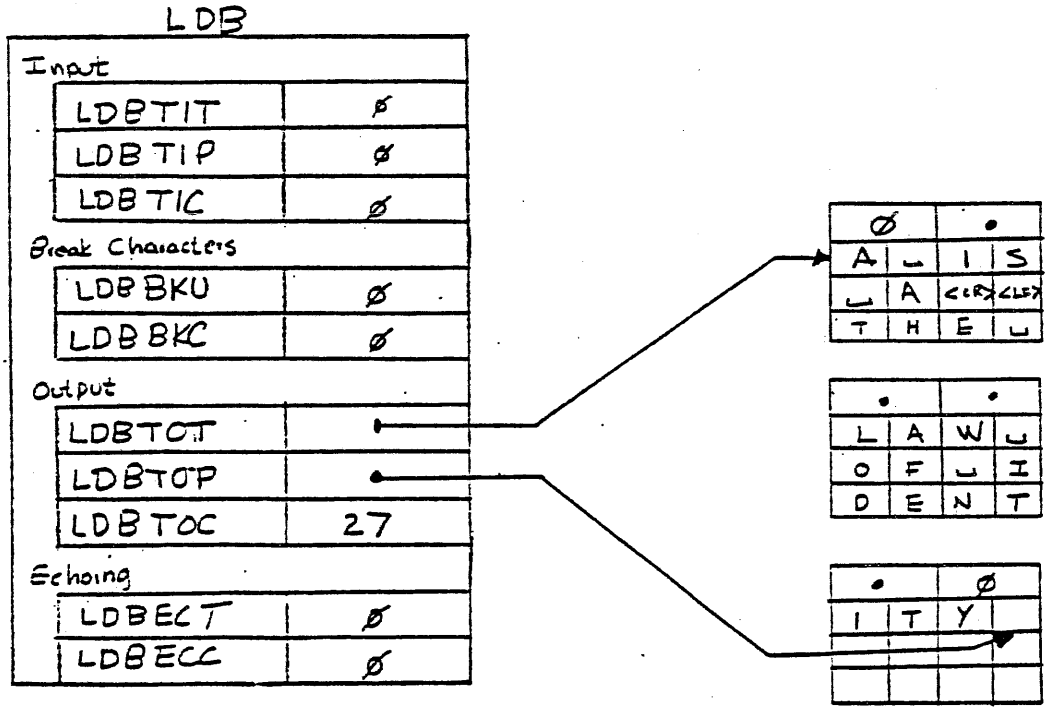
LDBECC - count of characters to echo

CASE VI - Simple Output

Only the output pointers and counts are used when a OUT or OUTSTR TTCALL is issued by a program.

Case 6 - Output

SCN-36



LDBTIT - where to take characters from input buffer

LDBTIP - where to put characters into input buffer

LDBTIC - count of echoed chars in input buffer

LDBBKU - pointer to last break character

LDBBKC - count of break characters in input buffer

LDBTOT - where to get characters from output buffer

LDBTOP - where to put characters into output buffer

LDBTOC - count of characters in output buffer

LDBECT - where to take characters from for echoing

LDBECC - count of characters to echo

To Be Printed

A ␣IS␣A <CR><LF>
THE␣LAW␣OF␣IDENTITY

CASE VII Mixed I/O

In this case, the user has typed a line while the program was outputting, scrambling the screen to a certain extent. The program typed

```
BOOKS BY AYN RAND
```

Then the user typed "GOOD B". Before the user could type more, the program printed

```
ATLAS SHRUGGED
THE FOUNTAINHEAD
```

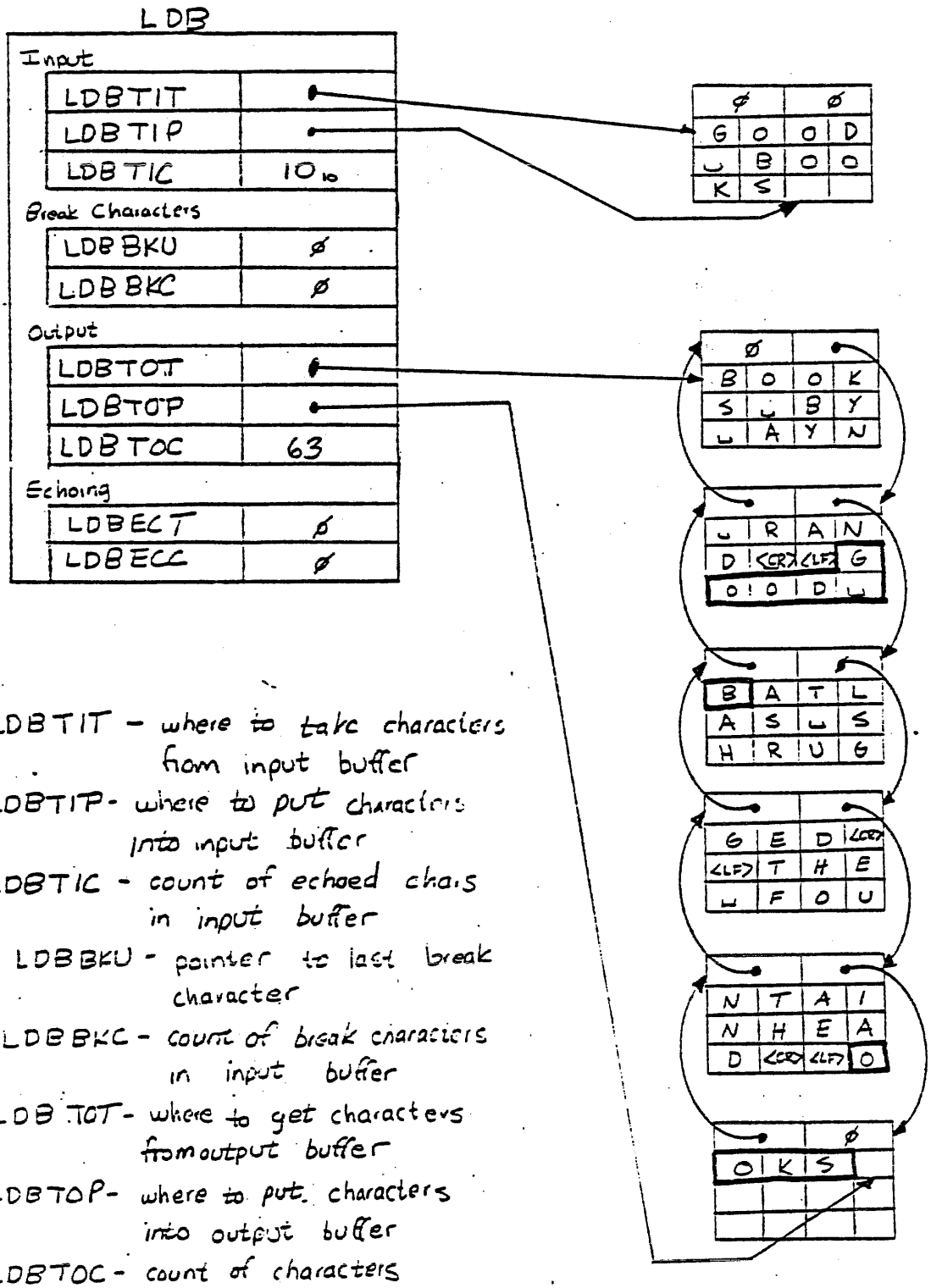
The remainder of the user line "OOKS" was then echoed.

The user would see on his terminal:

```
BOOKS BY AYN RAND<CR><LF>
GOOD BATLAS SHRUGGED<CR><LF>
THE FOUNTAINHEAD<CR><LF>
OOKS
```

Case VII Mixed I/O

SCN-37



LDBTIT - where to take characters from input buffer

LDBTIP - where to put characters into input buffer

LDBTIC - count of echoed chars in input buffer

LDBBKU - pointer to last break character

LDBBKC - count of break characters in input buffer

LDBTOT - where to get characters from output buffer

LDBTOP - where to put characters into output buffer

LDBTOC - count of characters in output buffer

LDBECT - where to take characters from for echoing

LDBECC - count of characters to echo

CONTROL CHARACTER HANDLING

All input characters are checked to see if they require special handling (except when the terminal is in image mode or packed image mode). The test is made by indexing into a table, CHTABL, with the octal value of the character. Each entry in CHTABL contains definition bits in the LH and a routine address in the RH. If the RH is zero, there is no special handling routine. For a definition of the bits in the LH, see the monitor table descriptions. The routines for special handling are:

RINUL	Null character
RICA	CONTROL/A (MIC)
RICB	CONTROL/B (MIC)
RICC	CONTROL/C
RICD	CONTROL/D (MIC)
RIBSP	BACKSPACE
RICM	CONTROL/M (Carraige Return)
RICO	CONTROL/O
RICP	CONTROL/P
RICQ	CONTROL/Q
RICR	CONTROL/R
RICS	CONTROL/S
RICT	CONTROL/T
RICU	CONTROL/U
RICDEL	DELETE and CONTROL/W
RIALT	ALTMODE

A brief discussion of some of the routines now follows.

CONTROL/O

When CONTROL/O is typed, all output to the terminal is suppressed until the terminal needs input, a CONTROL/C is typed or another CONTROL/O is typed. This function is controlled by the LDROSU (output suppress) bit in the LDBDCH word in the LDB. When on, all output will not be sent.

The routine to handle CONTROL/Os is RICO in SCNSER. It performs three functions; 1) complement the LDROSU bit, 2) clear the output buffer and 3) echo "O" on the terminal. Successive CONTROL/Os act as a toggle for LDROSU.

The initial CONTROL/O clears any pending output. Any further output will be intercepted in the TYO routine where output is discarded when LDROSU is turned on.

CONTROL/C

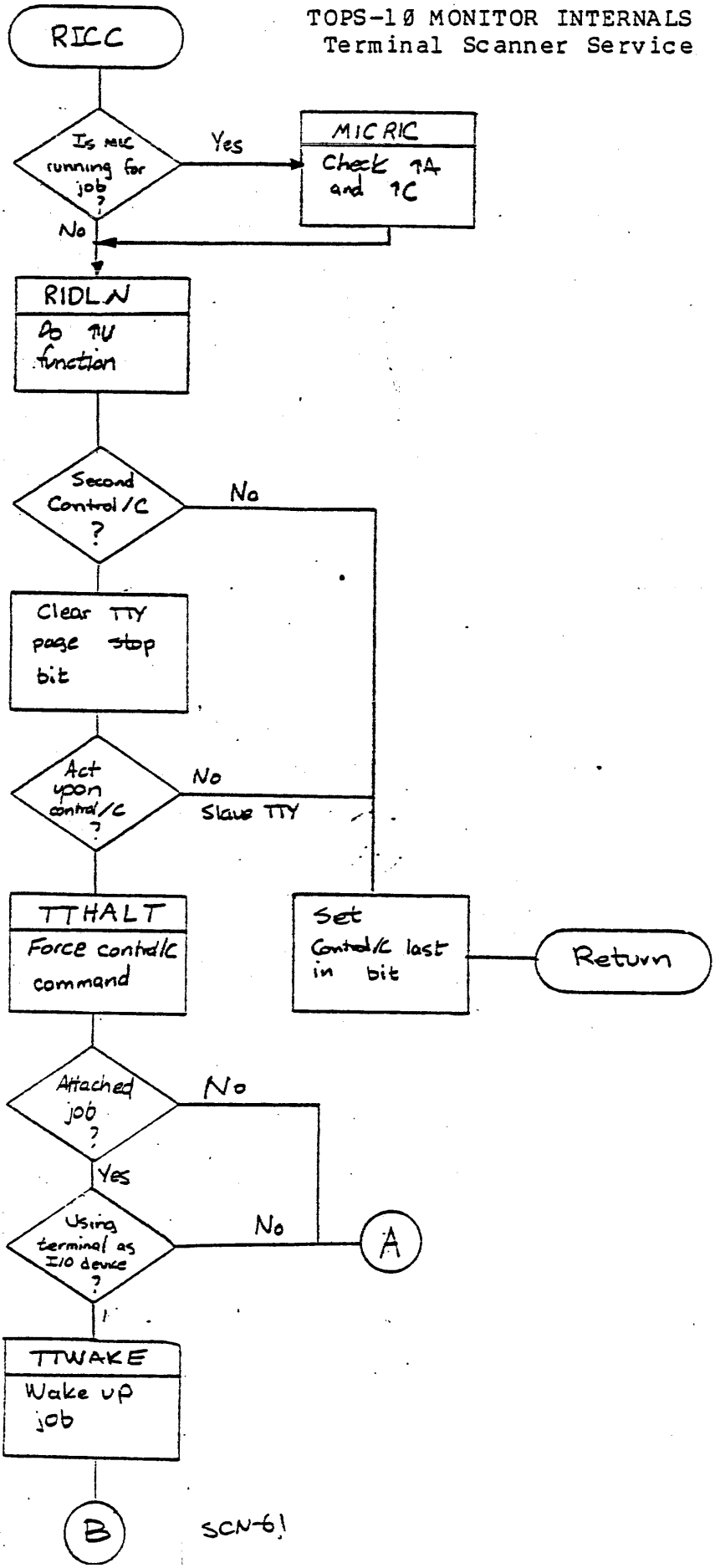
A CONTROL/C interrupts the currently running program and returns the terminal to monitor mode. It also causes the input line, back to the last break character, to be deleted (equivalent to a CONTROL/U). Two CONTROL/Cs must be typed if the program is in the middle of execution.

In order to return the job to monitor mode, a forced command is used. A forced command is used instead of rescheduling because a CONTROL/C implies that the user wants his job stopped immediately and does not want to wait for rescheduling. Two CONTROL/Cs also have the effect of clearing both the input and output buffers for a terminal.

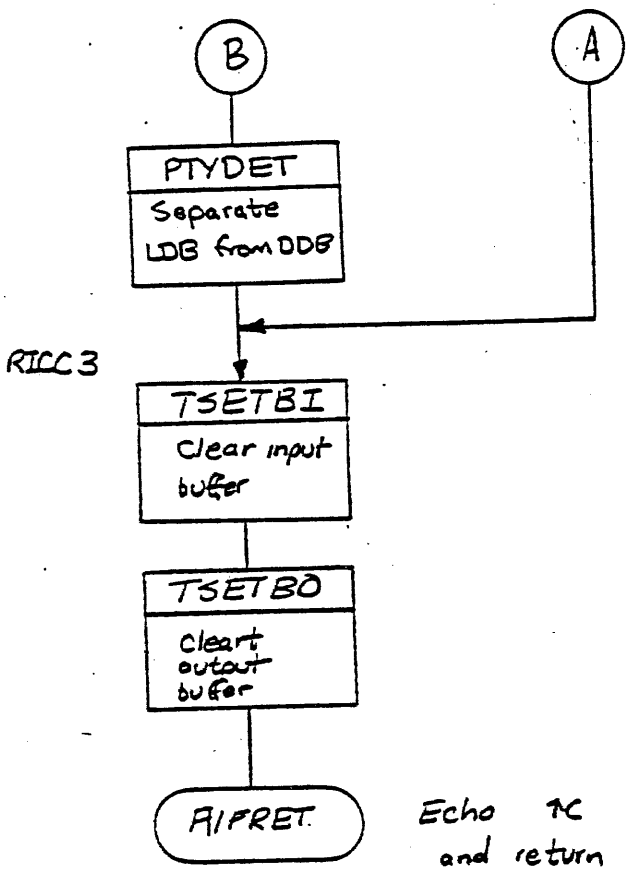
In order to distinguish between the first and second CONTROL/C, the L2LCCS bit in LDBBY2 is used. It is set with the first CONTROL/C and cleared with the second.

DIGITAL

TOPS-10 MONITOR INTERNALS
Terminal Scanner Service



SCN-39


B
A
PTYDET
Separate LDB from DDB
RICC3
TSETBI
Clear input buffer
TSETBO
Clear output buffer
AIFRET
Echo ^C and return

CONTROL/D

CONTROL/D has a special purpose that is used when debugging the monitor. The code to handle CONTROL/D is located in the low segment of the monitor unlike the rest of SCNSER. It is placed there so that breakpoints may be set. The high segment of the monitor is genereally write locked and can not be modified. Therefore, one easy to always gain control is to set a breakpoint in RICD while in EDDT and then type a CONTROL/D. Control will always pass through that point, stopping the monitor.

CONTROL/T

CONTROL/T performs the same function as the USESTA monitor command, reporting various user statistics. Both USESTA and CONTROL/T use the USESTA forced command to force the report. The forced command is used so that the amount of interrupt level code is reduced and so that the user must pay for his own command.

CONTROL/R

Typing a CONTROL/R will cause the system to retype the current input line, processing all deletes. Like CONTROL/T, CONTROL/R is implemented as a forced command so that some processing may be eliminated from interrupt level. The routine RETYPE handles the retyping. It will find the last break character in the buffer (pointed to by LDBBKU) and print every character from that point on.

CONTROL/Q and CONTROL/S

These two control characters allow terminal output to be selectively started or stopped by a user. CONTROL/Q stops all output (including echoing); CONTROL/S starts output. This feature operates only if the TERMINAL PAGE command has been enabled. CONTROL/C will issue an automatic CONTROL/Q.

The routines to handle CONTROL/Q and CONTROL/S are in SCNSER; the output is stopped or started in the front end. This requires queued protocol messages to be sent to the

front end.

When a CONTROL/S is typed, the RICS routine in SCNSER will turn on two bits in the LDB; LDLSTP (to indicate that output is to be stopped) and L1RCHP (to announce the need to change hardware parameters). The LDB of the line is then placed in the output request queue. At clock level, TTDSTO will be called to remove the entry from the LDB queue. A test is made to see if L1RCHP is set and if so, control passes to TTDCHP (also in TTDINT). There, the need for a CONTROL/S is uncovered and the appropriate queued protocol message is sent.

Output will then accumulate at the front end until CONTROL/Q is typed. The RICQ routine will then follow the same procedure as CONTROL/S to send the XON message to the front end.

TERMINAL IMAGE MODES

These two data modes for terminals have been implemented so that users may bypass most of SCNSER's character processing. They are for use with special devices or situations. Special characters will be treated as any regular character. Neither mode can be used with PTYs.

Image Mode

When image mode is used, every input character (eight bits) is stored right justified in a PDP-10 word and causes an interrupt. There are no break characters. The terminal must first be ASSIGN'ed and INITed before it can be used.

The image input state begins when the program starts waiting as a result of an INPUT in image mode. It ends when a program executes any non-image mode terminal output operation. If no input characters are received for ten seconds, the monitor forces an EOF. After another ten seconds, the monitor terminates the image input state and simulates a CONTROL/C. Characters will be stored directly in the user input buffer, one character per word.

Packed Image Mode (PIM)

PIM is designed for high efficiency character throughput between programs and external devices, accomplished by minimizing character manipulation and testing. In packed image mode, characters are maintained as 8-bit quantities (7 data bits and 1 parity bit). These 8-bit characters are stored in user buffers at the rate of four per word. A program may define from one to four break characters for each line using the TRMOP. monitor call.

When the monitor receives a charcter via an IN or INPUT, the character is compared to each field in the break set. If no match is found, the character is put in the buffer and the interrupt is dismissed. In the case where a match does occur, the character is put into the buffer. The input wait is then terminated and the controlling program awakened. To avoid the poossibility of a terminal getting stuck in PIM mode and to allow for the case where your program wishes to be awakened on each character, a program can specify an empty (Ø,,Ø) break set. In general operation, all characters, including control characters, are passed by PIM with no monitor intervention with the following exception: if page mode is set, the characters CTRL/S (XOFF) and CTRL/Q (XON) react the same as for normal page mode.

Half Duplex Terminals

Conceptually, terminals can be considered as two separate devices; a keyboard for input and a screen or printer for output. Most terminals can transmit (input) and receive (output) data over the same line at the same time. This is known as a full duplex line. Half duplex terminals can send and receive data over the same line but not at the same time. When transmission occurs in one direction, the receiver cannot send until the incoming message is finished.

In order to regulate the flow of messages, the following convention is used. When either side of the link wants to send a message over an idle half duplex line, it sends a control message to the receiver. When an ACK is sent back, the message is started. This is all accomplished at the hardware level between the terminal and the device controller.

Little must be done at the software level to handle half duplex lines. In fact, the only code in SCNSER for half duplex is concerned with error checking. At RECHDX, a test is made to see if a receive interrupt has occured while output is in progress. If so, the line is not functioning as a half duplex line.

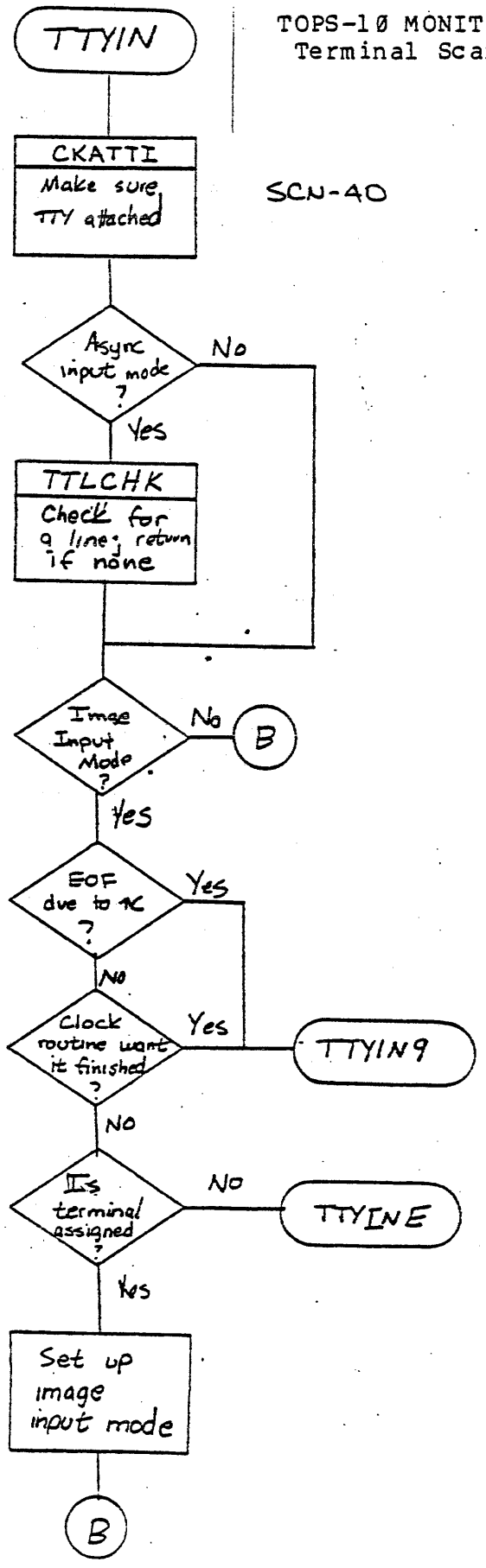
OTHER TERMINAL MONITOR CALLS

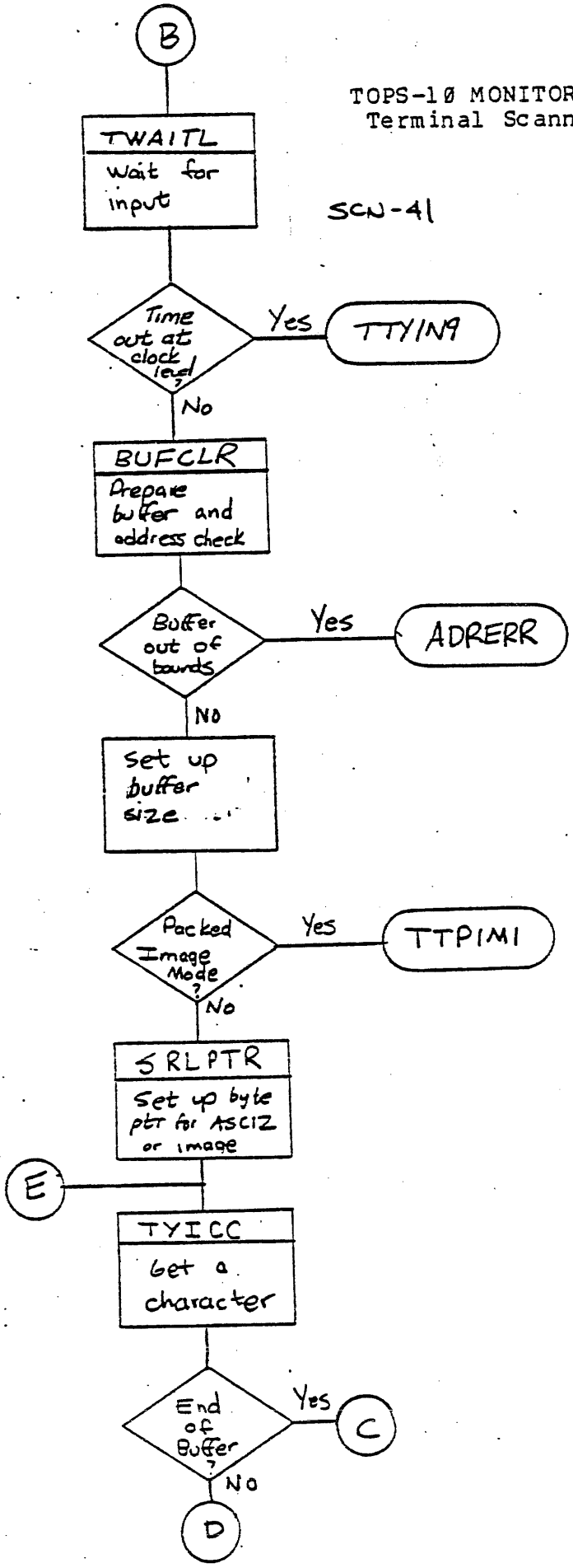
Terminal I/O can be performed using IN and OUT monitor calls besides the simpler TTCALL mecchanism. The INs and OUTs obey the same structure as was mentioned in the chapter on I/O processing; data is read in and out of user buffers that are pointed to by a ring buffer header. The header must be setup using an INIT or OPEN monitor call. The code for the monitor calls is necessarily larger (to accommodate more argument checking) but has many subroutines in common with TTCALLS.

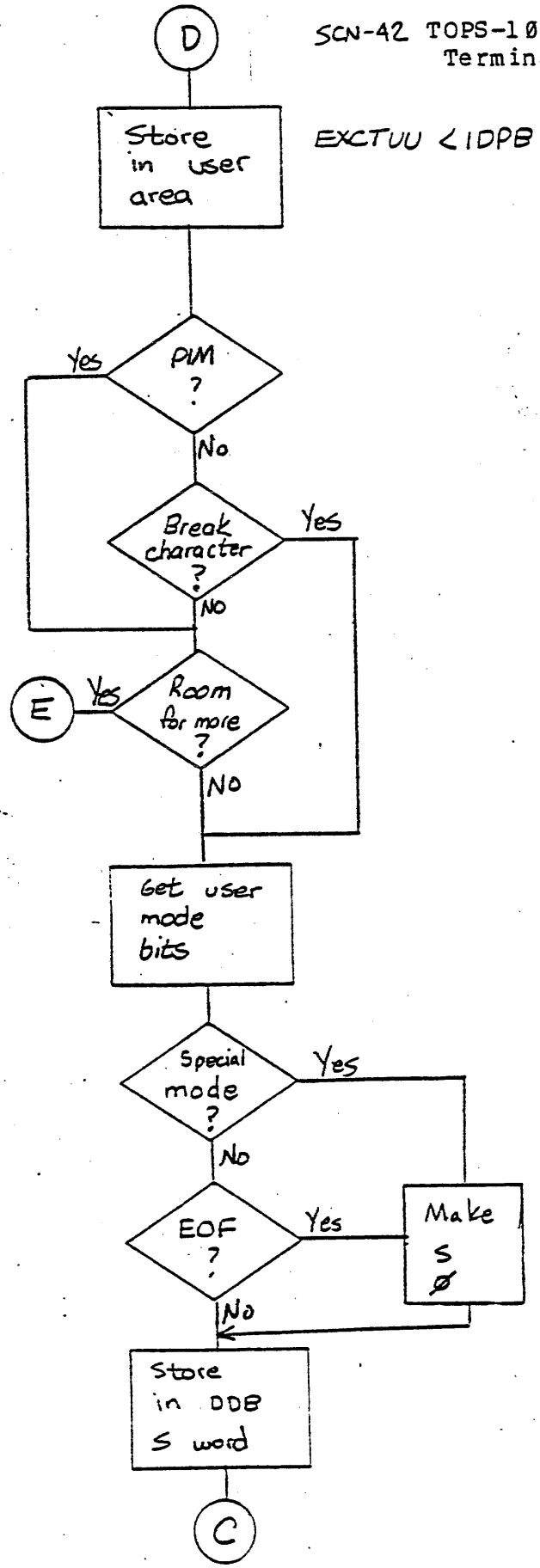
TTYOUT (in SCNSER) is the routine that performs the OUT and OUTPUT monitor call. It uses the saeme output routine (TYO) to enter data in the chunks as do TTCALLs. Since packed image mode and image mode use OUTPUTs, extra code is in TTYOUT to handle this special case. Extra code is also necessary for the proper stoarge of data in the user buffers.

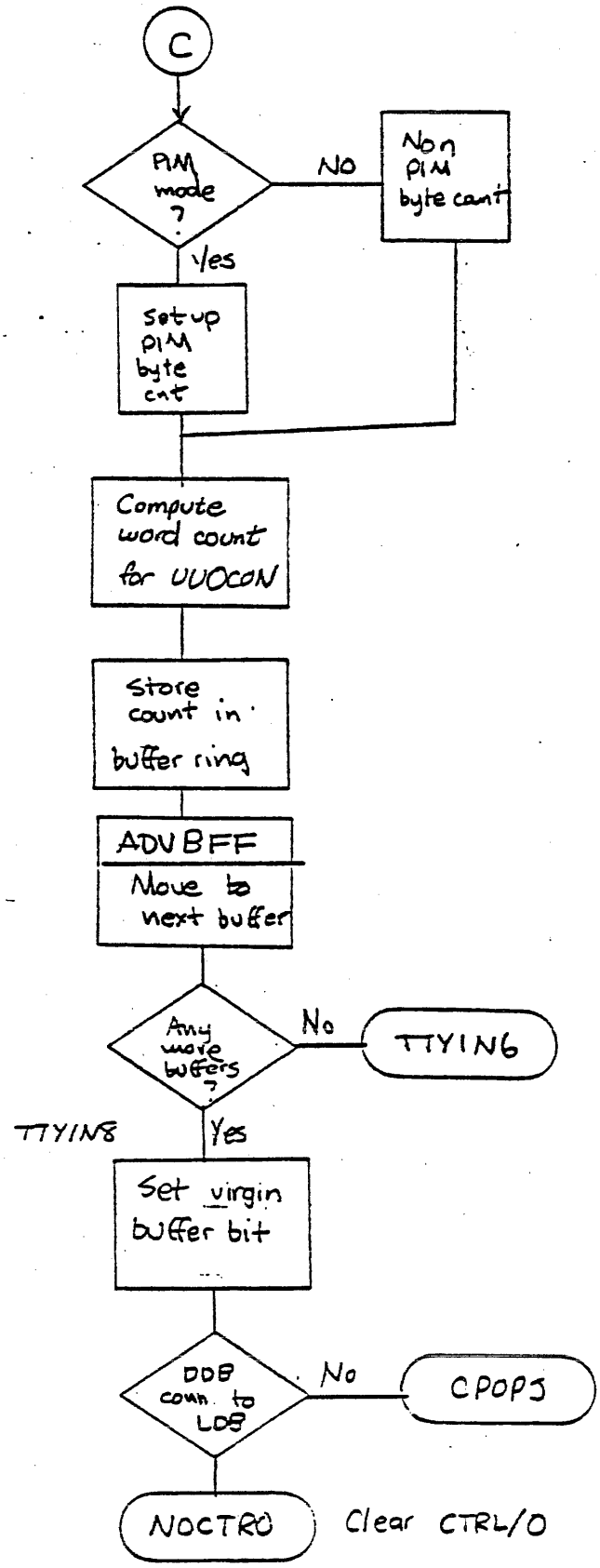
TTYIN handles IN and INPUT monitor calls. Like TTCALLS, they use the TWAITL routine to wait for input and will return when a break character is detected. Data will be read out of the chunks and stored in the user buffer.

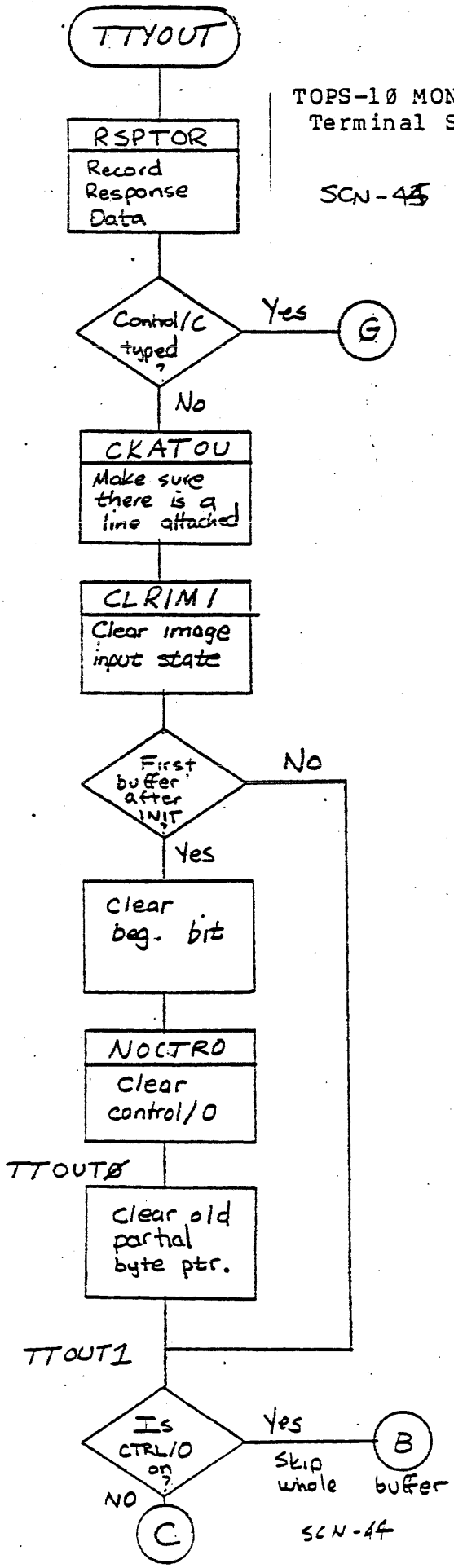
TRMOP. monitor calls are useful for reporting/setting terminal characteristics and for special terminal functions. They use the same exact routines as the TTCALLs for outputting characters or groups of lines.


TTYIN
CKATTI
Make sure TTY attached
SCN-40
Async input mode ?
No
Yes
TTLCHK
Check for a line; return if none
Image Input Mode ?
No
B
Yes
EOF due to ^C ?
Yes
No
Clock routine want it finished ?
Yes
TTYIN9
No
Is terminal assigned ?
No
TTYINE
Yes
Set up image input mode
B


B
TWAITL
wait for input
SCN-41
Time out at clock level?
Yes
TTYIN9
No
BUFCLR
Prepare buffer and address check
Buffer out of bounds
Yes
ADRERR
No
Set up buffer size
Packed Image Mode?
Yes
TTPIMI
No
SRLPTR
Set up byte ptr for ASCIZ or image
E
TYICC
Get a character
End of Buffer?
Yes
C
No
D

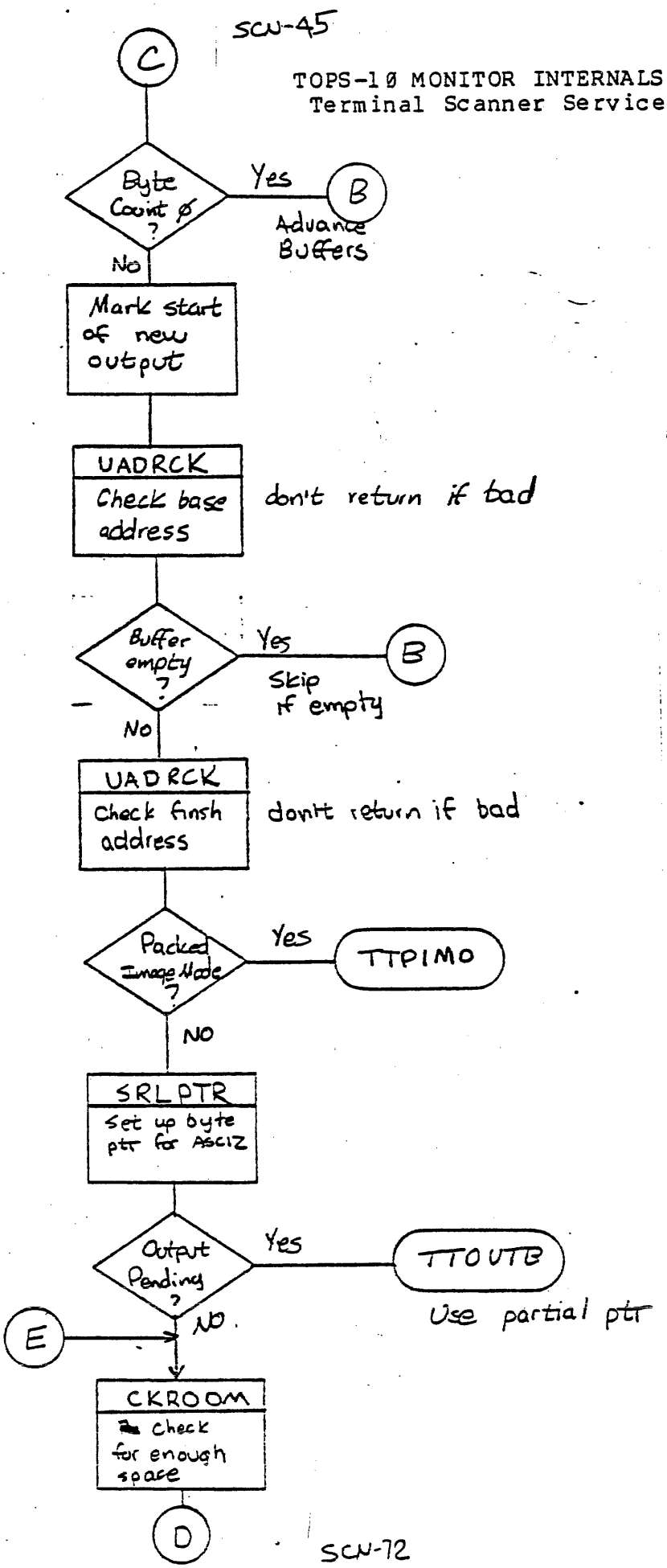

D
Store in user area
EXCTUU <IDPB P3,P1>
PIM ?
Yes
No
Break character ?
Yes
No
E
Yes
Room for more ?
No
Get user mode bits
Special mode ?
Yes
No
EOF ?
Yes
Make S Ø
No
Store in DDB S word
C


C
PIM mode ?
No
Non PIM byte cant
Yes
Set up PIM byte cnt
Compute word count for UUOCON
Store count in buffer ring
ADVBFF
Move to next buffer
Any more buffers ?
No
TTYIN6
TTYIN8
Yes
Set virgin buffer bit
DDB conn to LOG
No
CPOPJ
NOCTRO
Clear CTRL/O


TTYOUT
RSPTOR
Record Response Data
Control/C typed ?
Yes
G
No
CKATOU
Make sure there is a line attached
CLRIM1
Clear image input state
First buffer after INIT ?
No
Yes
Clear beg. bit
NOCTRO
Clear control/O
TTOUTØ
Clear old partial byte ptr.
TTOUT1
Is CTRL/O on ?
Yes
Skip whole buffer
B
NO
C

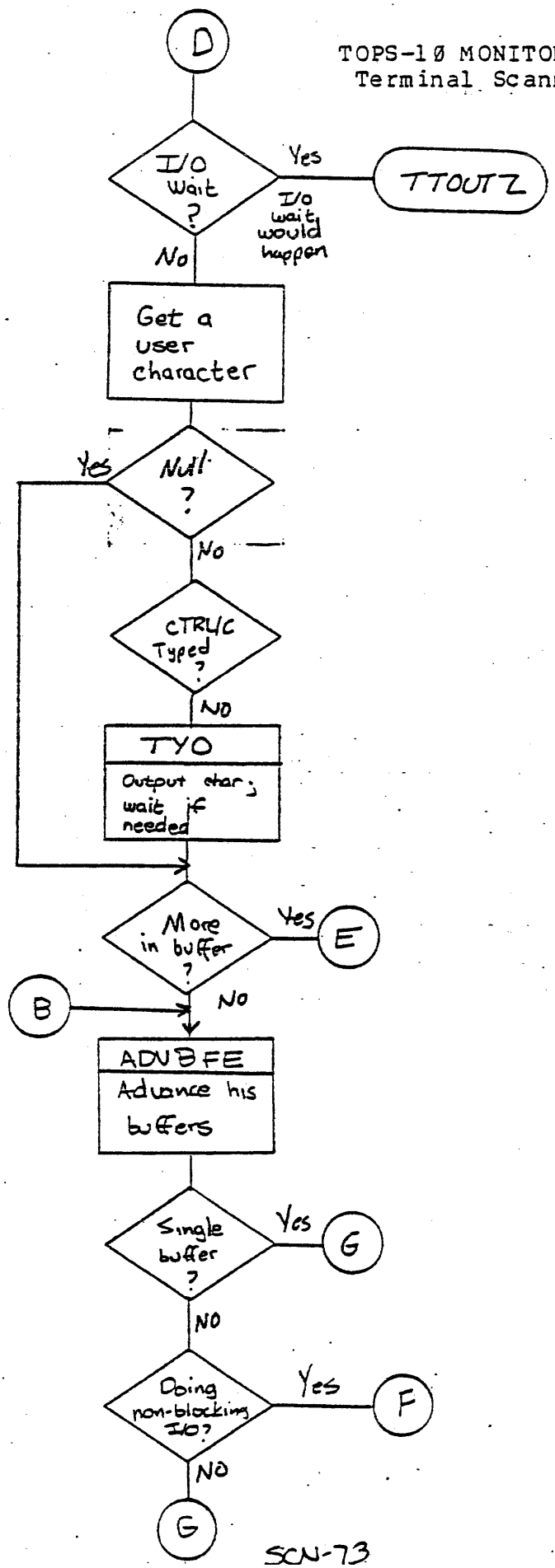
DIGITAL

TOPS-10 MONITOR INTERNALS
Terminal Scanner Service

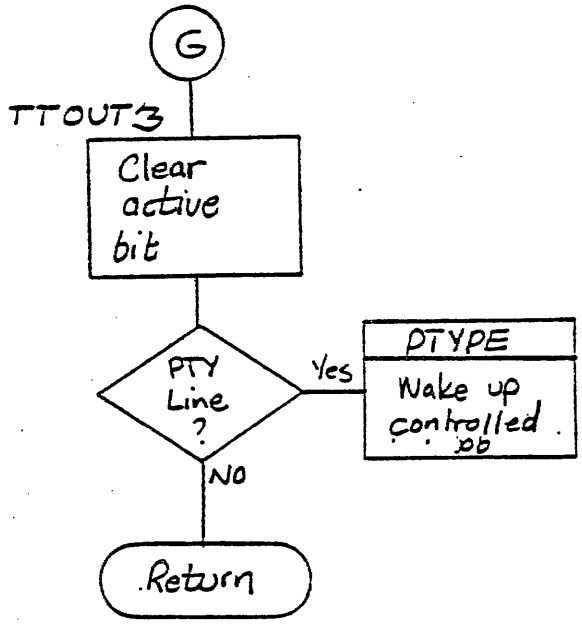


DIGITAL

TOPS-10 MONITOR INTERNALS
Terminal Scanner Service



SCN-47

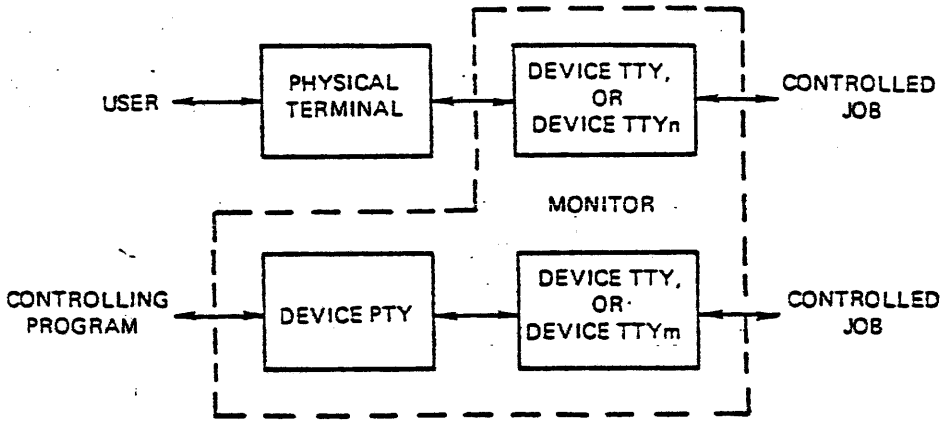

G
TTOUT3
Clear active bit
PTY Line ?
Yes
PTYPE
Wake up controlled job
NO
Return

PSEUDO-TERMINALS (PTYs)

A pseudo-TTY (or PTY) is a simulated terminal that allows a job to be initiated by a program instead of a user. The program sends commands to and receives typeout from the controlled job. The PTY is the monitor's method of allowing the connection.

The controlling program uses the PTY in the same way a user uses a physical device. It initiates the PTY, inputs to and waits for output from the PTY, and closes the PTY using the appropriate monitor calls. The job controlled by the program performs I/O to the PTY as though the PTY were a physical terminal.

SCN-48



The controlling job cannot wait for PTY I/O because it may be controlling several PTYs or the controlled job may go into a loop. To solve this problem, the HIBER monitor call has been modified so that when the controlling job HIBERS, any activity by the controlled job will wake it.

When a controlling program opens a PTY, its DDB reflects that fact. The controlled job, however, thinks it is connected to a regular TTY. Its DDB is the regular TTY DDB. The important difference between the DDBs is in the RH of the DEVSER word. The RH contains the device service dispatch table for that device, which points to specific I/O routines. The device service routines are different for PTYs and TTYs. But both sets of service routines will use

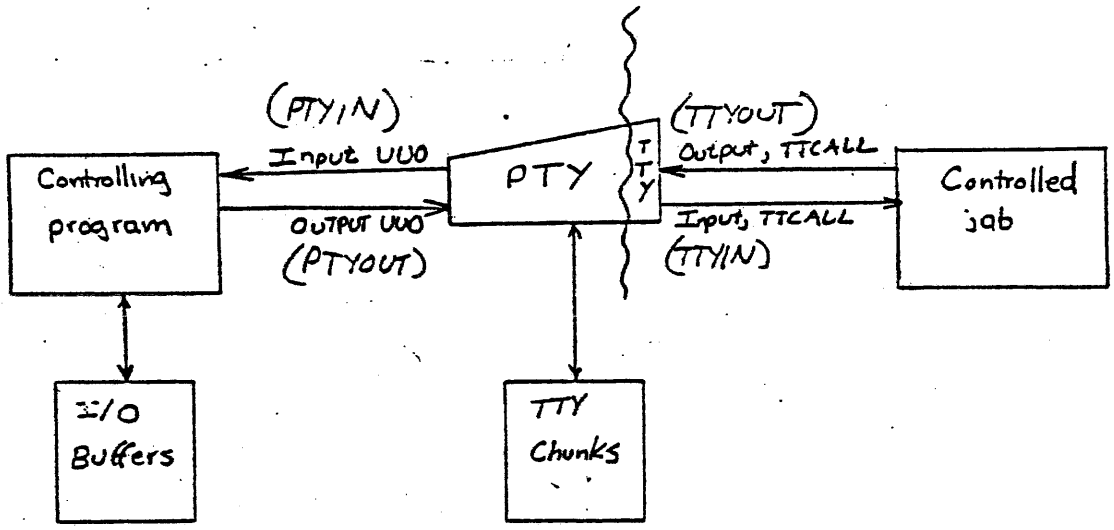
the same TTY chunks for the PTY "line".

There are two directions that data may go: from the controlling program to the controlled job or vice versa. Consider the first case. The controlling program, after having setup the PTY with an INIT or OPEN, will send data to the PTY by using an OUTPUT monitor call. UUOCON searches the RH of DEVSER for the dispatch table (in this case PTYDSP) and transfers control to PTYOUT in PTYSER. PTYOUT will take characters from the controlling job's buffer and store them in the PTY's chunks using PTYPUT. PTYPUT (in SCNSER) will not permit echoing of the characters and will wake the controlled job via a call to ECHBRK if a break character is received. Then the controlled job will proceed in the normal way either at command level or user level. If the controlled job had done an INPUT to get into the wait state, the routine it uses (for input from a TTY) is TTYIN, not PTYIN.

The other direction is slightly more complicated because the monitor must prevent characters from being typed anywhere when the controlled job does output to what it thinks is a TTY. When the controlled job issues an OUTPUT monitor call, OUTSTR TTCALL or OUTCHR TTCALL, the data will be stored in the chunks but the LDB is not placed in the output request queue. Instead control passes to PTSTRT where the controlling job is awakened from its HIBER using a call to PTYPE (in PTYSER). The controlling program will then get the data using an INPUT UUO. The routine to handle PTY input is PTYIN (in PTYSER). PTYIN gets characters from the chunks using PTYGET and stores them in the controlling job's buffers.

SCN-49

The controlled job believes it is performing I/O with a TTY.



MACRO INTERPRETED COMMANDS (MIC)

MIC, or Macro Interpreted Commands, is a feature of TOPS-10 that allows a user to execute a command file at his terminal. The commands are processed in a similar manner to BATCH commands with several important 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 printed directly on his terminal. Batch is not involved at all. For a complete description of features and operation, see MICV2.DOC. This discussion is concerned with the monitor changes, specifically in SCNSER, necessary to accommodate MIC.

The MIC system revolves around a copy of MIC.EXE that is always running under an operator ([1,2]) or privileged user. That one copy of MIC controls all jobs that are using MIC. In the low segment of this MIC "master" is a PDB (Process Data Block) for each job wanting to using the MIC facility. The PDB holds such items as the file to get commands from, the arguments from the DO command line and information about labels within the file. The master responds to the needs of the slaves, feeding them command lines from the appropriate files. The command lines are fed directly into the terminal's input chunks where they can be processed via regular channels.

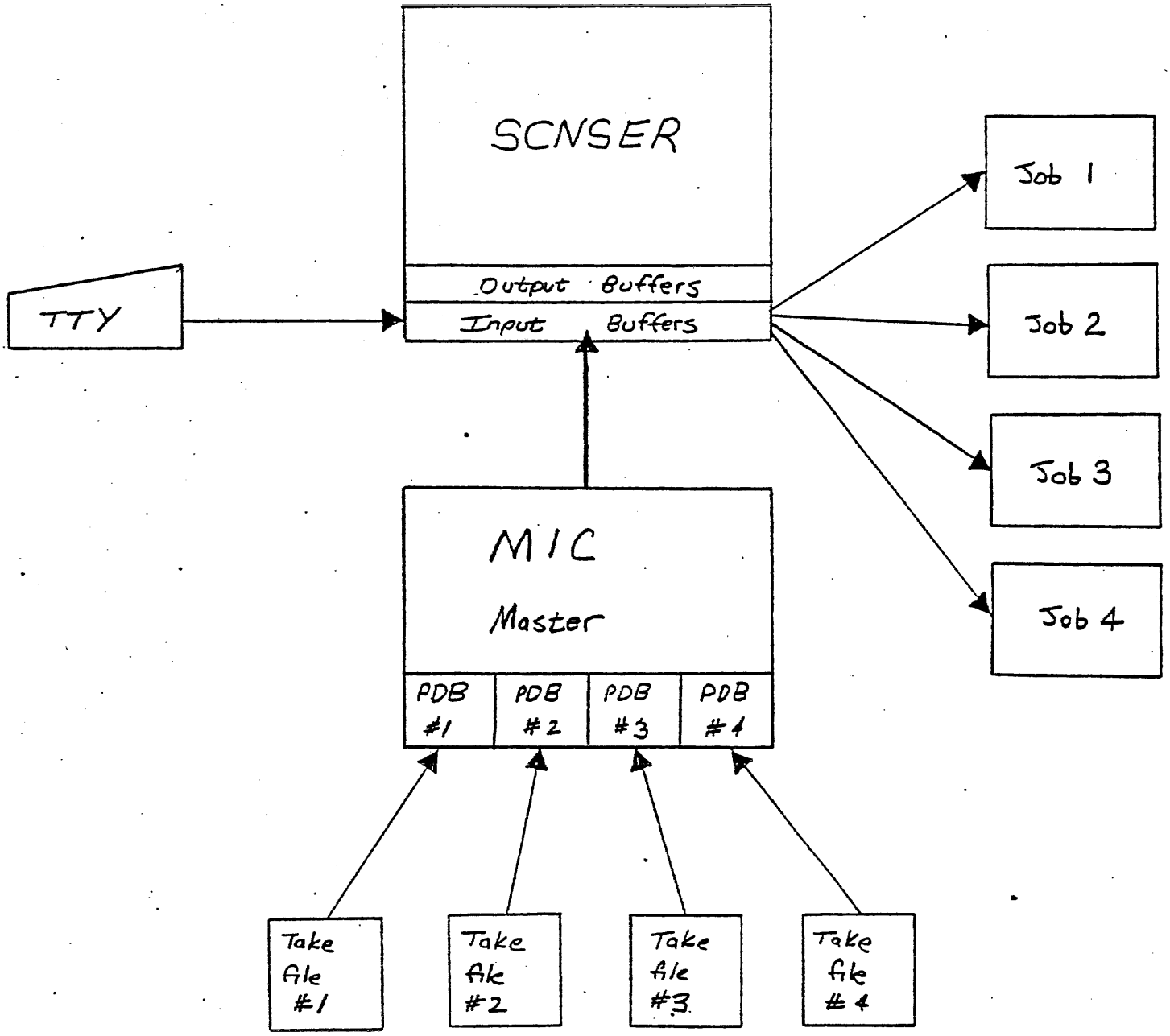
When the user issues the DO command, COMCON sets up and runs the MIC program. The user enters a section of code different from that of the master, setting up the PDB for itself. Once the user exits from MIC, the master takes control. It will open the command file. For each line in the command file, it resolves all arguments and then issues a TRMOP. monitor call, function 21. This places an ASCIZ string directly into the terminal's input buffer. The routine in SCNSER to perform this task is TOPMTY. Characters are entered one by one via calls to RECINU (which as we know is the receive interrupt routine). Refer to earlier flowcharts for a detailed look at RECINU. SCNSER will then perform the usual echoing, alerting UUOCON or COMCON when a break is received.

The only time when special monitor handling is necessary for MIC occurs when the master MIC must be awakened to read another line from a file and keep the slave running. When not servicing slaves, the master HIBERs. A

WAKE. monitor call from SCNSER will wake up the master. There are two situations when wakeup calls must be made: return is about to be made to command level or the job is about to enter TI wait state. Whenever a command has finished, COMCON will call TTYCMR to alert SCNSER. In TTYCMR, the MICWAK routine is called, waking the master for more useful work. If the job is about to enter TI, the TIWAIT routine is called which will also call MICWAK. Notice that in either case, the job will still go into its wait state. The master will soon follow with a command line.

Note that nothing has been mentioned about an interlock against the user typing during the processing of a MIC command file. If a user types during this process, the characters will be stored as usual in the input buffer. The line of data could slip in and cause an error to the MIC command. The user should be careful about typing during MIC if he/she wants proper execution to occur.

The organization of MIC is summarized in the following diagram:


SCNSER
Output Buffers
Input Buffers
TTY
Job 1
Job 2
Job 3
Job 4
MIC
Master
PDB #1
PDB #2
PDB #3
PDB #4
Take file #1
Take file #2
Take file #3
Take file #4

MODULE TEST

On all questions which ask "where", specify page and line number and describe the circumstances under which the line will be executed.

1. When will UUOCON call the device dependent routine in SCNSER, for:

   a. INPUT

   b. OUTPUT

2. Where is the decision made to put a job into IO wait for TTY input?

3. Why can a job be swapped out while in TTY IO wait, but not while in IO Wait for other devices?

4. Where is the decision made to set the "Command Ready" bit, LDBCMR?

5. Where is the "Output in Progress" bit, LDLOIP?

   a. Set?

   b. Cleared?

6. When is a TTY "attached" to a job? What actions does this involve?

7. When are the characters typed as a monitor command discarded? Characters typed as TTY input?

8. If a paper tape is being read on a TTY and the buffer is nearly full, the Receive Interrupt Routine stops the reader by sending out an XOFF character. When and how is the reader restarted?

9. What determines whether a special character is acted on by the Scanner Service or passed to the program?

1Ø. What action does the Scanner Service take if a program doing image mode input "times out"?

11. Suppose a job is running deteached and runs into a program error. When and where will the monitor error message be typed?