TOPS-10 MONITOR INTERNALS

Supplemental Readings

EDUCATIONAL SERVICES
DIGITAL EQUIPMENT CORPORATION
BEDFORD, MASS.

REVISION 3 JULY 1979
REVISION 4 AUGUST 1979
REVISION 5 FEBRUARY 1980
REVISION 6 NOVEMBER 1980

EY-CD013-RB-006

PREFACE

The purpose of this document is to supplement the TOPS-10 MONITOR INTERNALS COURSE MATERIAL prose with graphic illustrations and additional support documents.

This supplement is divided into parts with the page numbering continuous within each part but not continuous across parts. The page numbering in part 1, Graphics, is of the form "a-b" where "a" corresponds to the chapter number in the course materials and "b" is the page number within chapter "a". This numbering scheme facilitates cross referencing the course materials to the supplement.

TABLE OF CONTENTS

PART 1 - GRAPHICS

INTRODUCTION	1-1
MAPPING	1-3
PI SYSTEM	1-10
MONITOR BUILDING	1-11
MONITOR CODING CONVENTIONS	1-12
CLOCK CYCLE OVERVIEW	2-1
CLOCK CYCLE LEVEL "3"	2-2
CLOCK CYCLE LEVEL 7	2-3
TIMING CHARTS	2-6
ACCOUNTING	2-11
CORE MANAGEMENT	3-1
PAGE FAULTS	3-11
COMMAND PROCESSING OVERVIEW	4-1
COMMAND PROCESSING DETAILS	4-2
DELAYED COMMANDS	4-11
JOB STATE TRANSITIONS	5-1
SCHEDULER QUERIES	5-2
SCHEDULER DETAILED FLOWS	5-3
SWAPPER	6-1
UUO PROCESSING OVERVIEW	7-1
UUO PROCESSING DETAILS	7-2
HOW TOPS-10 DIES	7-7
STOPCODES	7-15
I/O MODULE ARCHITECTURE	8-1
INIT UUO	8-7
INBUF/OUTBUF UUO	8-13
INPUT UUO	8-14
OUTPUT UUO	8-16
CLOSE UUO	8-17
RELEASE UUO	8-19
NOTES ON I/O UUOS	8-21
INTERRUPT CHAIN	9-1
MONGEN PI ASSIGNMENT	9-2
PI ASSIGNMENT DEVICE GROUPS	9-3
DEVICE INTERRUPT ROUTINE OVERVIEW	9-4

ADVANCE BUFFER ROUTINE 9-5
DEVICE DATA BLOCKS 9-8
WAIT ROUTINES 9-9
CALIN - START DEVICE ROUTINE 9-11
SETOID - ROUTINE TO UNBLOCK A JOB 9-14
A RACE CONDITION 9-20

DISK RESIDENT DATA BASE 10-1
CORE RESIDENT DISK DATA BASE 10-5
DISK UUO I/O FLOWS 10-10
I/O INSTRUCTION FORMAT 10-17
DISK QUEUES 10-20
DISK INTERRUPT LEVEL FLOWS - FILINT 10-21
DISK POSITIONING OPTIMIZATION 10-24
DISK TRANSFER OPTIMIZATION 10-25
START I/O 10-26
SET UP COMMAND LIST 10-27

PART 2 - KL DOCUMENT

PART 3 KL SYSTEM OPERATIONS (CHAPTER 3 HARDWARE REFERENCE MANUAL)

PRIORITY INTERRUPTS 3.1
CACHE MANAGEMENT 3.2
TOPS-10 PAGING AND PROCESS TABLES 3.3
MEMORY MANAGEMENT 3.5
TIMING AND ACCOUNTING 3.6
ERROR AND DIAGNOSTIC INSTRUCTIONS 3.8

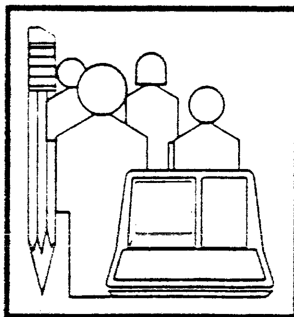
PART 4 - SCHEDULER/SWAPPER PLM

PART 5 - DISK I/O PROCESSING

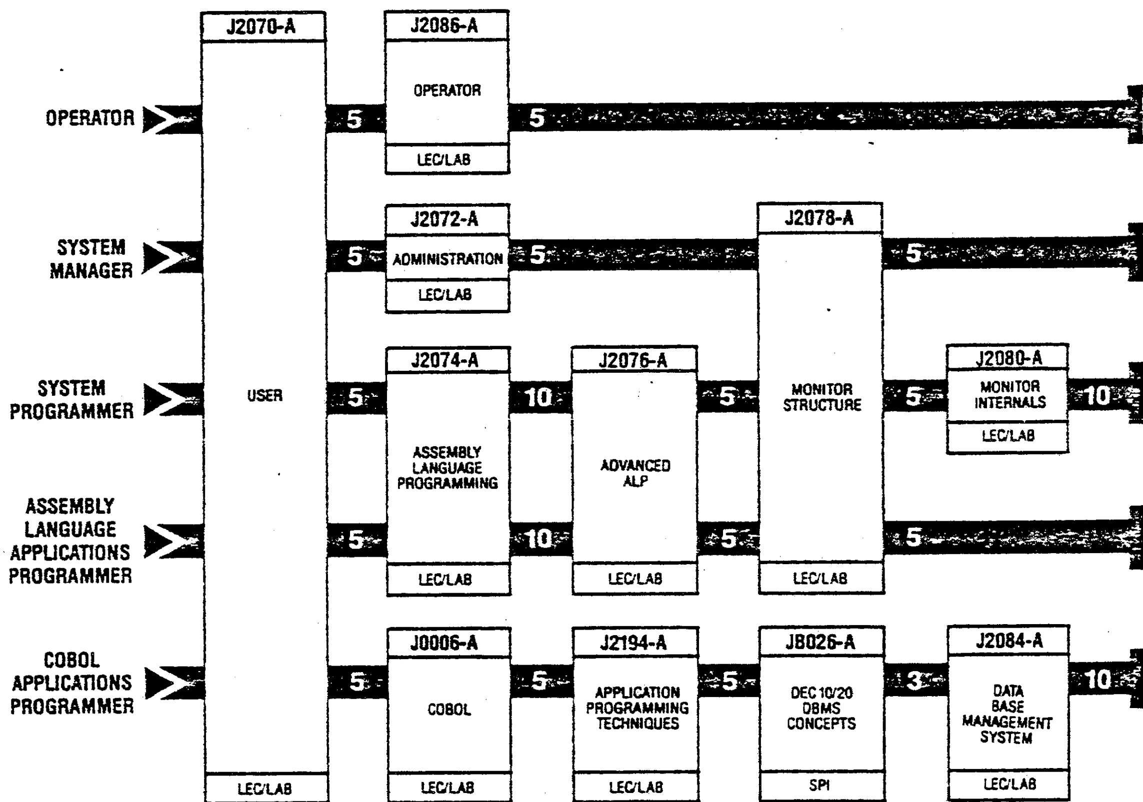
PART 6 - LABS

PART 1

GRAPHICS



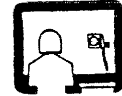
TOPS-10 (DECsystem-10) TRAINING PROGRAM



J2080-A

TOPS-10 Monitor Internals

Length: 10 days



Lecture/Lab

This quite advanced course teaches the experienced programmer the internal algorithms of the TOPS-10 operating system in detail. In-depth studies of the monitor clock cycle and device service routines receive equal emphasis. Students will study monitor MACROs and conventions, TOPS-10's data base in great detail, and will learn methods for adding new commands, monitor calls (UUO), and device service routines to TOPS-10. Laboratory exercises introduce on-line examination of the data base and post-mortem crash analysis with the FILDDT utility.

The experienced Programmer who completes this course will be well-grounded in the monitor's major algorithms, from core management to communications service routines. He will feel comfortable finding his way through the code, and will be capable of making modifications to TOPS-10 to implement new features for his installation.

Students:

- System Programmers

Will Learn to:

- Describe the steps which must be followed in adding either a new command or UUO to the monitor.
- Describe the principles involved in adding a new device service routine.
- Given a specific system state, trace the control path through the monitor.
- Describe the effects of an interrupt on the monitor data base and on subsequent monitor behavior.
- Describe how a user disk I/O request is handled by the disk service routines.
- Use FILDDT to examine the data base of a running monitor or to post mortem a crash.
- Efficiently find the section of TOPS-10 code that performs a particular function and follow its flow.

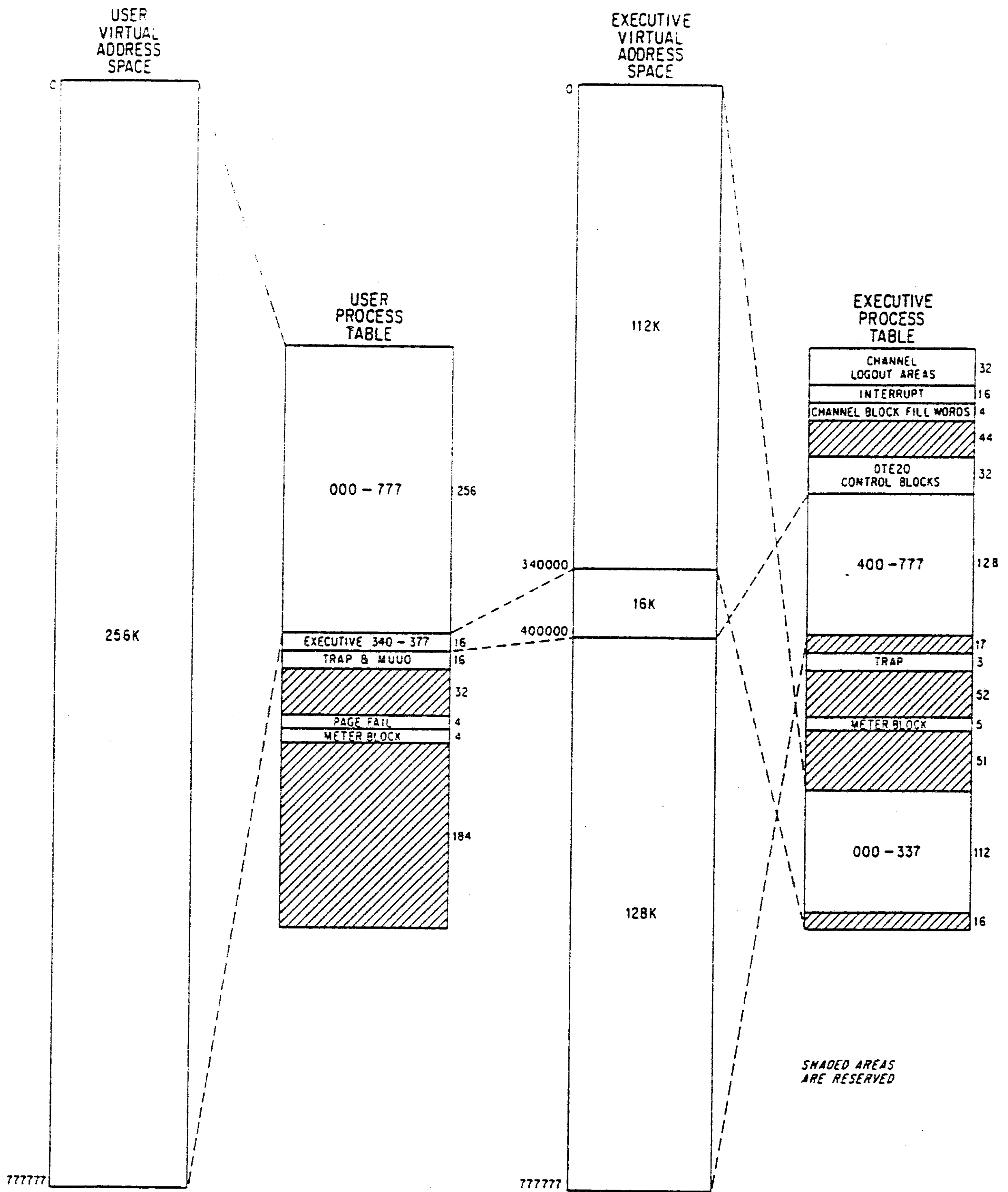
Ensuring Success:

The flowchart illustrates the proper course sequence for every job classification within the TOPS-10 training program.

In order to ensure the training success of every participant, it is mandatory that prospective students take all courses in the recommended sequence. For example, before taking this course, you should have completed **TOPS-10 Monitor Structure** and **TOPS-10 Assembly Language Programming.** We also recommend six months practical experience as a systems programmer under TOPS-10.

Topics:

- Monitor Coding Conventions and Cross-Reference Tools
- Clock Routine
- Core Management
- Command Processor
- Scheduler and Swapper
- Monitor Calls and Device Service Routines
- File Service Routine
- Communications Processor
- FILDDT and Introduction to Crash Analysis



TOPS-10 VIRTUAL ADDRESS SPACE AND PROCESS TABLE LAYOUT

MR-0750

USER PROCESS TABLE

Address	Contents	
0	USER PAGE 0	USER PAGE 1
377	USER PAGE 776	USER PAGE 777
400	EXECUTIVE PAGE 340	EXECUTIVE PAGE 341
417	EXECUTIVE PAGE 376	EXECUTIVE PAGE 377
420	RESERVED	
421	USER ARITHMETIC OVERFLOW TRAP INSTRUCTION	
422	USER STACK OVERFLOW TRAP INSTRUCTION	
423	USER TRAP 3 TRAP INSTRUCTION	
424	MUUO STORED HERE	
425	MUUO OLD PC WORD	
426	MUUO PROCESS CONTEXT WORD	
427	RESERVED	
430	KERNEL NO TRAP MUUO NEW PC WORD	
431	KERNEL TRAP MUUO NEW PC WORD	
432	SUPERVISOR NO TRAP MUUO NEW PC WORD	
433	SUPERVISOR TRAP MUUO NEW PC WORD	
434	CONCEALED NO TRAP MUUO NEW PC WORD	
435	CONCEALED TRAP MUUO NEW PC WORD	
436	PUBLIC NO TRAP MUUO NEW PC WORD	
437	PUBLIC TRAP MUUO NEW PC WORD	
440–477	RESERVED	
500	PAGE FAIL WORD	
501	PAGE FAIL OLD PC WORD	
502	PAGE FAIL NEW PC WORD	
503	RESERVED	
504–505	USER PROCESS EXECUTION TIME	
506–507	USER MEMORY REFERENCE COUNT	
510–777	RESERVED	

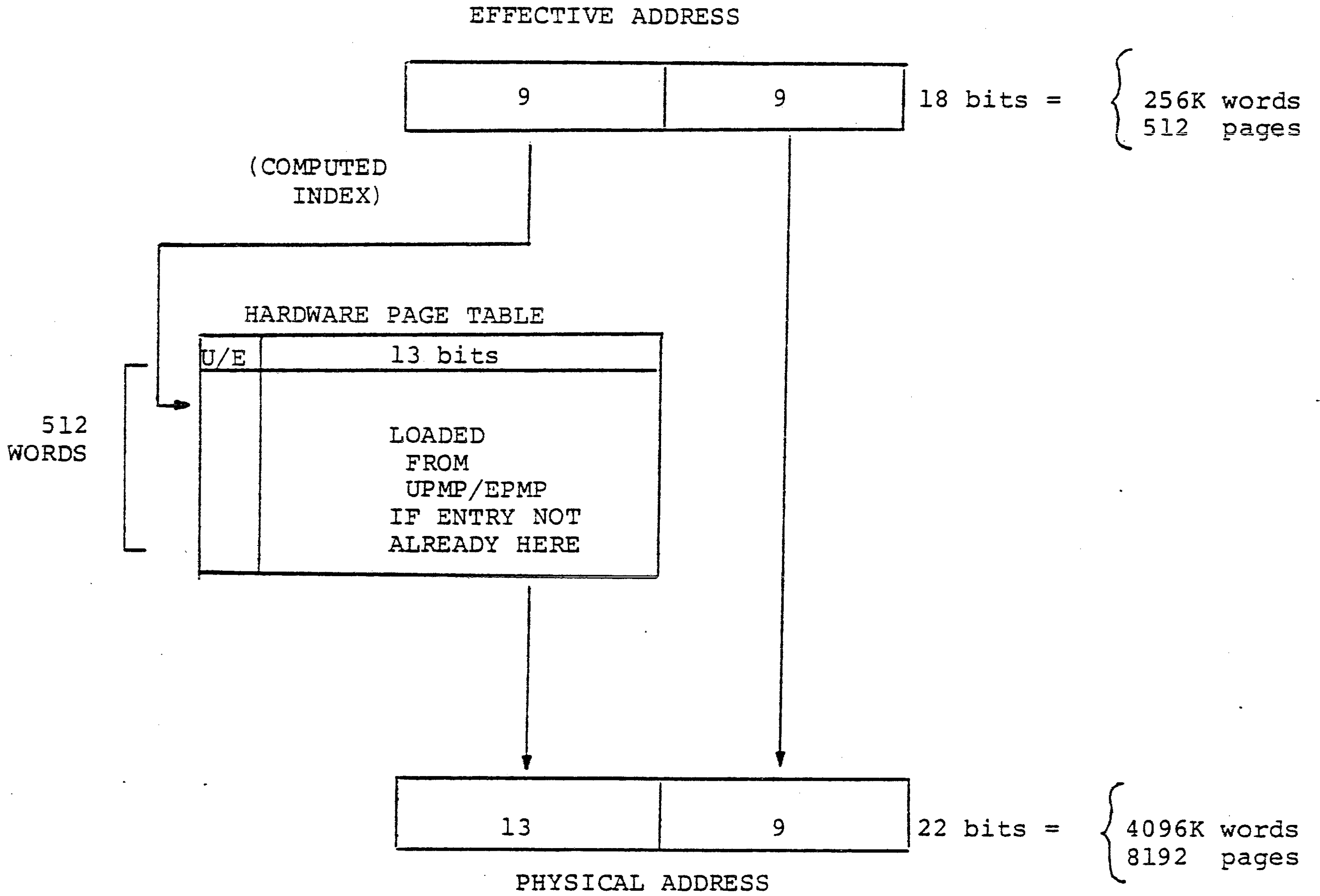
EXECUTIVE PROCESS TABLE

Address	Contents	
0–37	EIGHT CHANNEL LOGOUT AREAS EACH: 0 INITIAL CHANNEL COMMAND 1 GETS CHANNEL STATUS WORD 2 GETS LAST UPDATED COMMAND 3 RESERVED	
40–41	RESERVED	
42–57	STANDARD PRIORITY INTERRUPT INSTRUCTIONS	
60–63	FOUR CHANNEL BLOCK FILL WORDS	
64–137	RESERVED	
140–177	FOUR DTE20 CONTROL BLOCKS	
200	EXECUTIVE PAGE 400	EXECUTIVE PAGE 401
377	EXECUTIVE PAGE 776	EXECUTIVE PAGE 777
400–420	RESERVED	
421	EXECUTIVE ARITHMETIC OVERFLOW TRAP INSTRUCTION	
422	EXECUTIVE STACK OVERFLOW TRAP INSTRUCTION	
423	EXECUTIVE TRAP 3 TRAP INSTRUCTION	
424–507	RESERVED	
510–511	TIME BASE	
512–513	PERFORMANCE ANALYSIS COUNT	
514	INTERVAL COUNTER INTERRUPT INSTRUCTION	
515–577	RESERVED	
600	EXECUTIVE PAGE 0	EXECUTIVE PAGE 1
757	EXECUTIVE PAGE 336	EXECUTIVE PAGE 337
760–777	RESERVED	

TOPS-10 PROCESS TABLE CONFIGURATION

MR-0751

VIRTUAL TO PHYSICAL ADDRESS TRANSLATION



USER PAGE MAP (UPMP) MAPPING ENTRY

A	P	W	S	C	PAGE ADDRESS

18 BIT QUANTITY - 512 PER UPMP

A = 0 ACCESS DENIED, PAGE FAULT OCCURS

= 1 ACCESS ALLOWED

P = 0 CONCEALED PAGE (EXECUTE ONLY)

= 1 PUBLIC PAGE

W = 0 WRITE PROTECTED

= 1 WRITABLE

S = 0 ALLOCATED

= 1 ALLOCATED BUT ZERO

C = 0 CACHEABLE

= 1 NOT CACHEABLE

PAGE ADDRESS - 13 BIT PHYSICAL MEMORY PAGE NUMBER OR

- 17 BIT SWAPPING SPACE ADDRESS

INCLUDES P,W,S,C BITS

Exec Page Map

```
CPDEF MUUO (0)

40/
41/ 0 (JSR LUUOPC placed here               LUUOPC: 0
       at SYSINI time)                              EXCH T1, LUUOPC
 .                                                  MOVEM T1, UUO0
 .                                                  JRST   UUOERR##
 .
  LOC 420
420/MUUO SEILM##                   ; Page fault trap
421/JFCL                           ; Arithmetic trap
422/MUUO SEPDLO##                  ; Push down overflow trap
423/JSR TRP3PC                     ; Trap 3 Trap
```

User's Page Map   KI STYLE

```
Loc NLUPMP (= 2000)
NUPPPM = NLUPMP + 400
Loc NUPPPM

400/PM.ACC + PM.WRT + 340,, PM.ACC + PM.WRT + 341 \
    PM.ACC + PM.WRT + 342,, PM.ACC + PM.WRT + 343  |
         .                                         > Exer. Per Process
         .                                         |      Map
         .                                         |
417/PM.ACC + PM.WRT + 374,, PM.ACC + PM.WRT + 375 /

420/ MUUO SEILM##                  ; Page fault trap
421/ JFCL SAROUF##                 ; Arithmetic trap
422/ MUUO SEPDLO##                 ; Push down overflow trap
423/ JFCL                          ; Trap 3 Trap
424/ EXP 0                         ; MUUC stored here
425/ EXP 0                         ; PC word of MUUO stored here
426/ EXP 0                         ; Exec page fail word
427/ EXP 0                         ; User page fail word
430/ EXP IC. UOU + MUUO##          ; Kernel No trap MUUO new PC word
431/ EXP IC.UOU + KTUUO##          ; Kernel trap MUUO new PC word
432/ EXP IC.UOU + SNTUUO##         ; Supervisor No trap MUUO new PC word
433/ EXP IC.UOU + STUUO            ; Supervisor trap MUUO new PC word
434/ EXP IC.UOU + MUUO##           ; Concealed No trap MUUO new PC word
435/ EXP IC.UOU + CTUUO            ; Concealed trap MUUO new PC word
436/ EXP IC.UOU + MUUO##           ; Public No trap MUUO new PC word
437/ EXP IC.UOU + PTUUO            ; Public trap MUUO new PC word
KTUUO: JRST   @   .UPMP +. UPMUO ; Dispatch to kernel mode trap
                                    handler
SNTUUO: Halt
STUUO:  Halt
CTUUO:  JRST  @   .UPMPT.UPMUO   ; Dispatch to use mode trap handler
PNTUUO: JRST  @   .UPMP + .UPMUO ;     "                          "
PTUUO:  JRST  @   .UPMP + .UPMUO ;     "                          "

; Come here on a MUUO call to similuate a KA10 UUO
       MUUO: __
             __
             __
             .
             .
             __
```

Executive Virtual Memory

Monitor Virtual Address Space

6.03A

Page	Contents
Page 0	Absolute Locations
1	EPMP (CPU0)
2	EPMP (CPU1)
3	Null Job UPMP
	Monitor Low Segment
340	UPMP
341	JOBDAT
342	Vestigal JOBDAT
343	TEMP
400	Used to Build UPMP
401	Swapping Checksum
402	PI Level Temporaries
411	SKPCPU Instruction
412	PAGTAB
432	MEMTAB
452	Monitor High Segment
SYSSIZ	EVM
777	

7.01

Page	Contents
Page 0	Absolute Locations
1	EPMP (CPU0)
2	EPMP (CPU1)
3	Null Job UPMP
	Monitor Low Segment
340	Funny Space
367	
370	UPMP
371	JOBDAT
372	Vestigal JOBDAT
373	TEMP
400	Used to Build UPMP
401	Swapping Checksum
402	PI Level Temporaries
411	CDB
412	
414	PAGTAB
434	MEMTAB
454	Monitor High Segment
SYSSIZ	EVM
777	

MR 5489

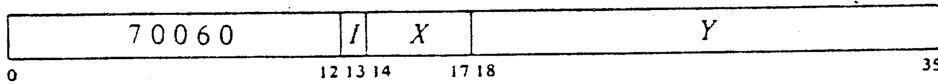
MONITOR ADDRESSABILITY

METHOD	USE	OVERHEAD	RESTRICTIONS
PER PROCESS	ACCESS UPMP & JOBDAT	SETTING UP UPMP MAPPING ENTRIES FOR EXEC MODE PAGING	CURRENT JOB
XCT	UUO ARGUMENTS USER ACS	NONE	CURRENT JOB
EVM	I/O BUFFERS	SETTING UP EPMP MAPPING ENTRIES FOR EXEC MODE PAGING	NONE

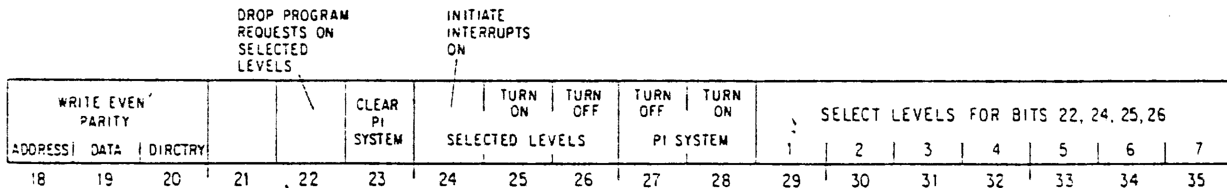
Interrupt Programming

The program can control the priority interrupt system by means of condition I/O instructions. The device code is 004, mnemonic PI.[7]

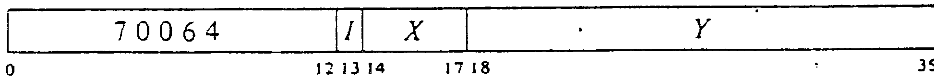
CONO PI, Conditions Out, Priority Interrupt



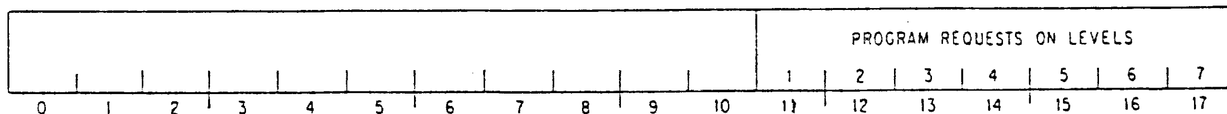
Perform the functions specified by the effective conditions E as shown[8] (a 1 in a bit produces the indicated function, a 0 has no effect).

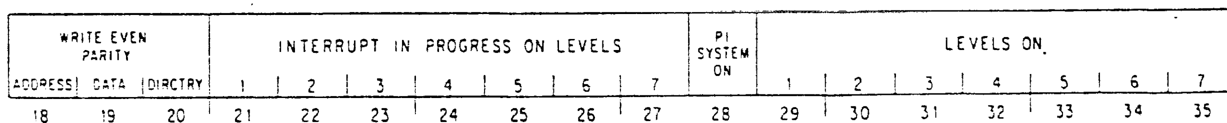


CONI PI, Conditions In, Priority Interrupt



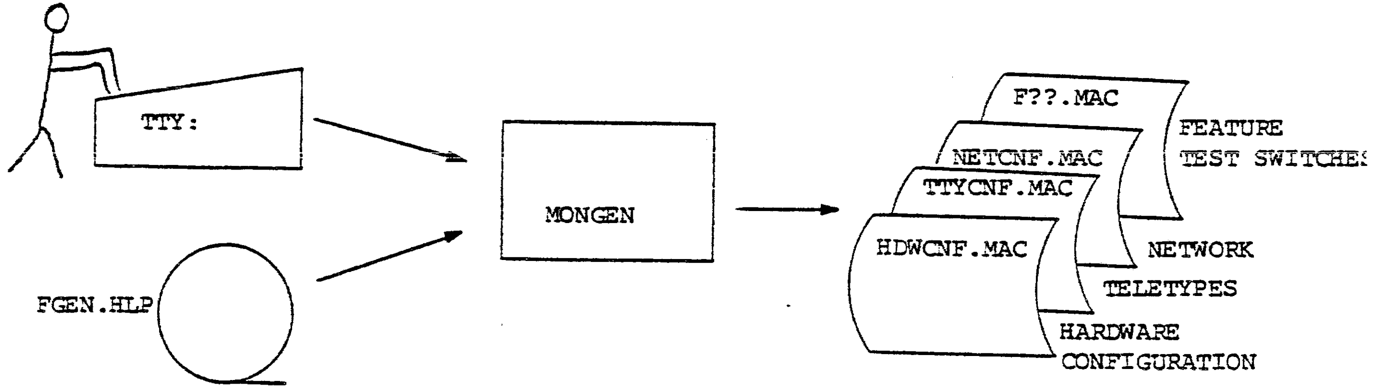
Read the status of the priority interrupt (and several diagnostic bits) into location E as shown.



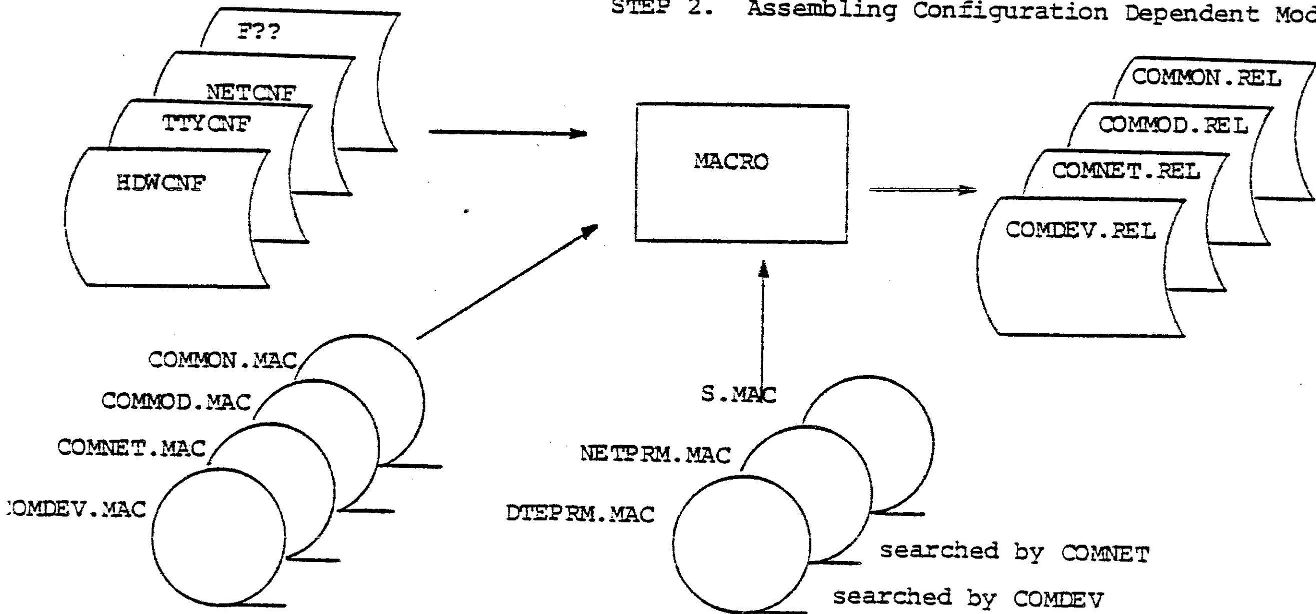


TOPS-10 MONITOR GENERATION

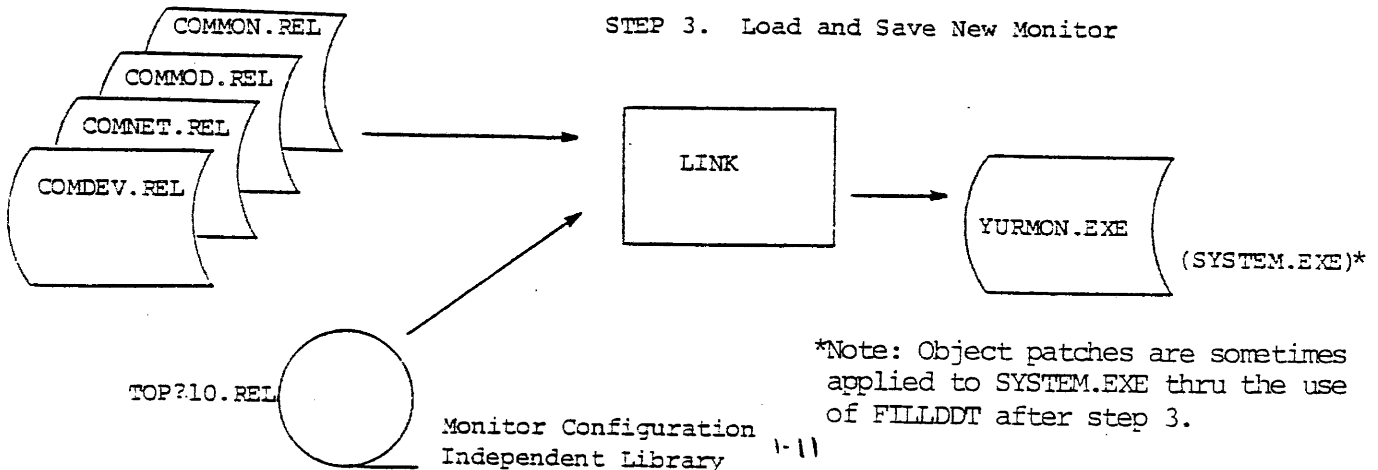
STEP 1. Specifying Configuration



STEP 2. Assembling Configuration Dependent Modules



STEP 3. Load and Save New Monitor



*Note: Object patches are sometimes applied to SYSTEM.EXE thru the use of FILDDT after step 3.

MONITOR AC LOCATIONS

All sixteen monitor AC locations (that is, the TOPS-10 system's sixteen fast-memory locations) have names that are descriptive of their contents. These names remain the same throughout the monitor. The following is a description of these locations, also known as CPU registers.

Fast-Memory Locations 0 to 17

0 S -- Contains the status word from a DDB while the monitor is processing I/O operations.

1 P -- Contains the pushdown stack pointer currently in use.

2 J -- Contains the job number, high-segment number, or controller data block address at interrupt level.

3 R -- Contains the job's relocation value. On KI or KL machines, this usually points to the user page map via exec page 341; that is, R contains the value of 341000. If the job is locked in EVM, however, it contains the exec virtual address of the job.

4 F -- Contains the file DDB address when the monitor is working with I/O. This AC is usually used as a temporary register when the monitor is executing code in an area not concerned with I/O.

5 U -- Contains the unit data block address in FILSER; holds the line data block address in SCNSER. This AC is generally associated with a particular I/O device.

6 T1 -- Is an unpreserved temporary AC.

7 T2 -- Is an unpreserved temporary AC.

10 T3 -- Is an unpreserved temporary AC.

11 T4 -- Is an unpreserved temporary AC.

12 M -- Contains a mask, or, in UUOCON, holds the UUO address and special bits.

13 W -- Usually contains the pointer to the process data block; is a general work register.

14 P1 -- Is a preserved temporary AC.

15 P2 -- Is a preserved temporary AC.

16 P3 -- Is a preserved temporary AC.

17 P4 -- Is a preserved temporary AC; often points to the CPU data block.

Notice that two sets of general-purpose registers are provided, T1 to T4 and P1 to P4. When the system programmer writes a subroutine, he knows he can use T1 through T4 without bothering to preserve the original contents, because they should be saved by the caller. The system programmer should also realize that any subroutine he may call need not worry about the original contents of registers T1 through T4; however, if he wants to use P1 through P4, he must take steps to save their data. Once this is saved, if the system programmer writing a subroutine uses P1 through P4, he can feel free to call other subroutines and expect to return with these registers intact.

Dot Convention in Symbol Naming

Symbols defining numbers begin with a dot, followed by a two-letter prefix. Masks start with a two-letter prefix, followed by a dot, and UUO opcodes end with a dot.

GETTAB word arguments start with %. GETTAB masks are of the form XX%YYY; error codes end with %; and $ symbols are reserved for the installations.

DATA BLOCK WORD ADDRESSES:

.JB???	Job data area symbols (JOBDAT.MAC)
?????.	CALLI UUO symbols implemented after 5.Ø3 (UUOCON.MAC)
.PD???	Symbols in the process data block (PDB), usually indexed by W.
.RB???	File extended arguments (LOOKUP, ENTER, RENAME) (S.MAC)
.CP???	Locations in CPU data Block (CDB),
.CØ???	Locations in CPUØ CDB (COMMON.MAC)
.C1???	Locations in CPU2 CDB (COMMON.MAC)
.GT???	GETTAB table numbers (UUOCON.MAC)
.EP???	KI1Ø exec page map page symbols (S.MAC)
.UP???	KI1Ø user page map page symbols (S.MAC)

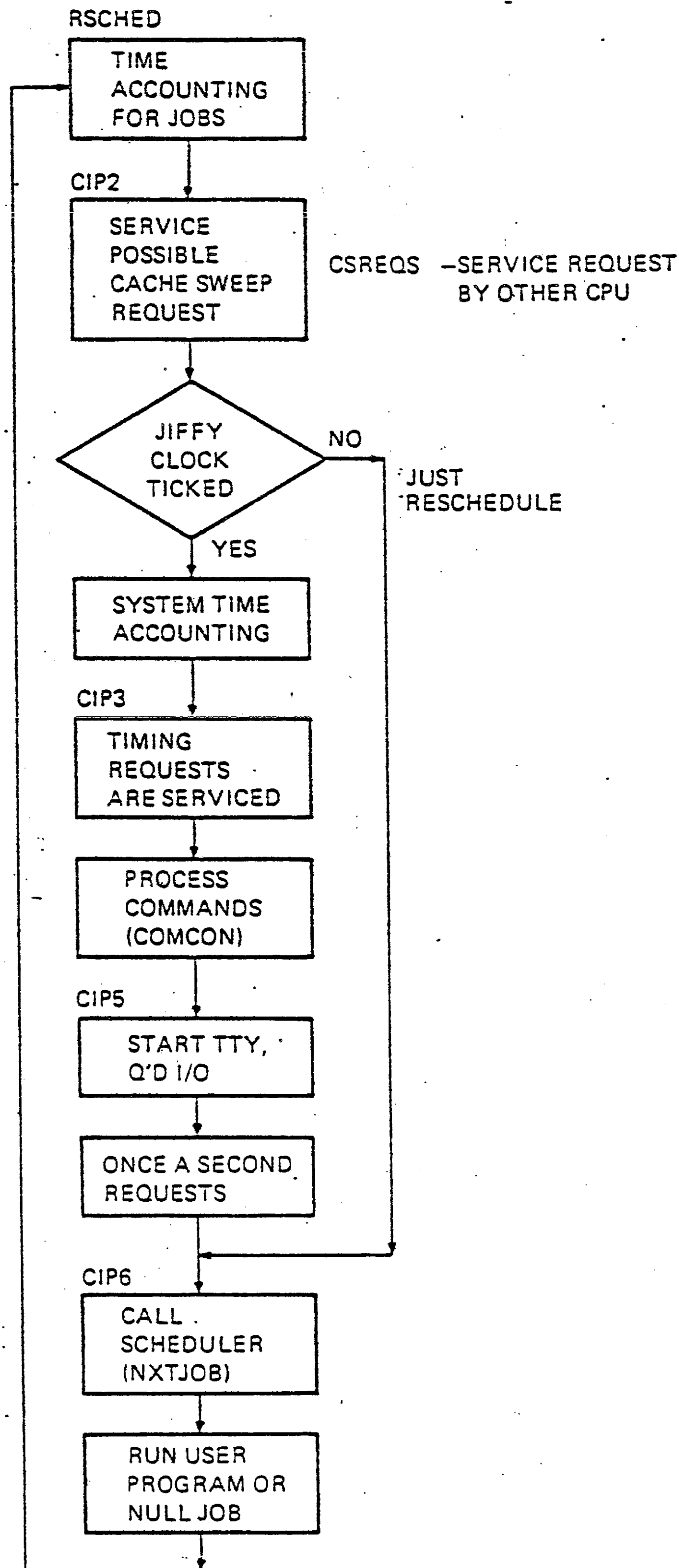
BITS: NOTE that the first four do not follow the convention in the monitor)

.SP???	Spool bits (S.MAC)
.TY???	DEVTYP UUO bits (S.MAC)
.ERMSG	Don't type error messages on error intercept
.OK???	Intercept device ok errors (S.MAC)
JB.L??	Job limit bits (S.MAC)
SP.???	Second processor status bits (S.MAC)
AP.???	APR CONI/CONO bits (S.MAC)
PC.???	PC word flag (S.MAC)
PI.???	PI CONI/CONO bits (S.MAC)
IP.???	KI1Ø APR CONI/CONO bits (S.MAC)
IC.???	KI1Ø PC word flags (S.MAC)
II.???	KI1Ø PI CONI/CONO bits (S.MAC)
XP.???	APR CONO/CONI bits for both KA1Ø and KI1Ø (S.MAC)
XC.???	PC word flags for both KA1Ø and KI10 (S.MAC)
XI.???	PI CONO/CONI bits for both KA1Ø and KI1Ø (S.MAC)

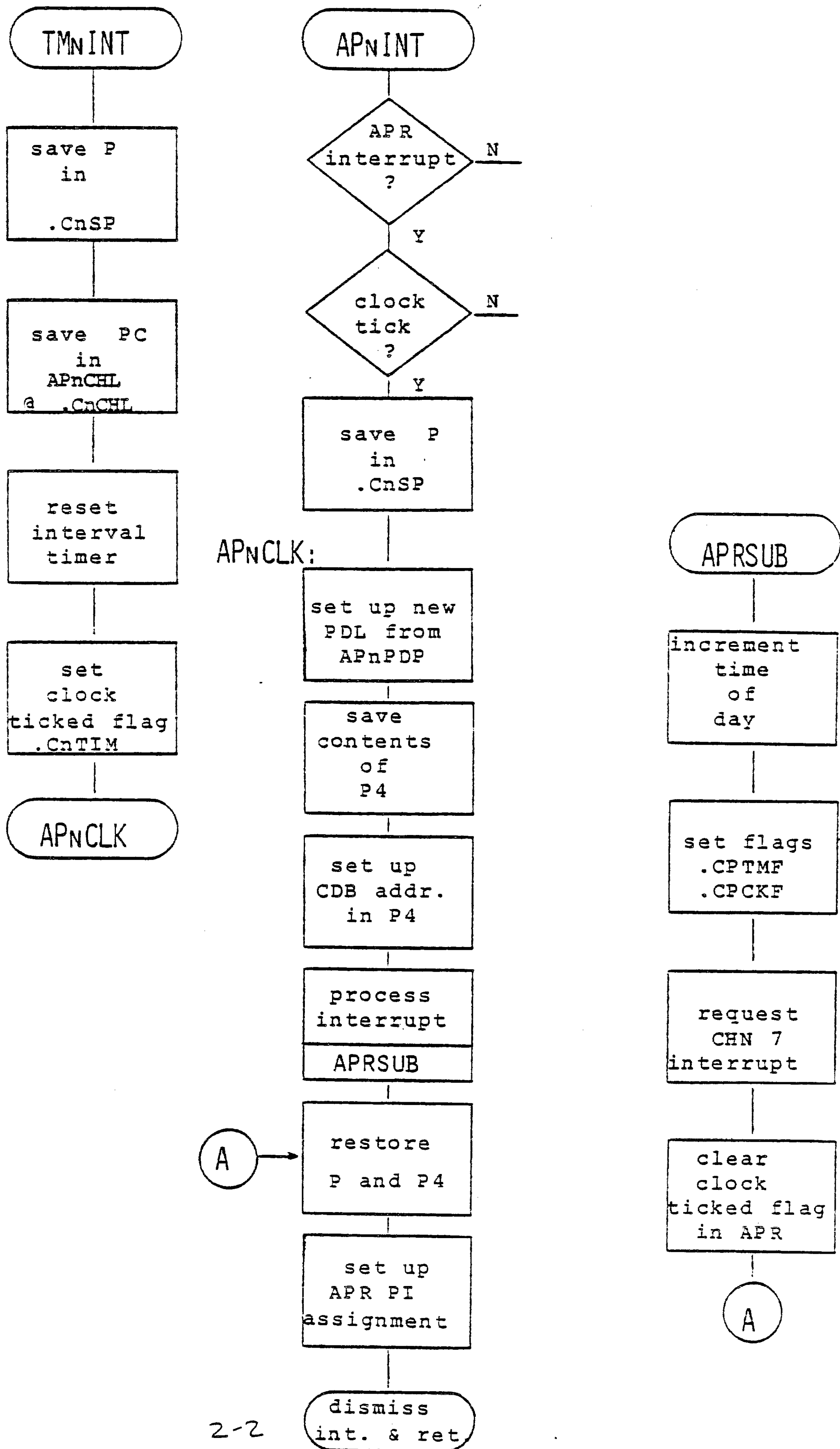
STRUCTURE UUO CODES: (NOTE: These do not follow monitor convention)

.FS???	STRUUO Function code
.ER???	STRUUO error code

Clock Cycle

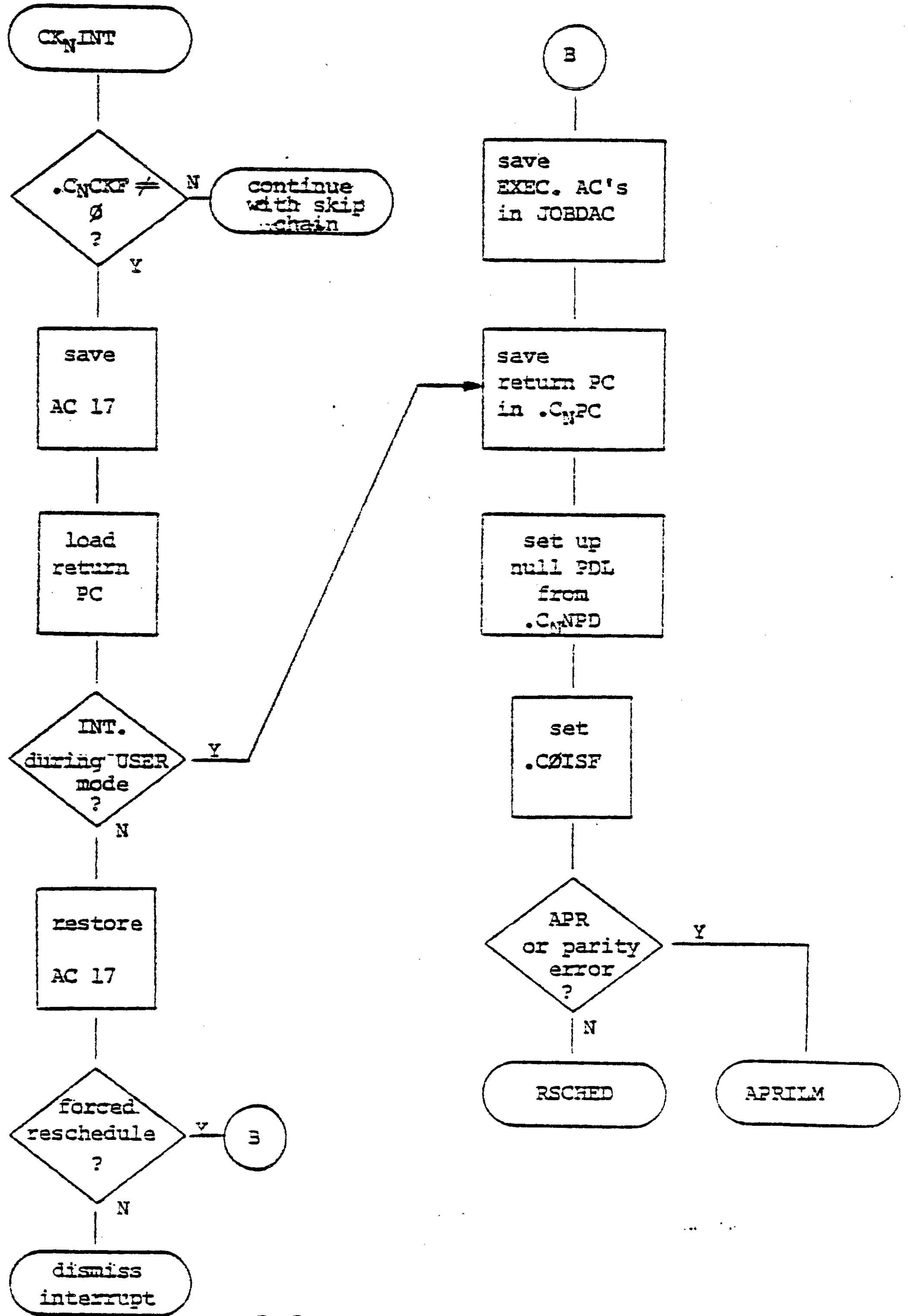


APR INTERRUPT ROUTINE

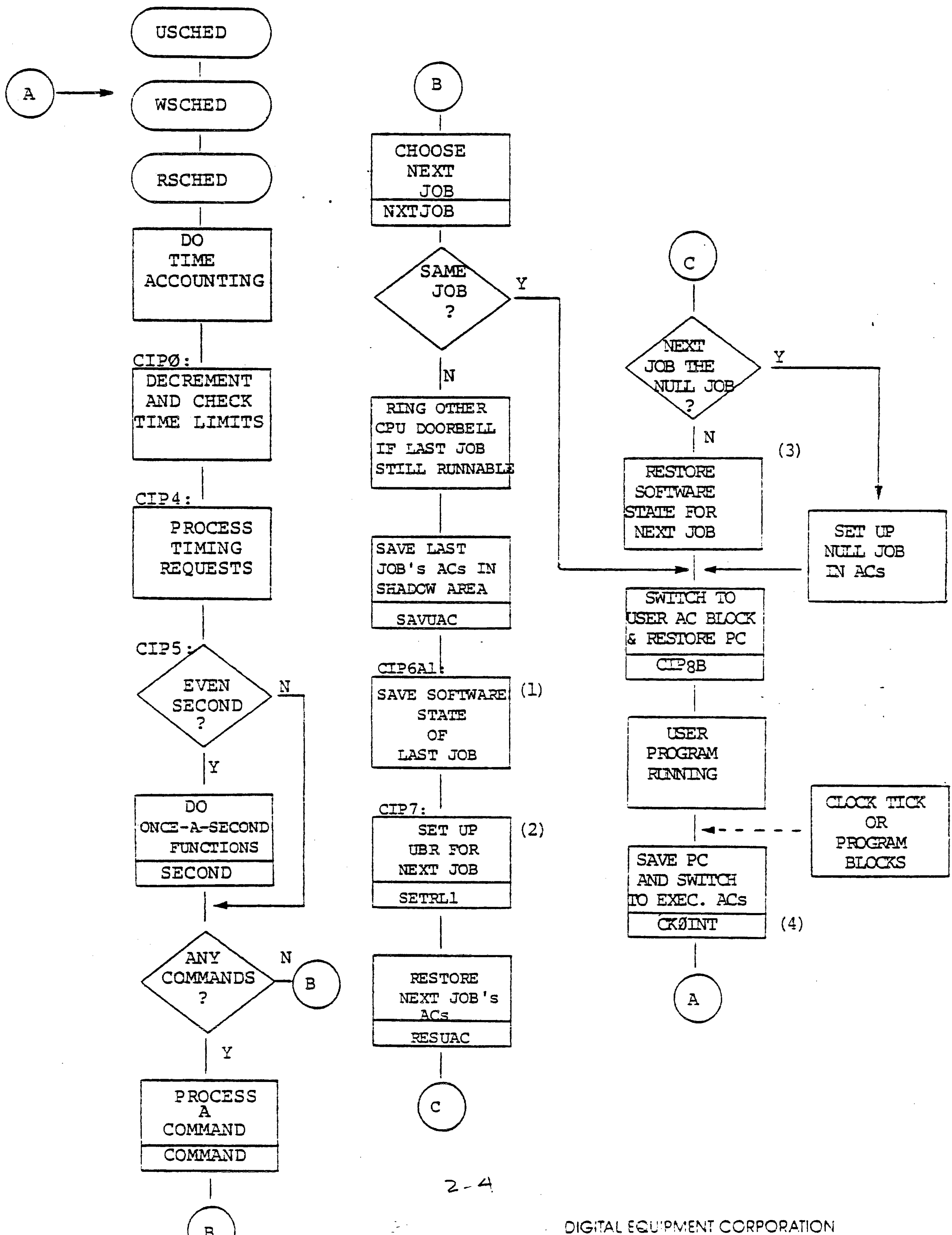


DIGITAL EQUIPMENT CORPORATION

SOFTWARE CLOCK INTERRUPT ROUTINE



THE CONTROL ROUTINE



NOTES FOR THE CONTROL ROUTINE

1. For VM Systems, this means saving the job's PC into JOBPC and the address of DDT into JOBDDT.

   For Non-VM Systems, this means saving the above items and copying the last user's Job Device Assignment table from the CDB into the JOB DATA AREA.

   (Also See Note 3 Below)

2. For KA Systems, this routine sets up the hardware relocation and protection registers.

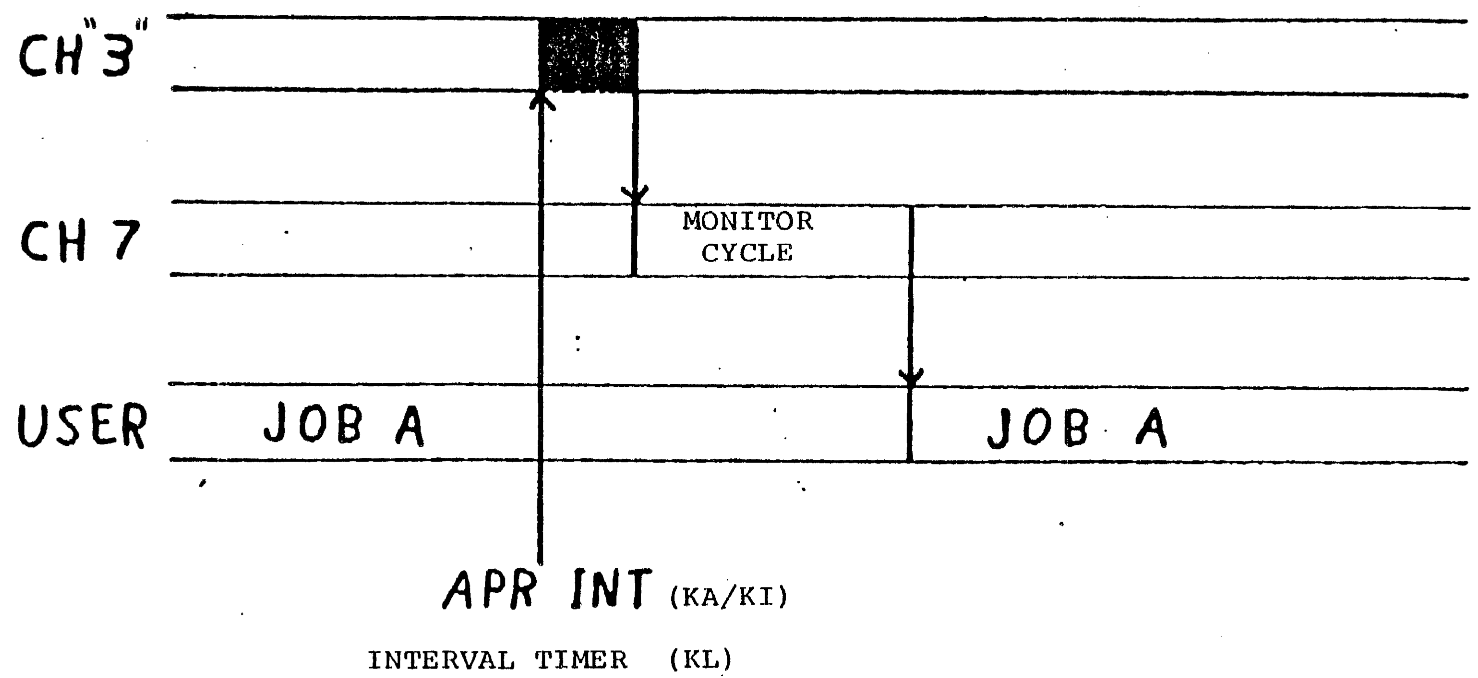
3. For VM Systems, this means restoring the job's PC into .CPPC and the address of DDT into .CPDDT (USRDDT) both in the CPU Data Block.

   For Non-VM Systems, this means restoring the above items and copying the next user's Job Device Assignment table from the JOB DATA AREA into the CPU Data Block.

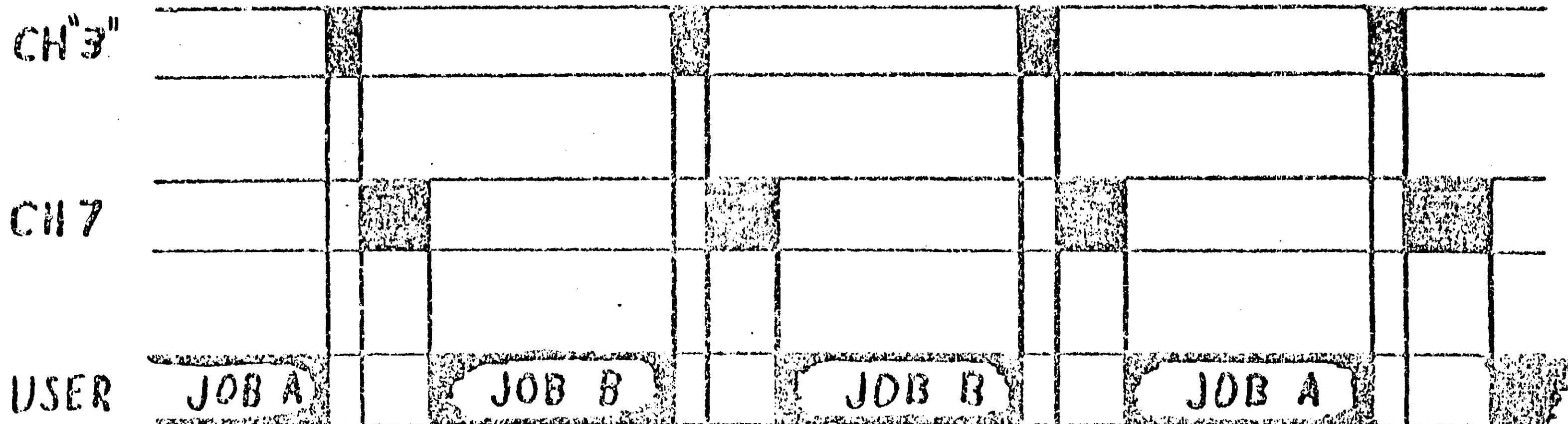
   (The Reverse of Note #1)

4. For KA Systems, this routine would save the User's PC and the User's AC.

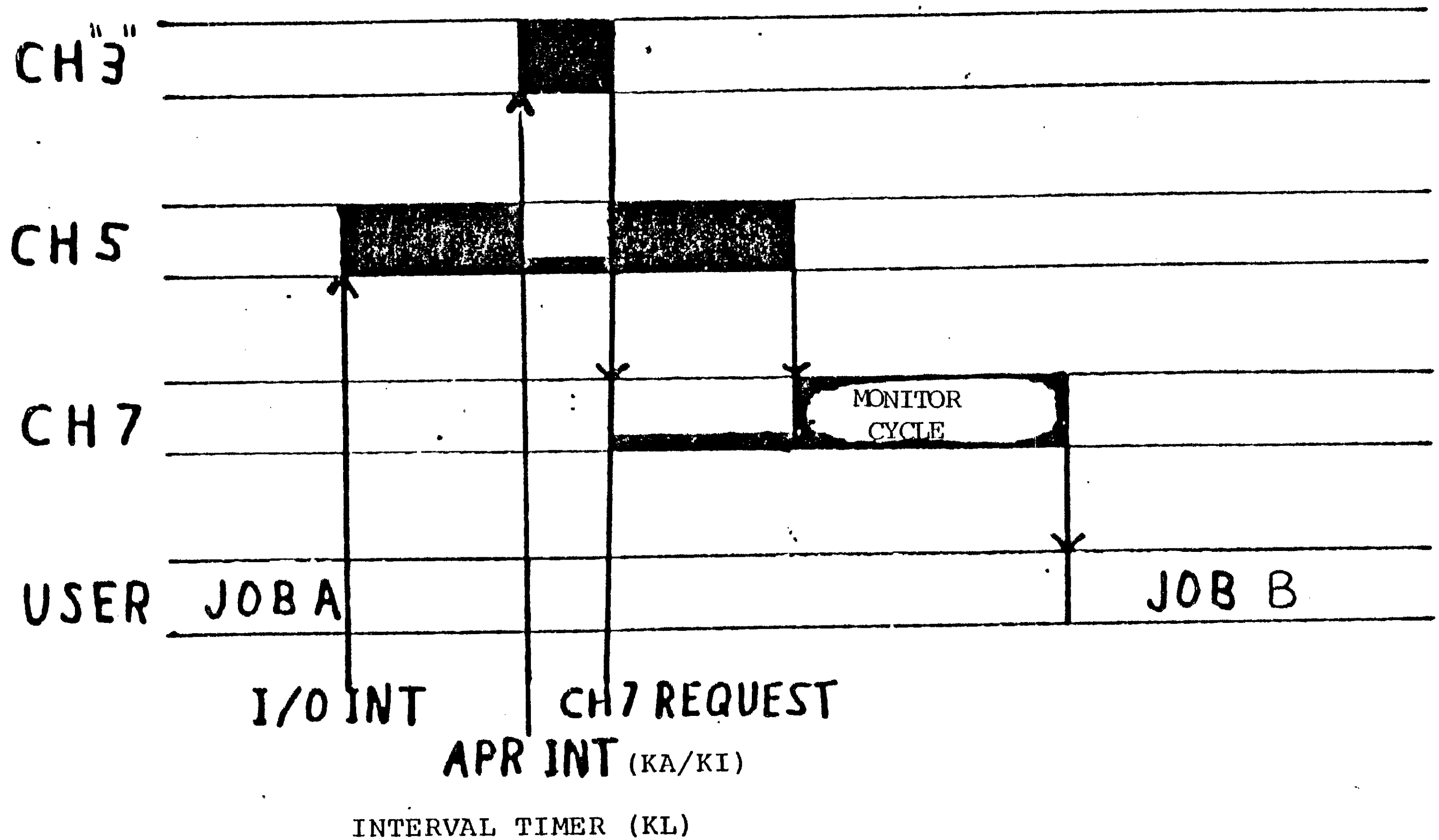
CLOCK TICK



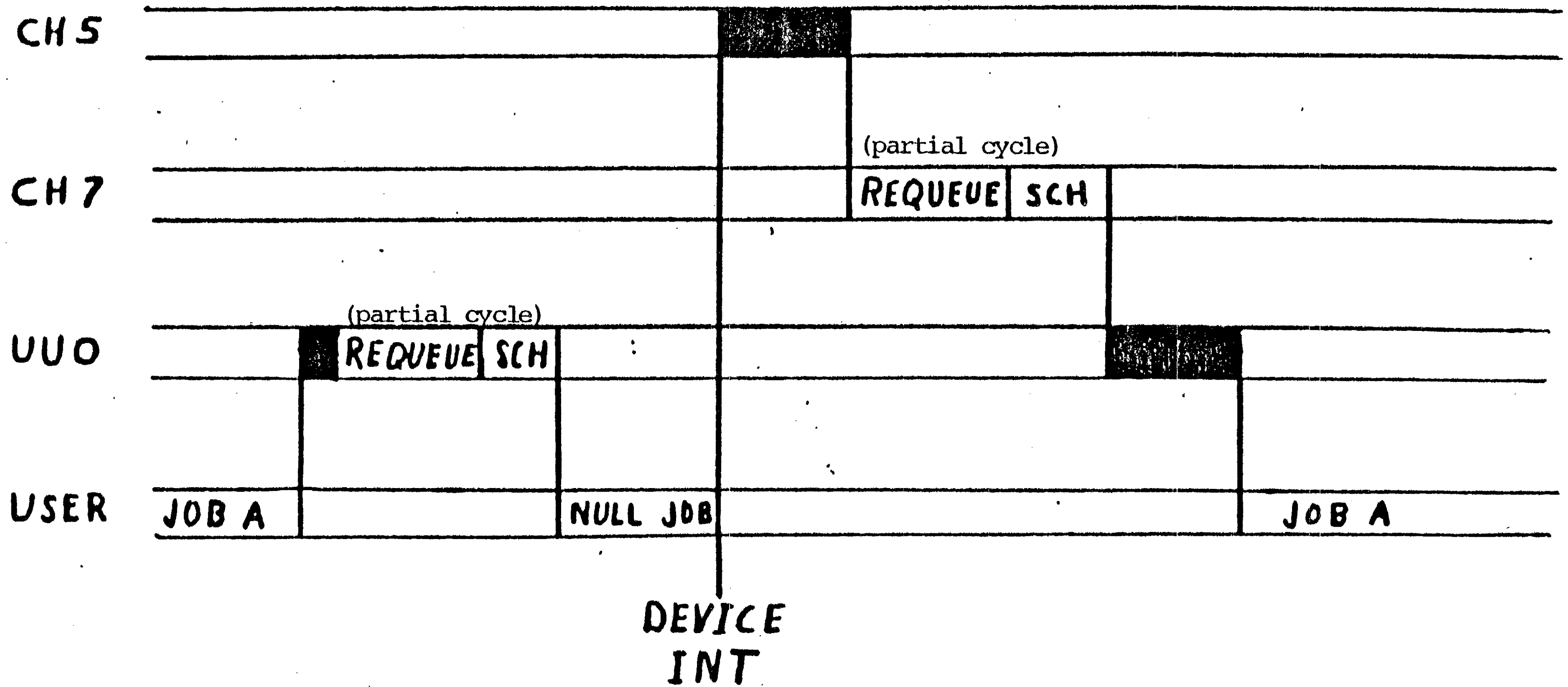
SCHEDULING



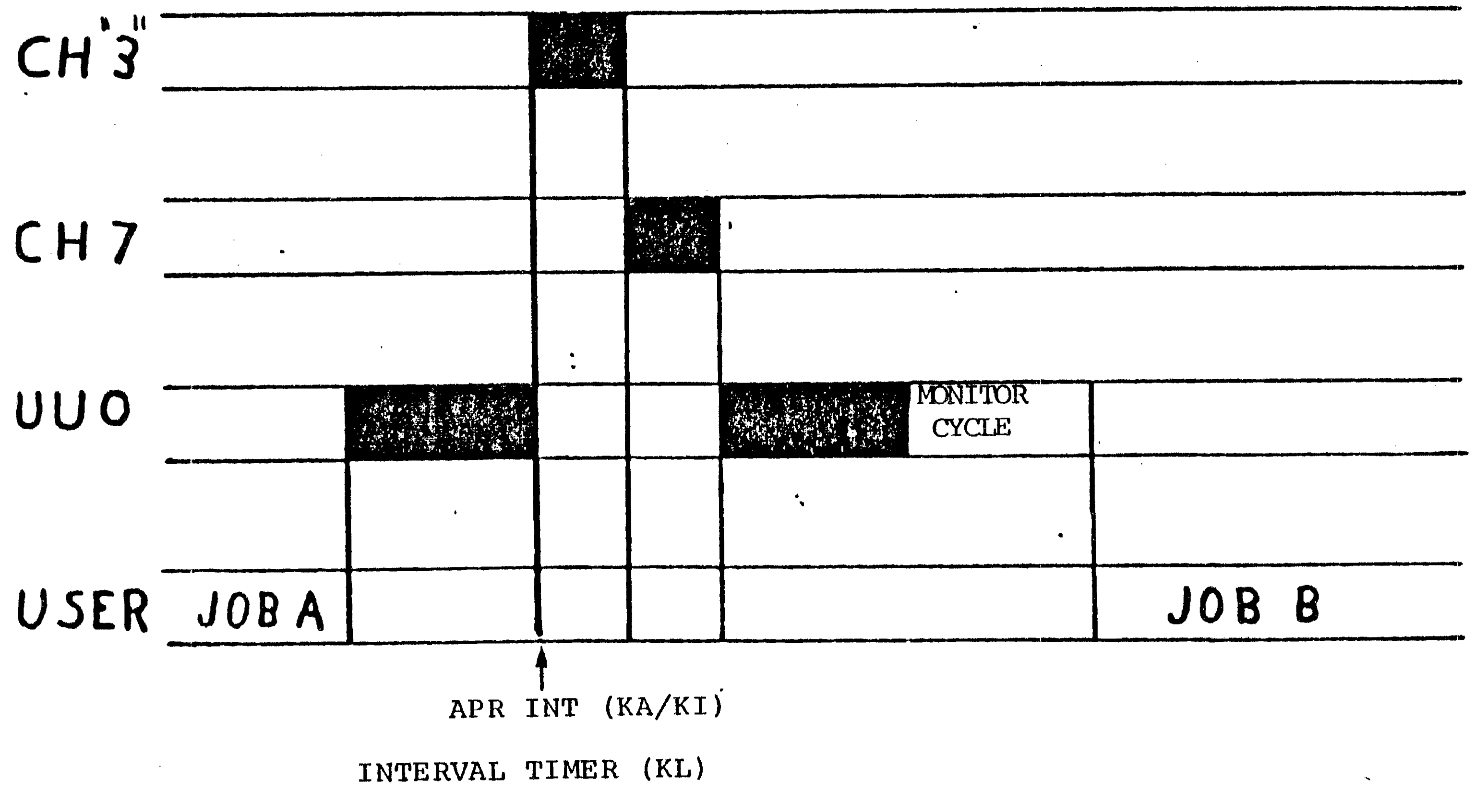
CLOCK TICK DURING INTERRUPT



I/O WAIT

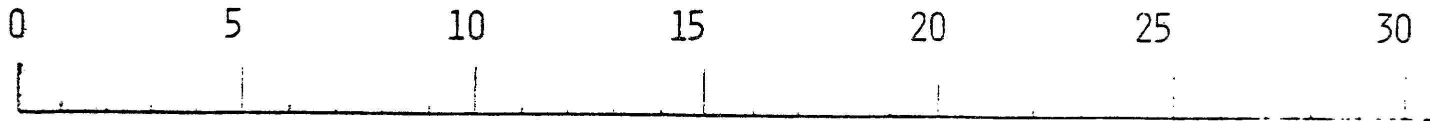


UUO INTERRUPTED



KL EBOX / MBOX TIME ACCOUNTING

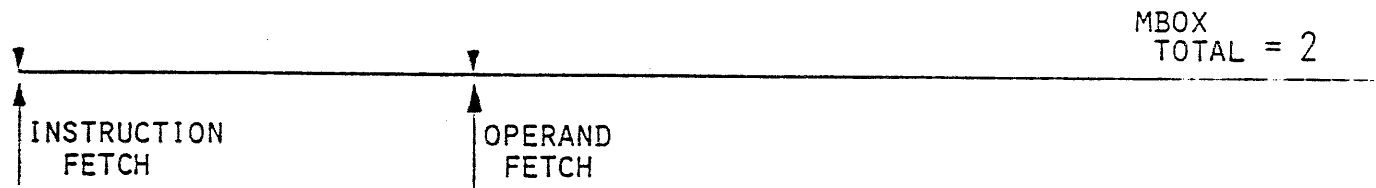
MACHINE CYCLE TIME



CASE # 1.

(LIGHT I/O)

MBOX REFERENCE COUNTS



EBOX BUSY TIME

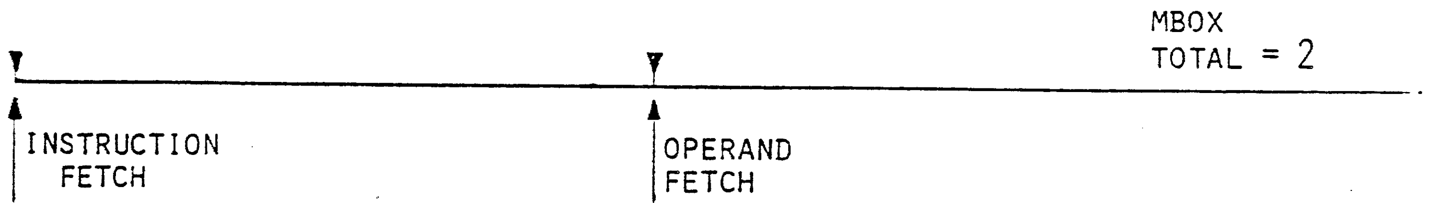
CASE 1 TOTAL = 12

TOTAL TIME = 18

CASE # 2.

(HEAVY I/O)

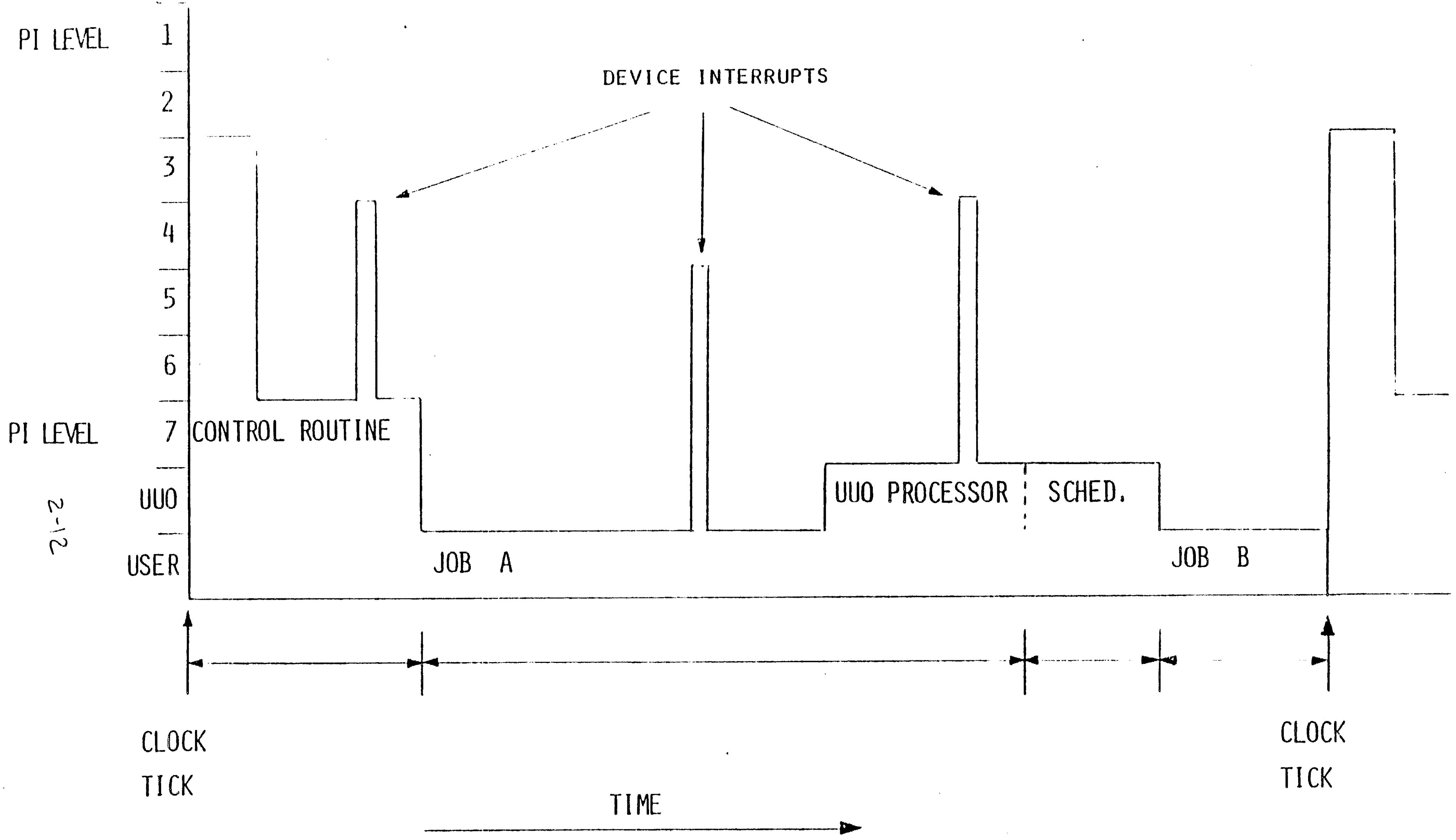
MBOX REFERENCE COUNTS

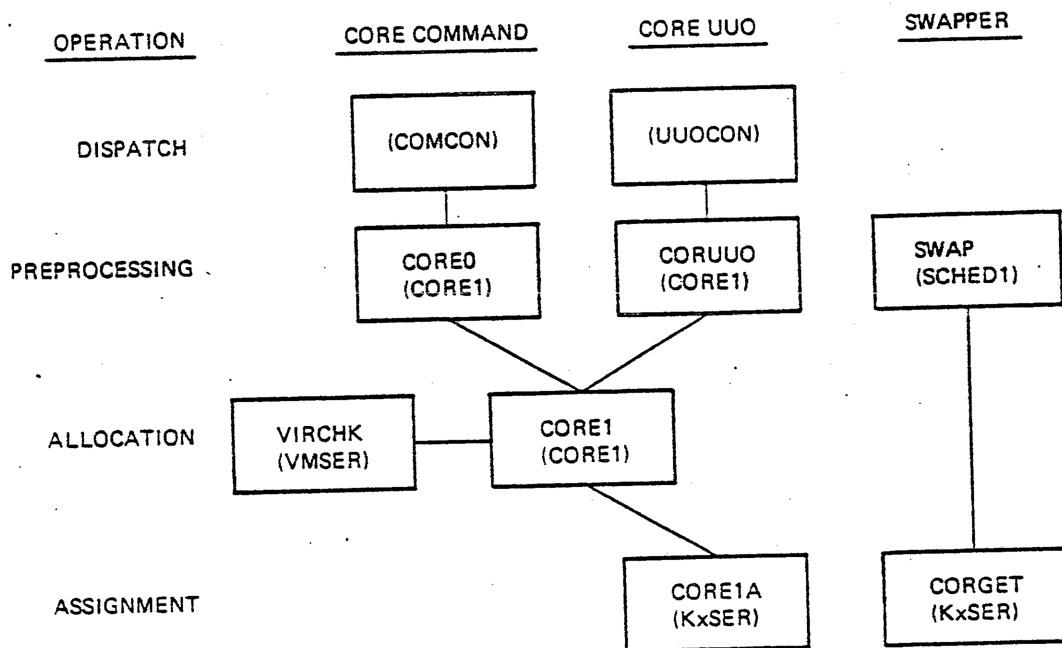


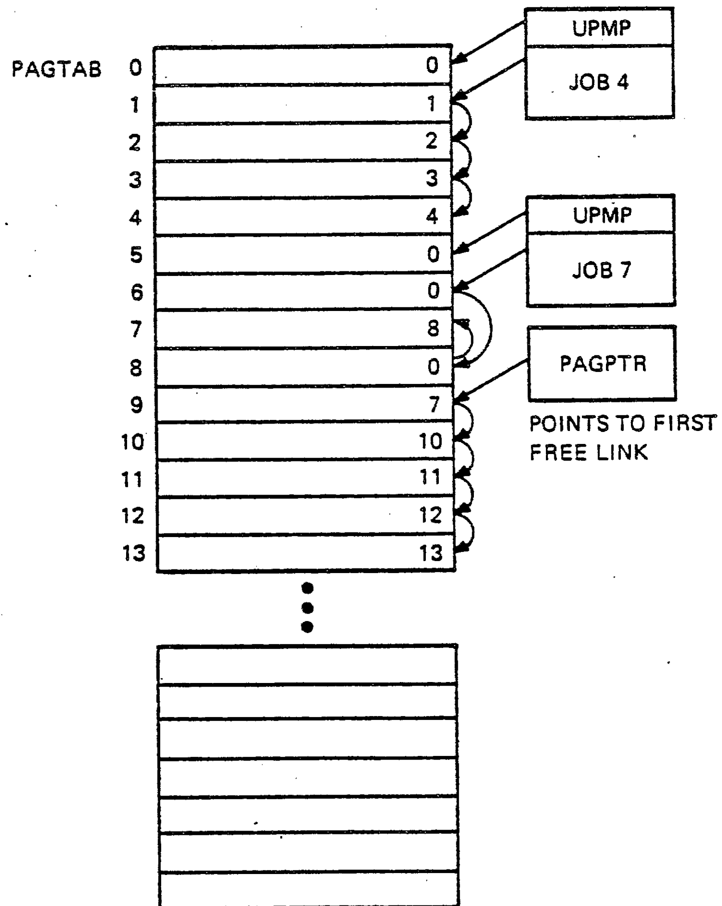
EBOX BUSY TIME

CASE 2 TOTAL = 12

TOTAL TIME = 26


TIME ACCOUNTING
PI LEVEL
1
2
3
4
5
6
PI LEVEL
7
UUO
USER
DEVICE INTERRUPTS
CONTROL ROUTINE
JOB A
UUO PROCESSOR
SCHED.
JOB B
CLOCK
TICK
CLOCK
TICK
TIME

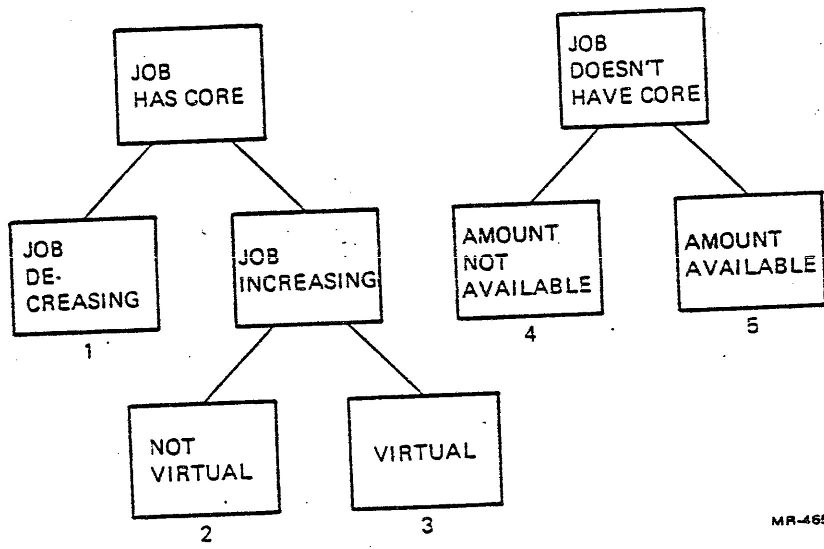

OPERATION
CORE COMMAND
CORE UUO
SWAPPER
DISPATCH
(COMCON)
(UUOCON)
PREPROCESSING
CORE0
(CORE1)
CORUUO
(CORE1)
SWAP
(SCHED1)
ALLOCATION
VIRCHK
(VMSER)
CORE1
(CORE1)
ASSIGNMENT
CORE1A
(KxSER)
CORGET
(KxSER)
MR-4648



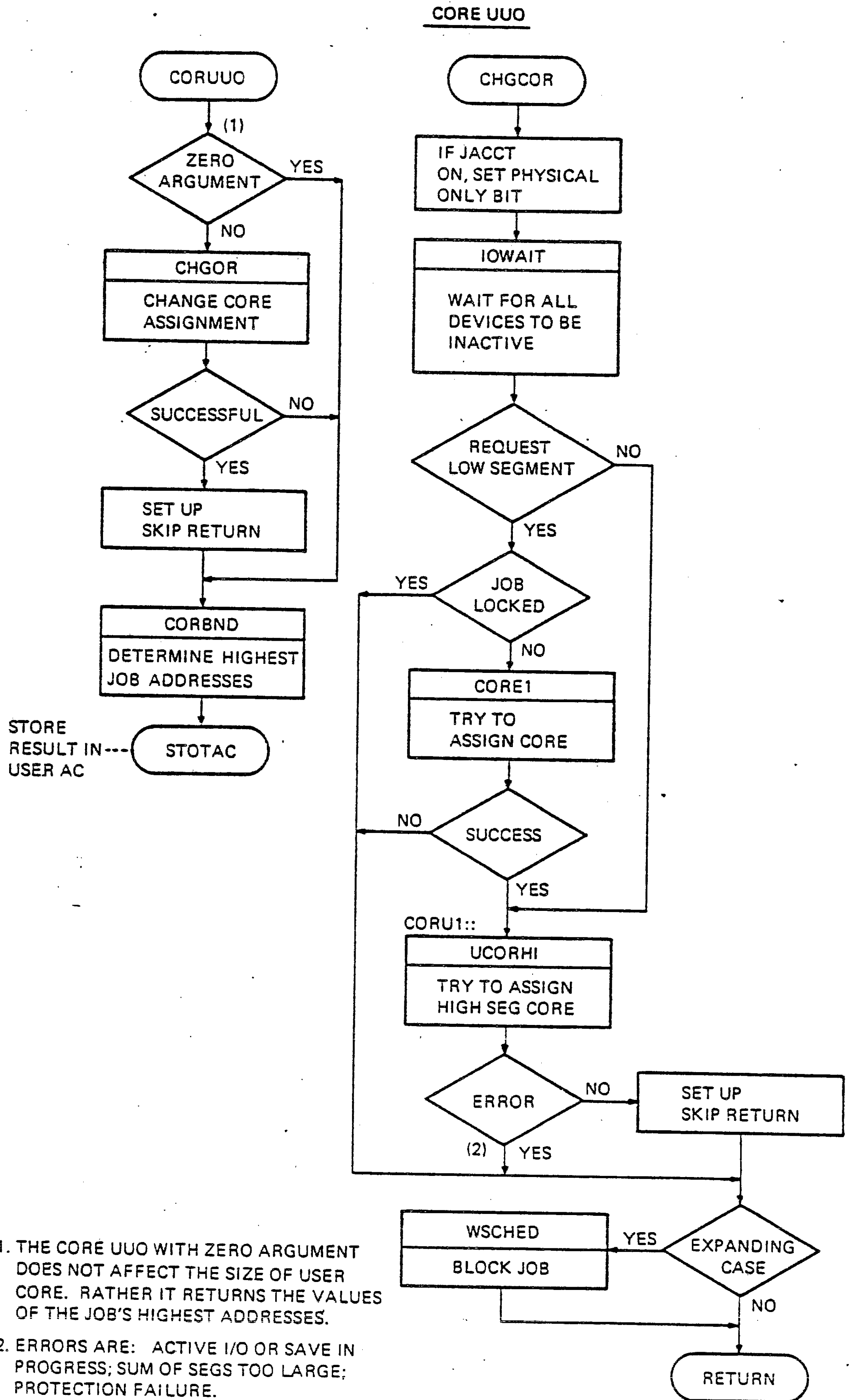
NOTE:

1. A ZERO ENTRY INDICATES THE END OF THE CHAIN.
2. ALL FREE PAGES ARE LINKED TOGETHER ALSO.

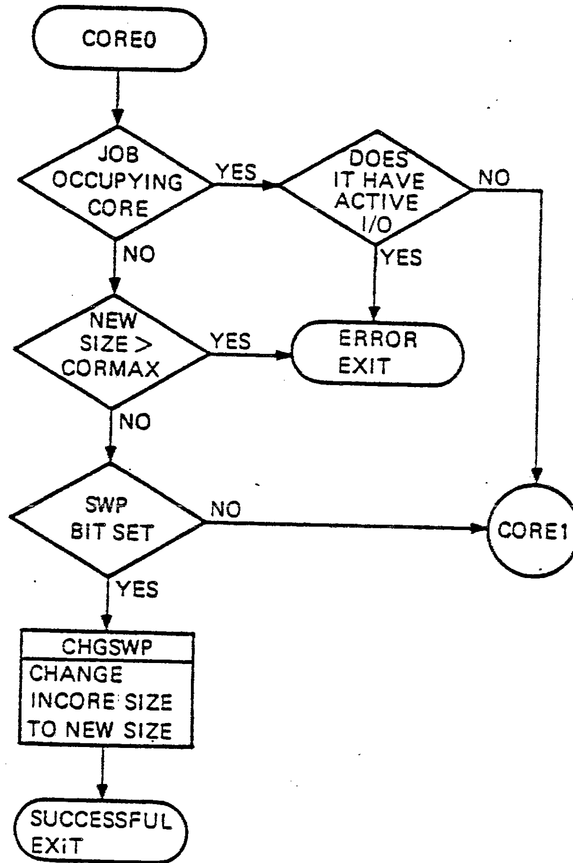
MR-4647


JOB
HAS CORE
JOB
DE-
CREASING
1
JOB
INCREASING
NOT
VIRTUAL
2
VIRTUAL
3
JOB
DOESN'T
HAVE CORE
AMOUNT
NOT
AVAILABLE
4
AMOUNT
AVAILABLE
5
MR-4653

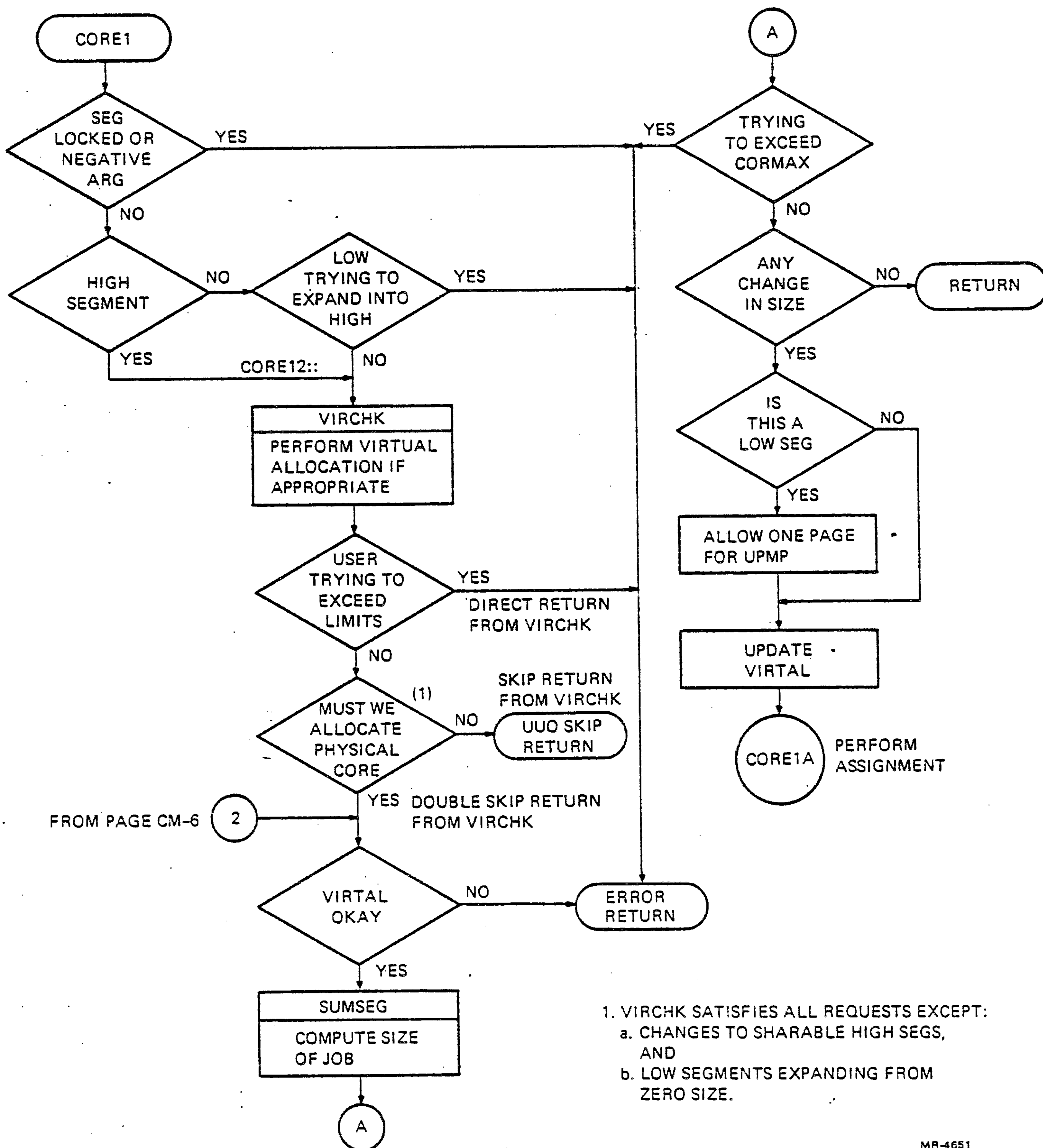
CORE UUO



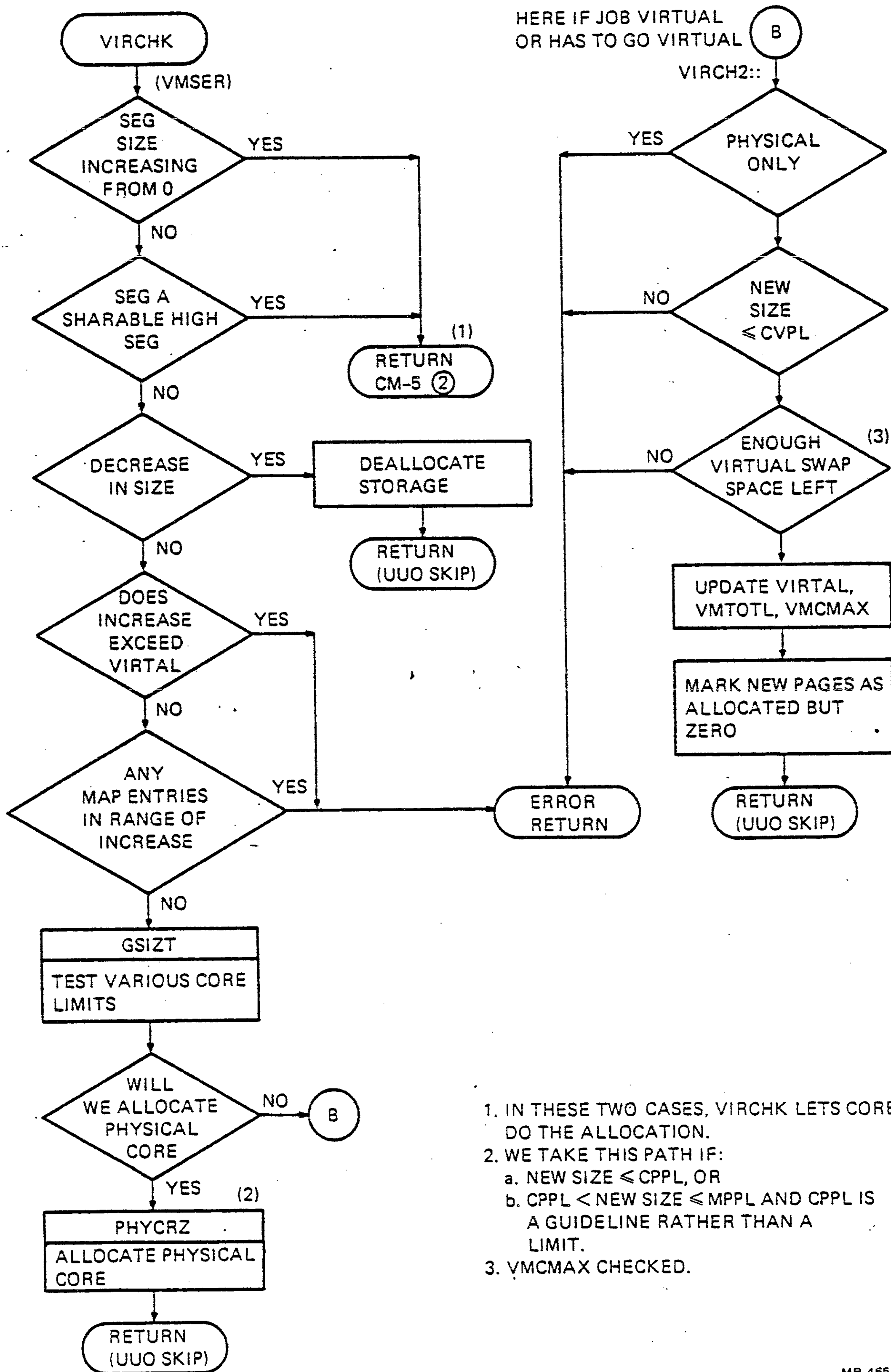
1. THE CORE UUO WITH ZERO ARGUMENT DOES NOT AFFECT THE SIZE OF USER CORE. RATHER IT RETURNS THE VALUES OF THE JOB'S HIGHEST ADDRESSES.
2. ERRORS ARE: ACTIVE I/O OR SAVE IN PROGRESS; SUM OF SEGS TOO LARGE; PROTECTION FAILURE.


CORE COMMAND
CORE0
JOB OCCUPYING CORE
YES
NO
DOES IT HAVE ACTIVE I/O
NO
YES
NEW SIZE > CORMAX
YES
NO
ERROR EXIT
SWP BIT SET
NO
YES
CORE1
CHGSWP
CHANGE INCORE SIZE TO NEW SIZE
SUCCESSFUL EXIT
MR-4649

CORE ALLOCATION

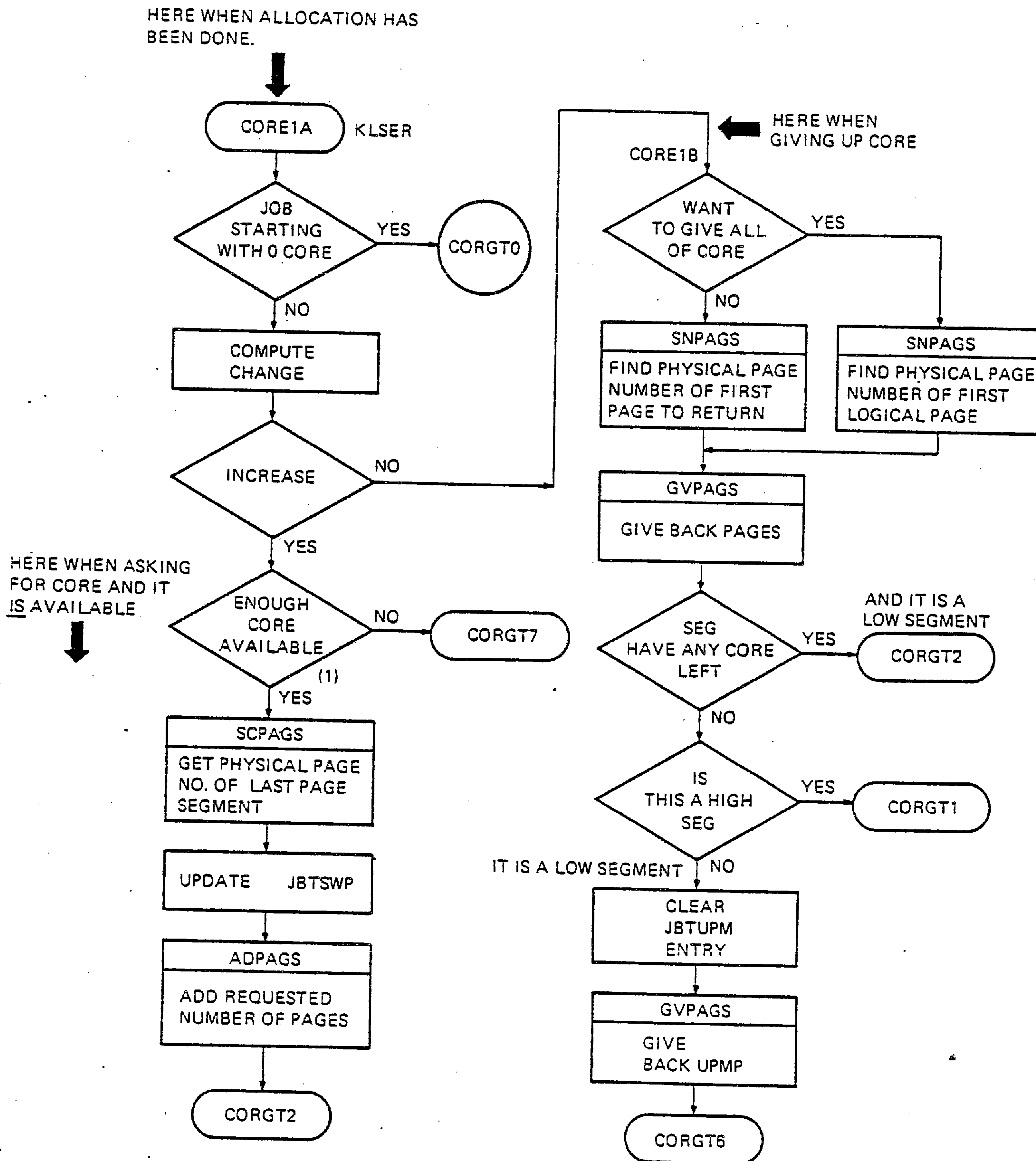


1. VIRCHK SATISFIES ALL REQUESTS EXCEPT:
   a. CHANGES TO SHARABLE HIGH SEGS, AND
   b. LOW SEGMENTS EXPANDING FROM ZERO SIZE.



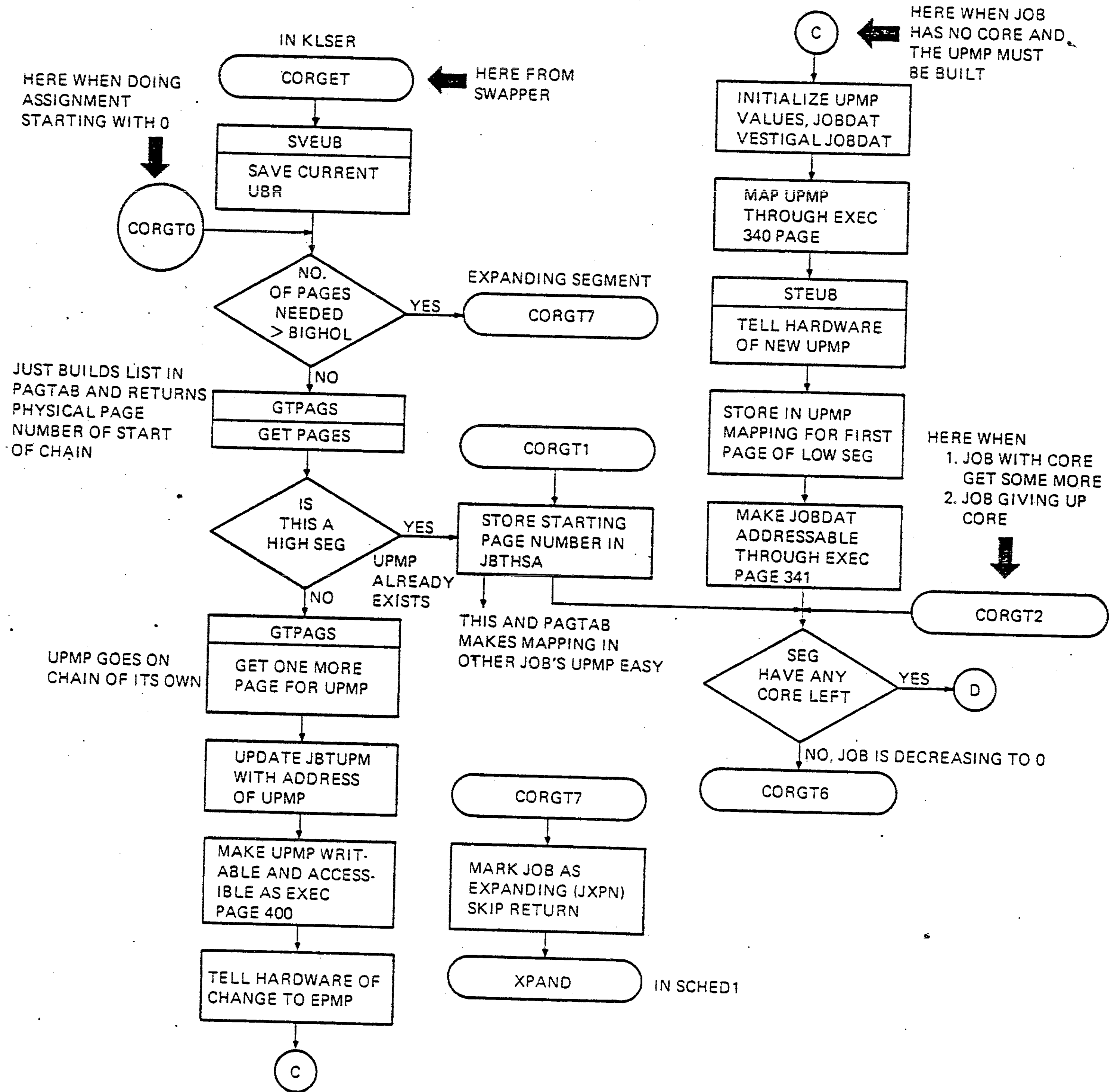
1. IN THESE TWO CASES, VIRCHK LETS CORE 1 DO THE ALLOCATION.
2. WE TAKE THIS PATH IF:
   a. NEW SIZE ≤ CPPL, OR
   b. CPPL < NEW SIZE ≤ MPPL AND CPPL IS A GUIDELINE RATHER THAN A LIMIT.
3. VMCMAX CHECKED.

MR-4652

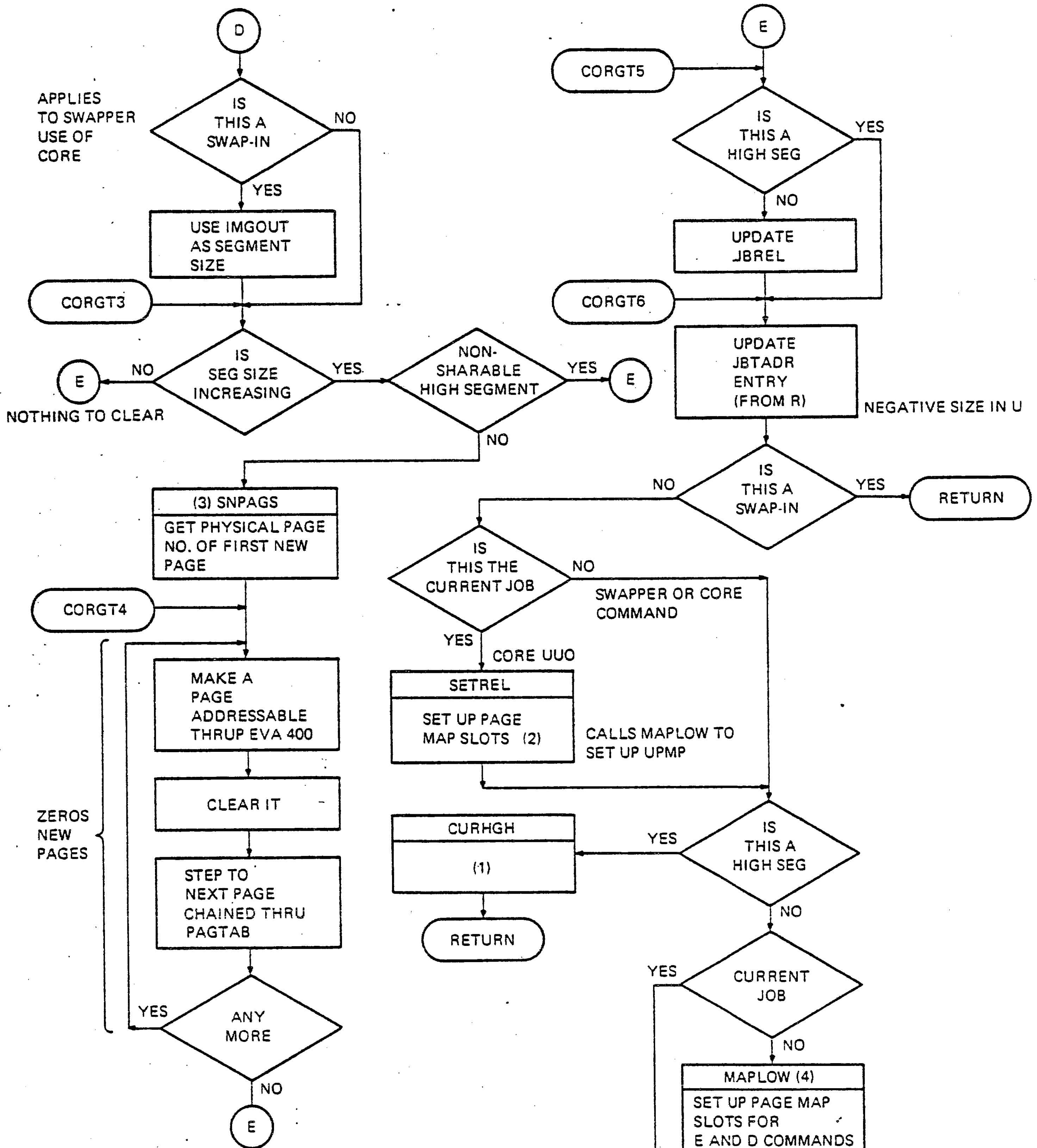


(1) CALLS FRDCR IN SEGCON TO FREE DORMANT AND IDLE HIGH

MR-4654



IN KLSER
CORGET
HERE FROM SWAPPER
HERE WHEN DOING ASSIGNMENT STARTING WITH 0
SVEUB
SAVE CURRENT UBR
CORGT0
NO. OF PAGES NEEDED > BIGHOL
YES
EXPANDING SEGMENT
CORGT7
NO
JUST BUILDS LIST IN PAGTAB AND RETURNS PHYSICAL PAGE NUMBER OF START OF CHAIN
GTPAGS
GET PAGES
CORGT1
IS THIS A HIGH SEG
YES
STORE STARTING PAGE NUMBER IN JBTHSA
UPMP ALREADY EXISTS
NO
THIS AND PAGTAB MAKES MAPPING IN OTHER JOB'S UPMP EASY
UPMP GOES ON CHAIN OF ITS OWN
GTPAGS
GET ONE MORE PAGE FOR UPMP
UPDATE JBTUPM WITH ADDRESS OF UPMP
MAKE UPMP WRIT-ABLE AND ACCESS-IBLE AS EXEC PAGE 400
TELL HARDWARE OF CHANGE TO EPMP
C
CORGT7
MARK JOB AS EXPANDING (JXPN) SKIP RETURN
XPAND
IN SCHED1
C
HERE WHEN JOB HAS NO CORE AND THE UPMP MUST BE BUILT
INITIALIZE UPMP VALUES, JOBDAT VESTIGAL JOBDAT
MAP UPMP THROUGH EXEC 340 PAGE
STEUB
TELL HARDWARE OF NEW UPMP
STORE IN UPMP MAPPING FOR FIRST PAGE OF LOW SEG
HERE WHEN
1. JOB WITH CORE GET SOME MORE
2. JOB GIVING UP CORE
MAKE JOBDAT ADDRESSABLE THROUGH EXEC PAGE 341
CORGT2
SEG HAVE ANY CORE LEFT
YES
D
NO, JOB IS DECREASING TO 0
CORGT6
MR-4656



1. IF CHANGE IS TO CURRENT HIGH SEG, UPDATE CURRENT JOB'S PAGE MAP.
2. SET UP R AND ADDRESS BREAK.
3. JOBDAT IS ALWAYS THE FIRST PAGE IN THE CHAIN. PICK UP THE PHYSICAL PAGE NUMBER FROM RH UPMP LOCATION 400 TO USE AS THE PAGTAB INDEX. CHAIN DOWN 0 IF HAD NO CORE OR NO. PAGES TO NEW ASSIGNMENT.
4. GIVEN STARTING PHYSICAL PAGE NUMBER, FILL THE UDMP SLOTS BY CHAINING DOWN PAGTAB AND EXTRACTING THE PHYSICAL PAGE NUMBERS FOR PLACEMENT IN THE UPMP.

MR-4655

USER PAGE MAP (UPMP) MAPPING ENTRY

ONE ENTRY PER PAGE (18 BITS)

18 BIT QUANTITY – 512 PER UPMP

A = 0 ACCESS DENIED, PAGE FAULT OCCURS

= 1 ACCESS ALLOWED

P = 0 CONCEALED PAGE (EXECUTE ONLY)

= 1 PUBLIC PAGE

W = 0 WRITE PROTECTED

= 1 WRITABLE

S = 0 ALLOCATED

= 1 ALLOCATED BUT ZERO

C = 0 CACHEABLE

= 1 NOT CACHEABLE

PAGE ADDRESS – 13 BIT PHYSICAL MEMORY PAGE NUMBER OR
– 17 BIT SWAPPING SPACE ADDRESS

TO FIND THE STATUS OF ANY PARTICULAR PAGE, USE THESE GUIDELINES:

1. IF A = 1 AND S = 0, THE PAGE IS IN CORE AT THE ADDRESS SPECIFIED BY PAGE-ADDRESS.

2. IF A = 0, S = 0, AND C = 0, THE PAGE DOES NOT EXIST.

3. IF A = 0 AND THE WSBTAB ENTRY FOR THIS PAGE = 1, THE PAGE IS IN CORE AT PAGE-ADDRESS.

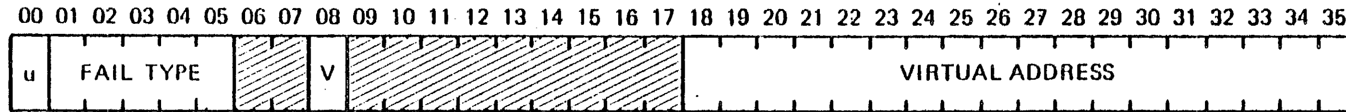
4. IF A = 0, AABTAB = 1 AND WSBTAB = 0, THE ENTRY CONTAINS A DISK ADDRESS.

5. IF A = 0, S = 1, AND AABTAB = 0, THE PAGE IS ALLOCATED BUT ZERO.

MR-4661

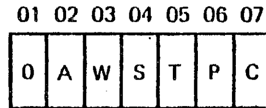
PAGE FAIL WORD

KL10



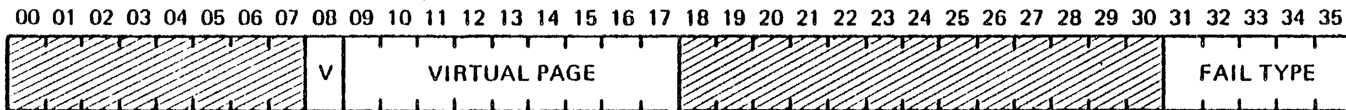
1 = USER VIRTUAL ADDRESS
0 = EXECUTIVE VIRTUAL ADDRESS

IF BIT 1 = 0, THEN BITS 1-7 ARE INTERPRETED AS FOLLOWS:

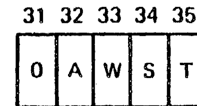


0 = READ ONLY
1 = WRITE

KI10



IF BIT 31 = 0, BITS 31-35 ARE INTERPRETED AS FOLLOWS:



MR-4662

ARGUMENT BLOCK IN THE PFH

OLD PC
PAGE FAULT WORD
VIRTUAL TIME
PAGE RATE
PSI VECTOR ADDRESS

THE PAGE FAULT WORD IS ORGANIZED AS FOLLOWS:

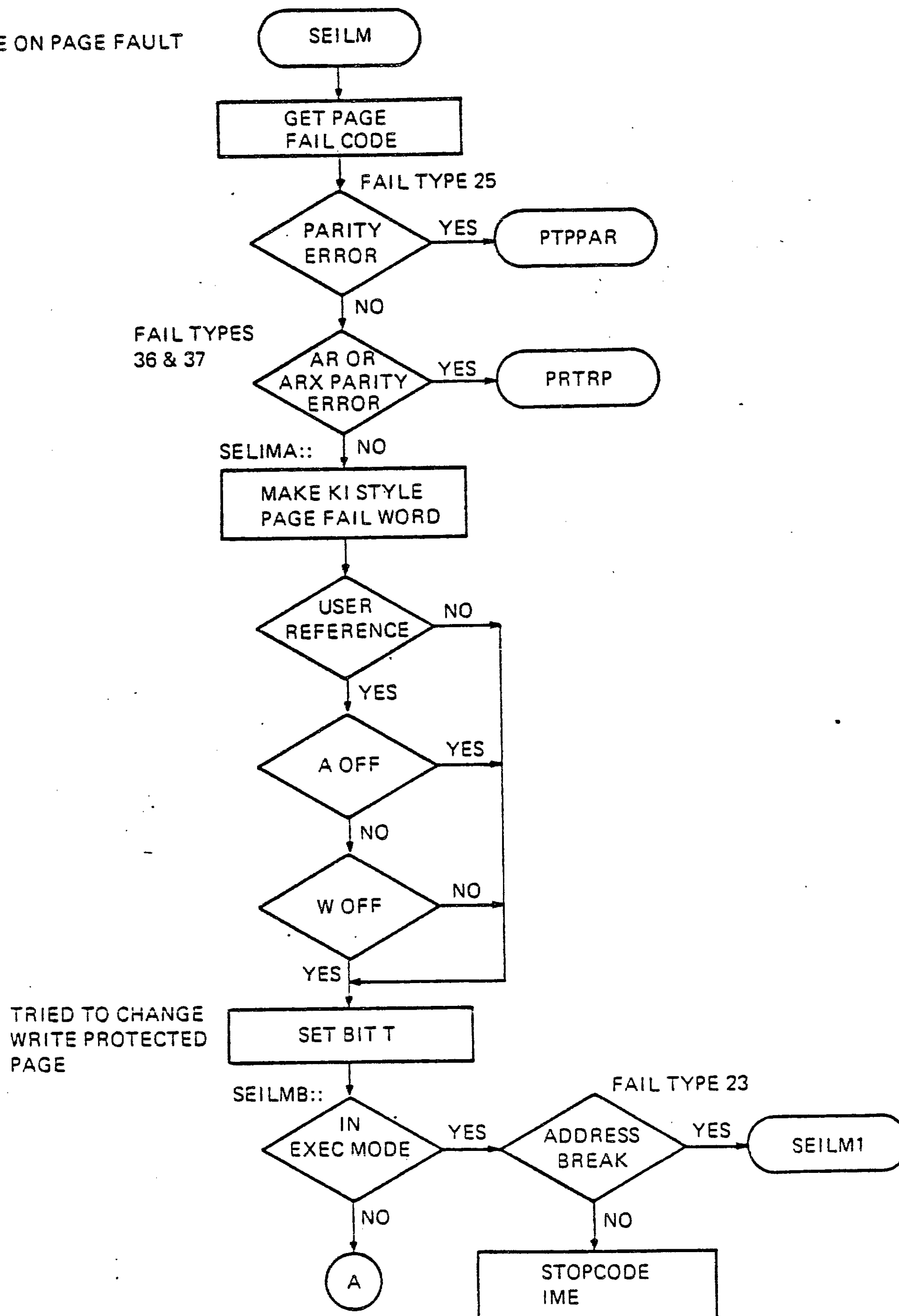
0	1–17	18–35
	PAGE NUMBER	FAULT TYPE

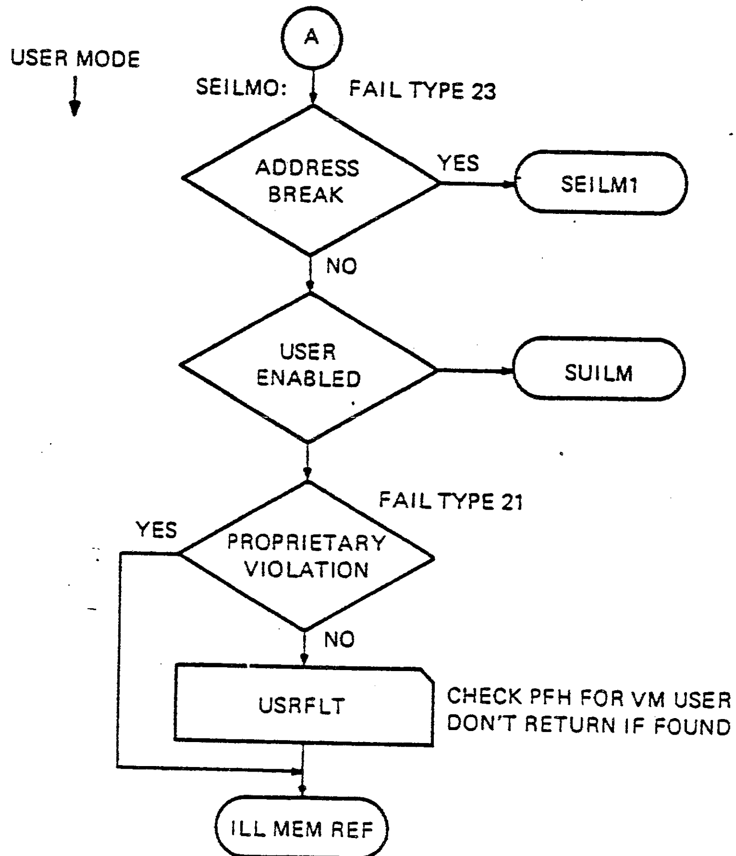
Bit 0: SET IF WORKING SET CHANGED

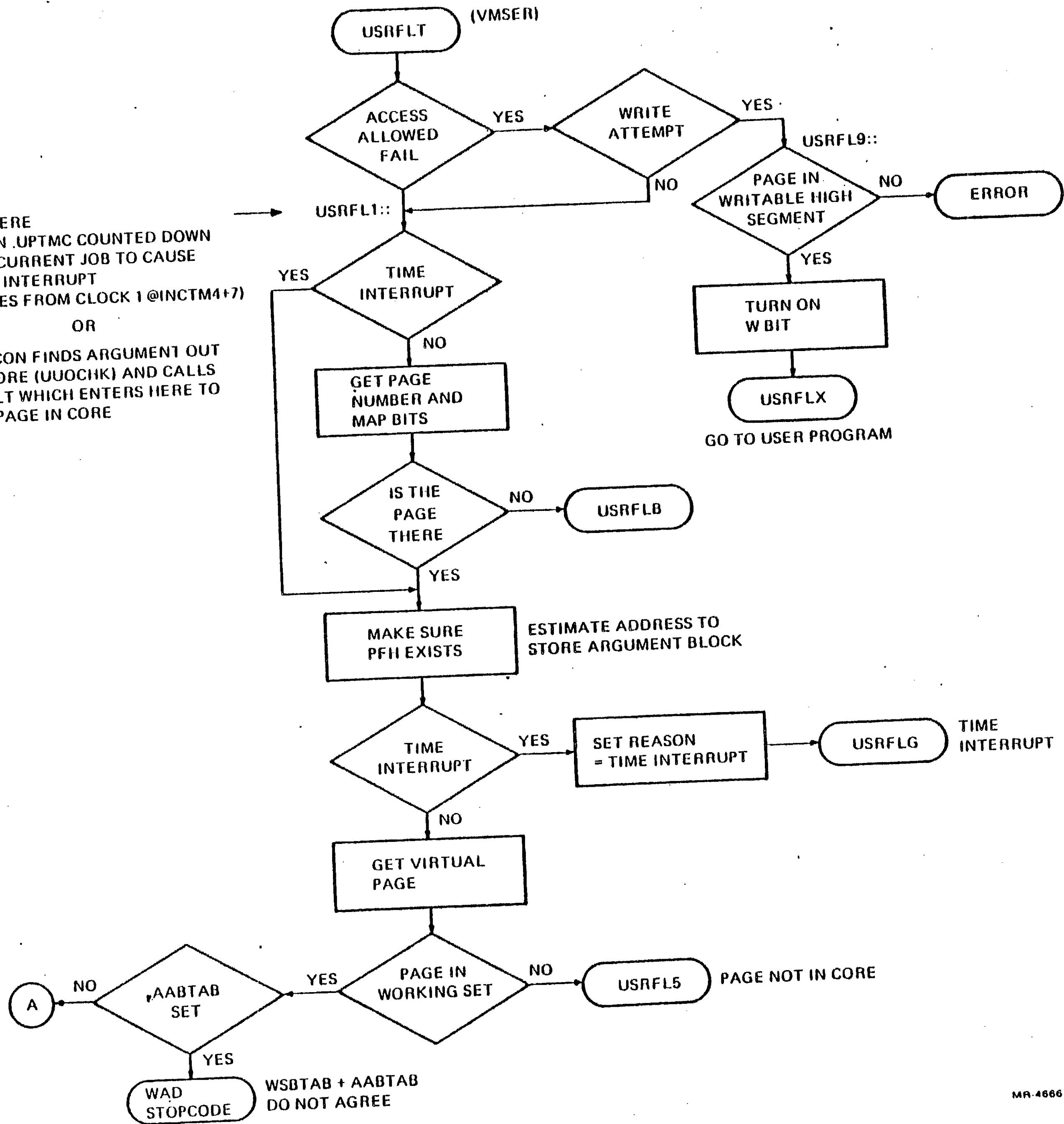
THE FAULT TYPES ARE:

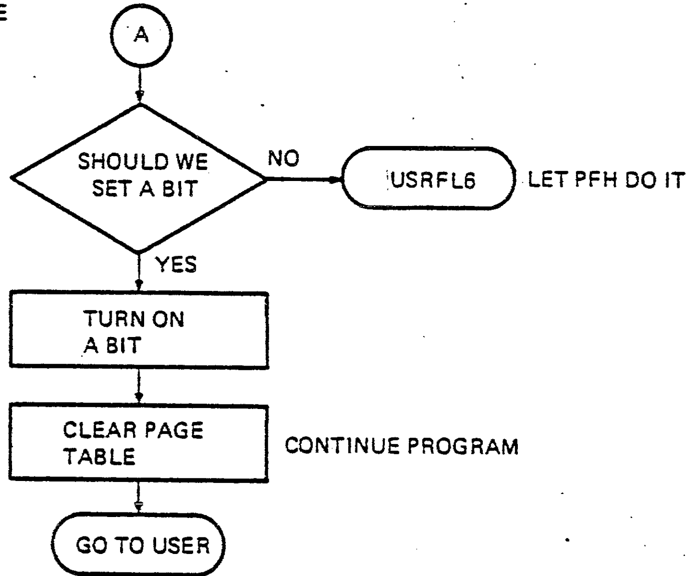
1 A OFF
2 PAGE NOT IN WORKING SET
3 PAGE CONTAINING UUO ARGUMENT NOT IN WORKING SET
4 TIME TRAP
5 ALLOCATED BUT ZERO (ABZ)
6 ABZ AFTER UUO

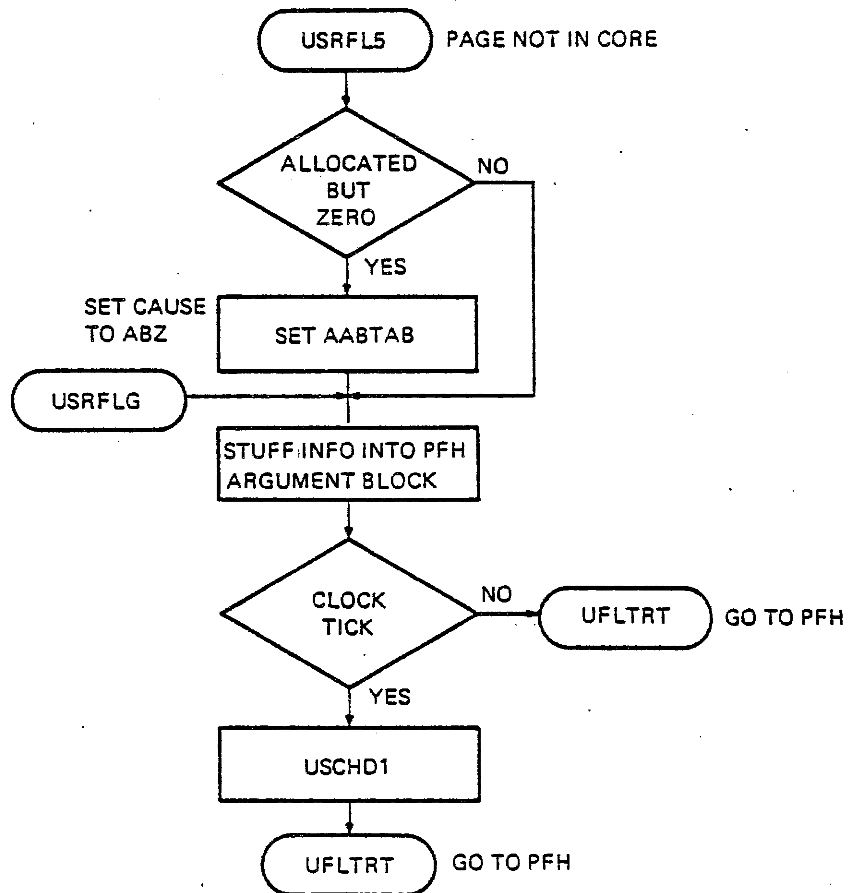
MR-4665


HERE ON PAGE FAULT
SEILM
GET PAGE FAIL CODE
FAIL TYPE 25
PARITY ERROR
YES
PTPPAR
NO
FAIL TYPES 36 & 37
AR OR ARX PARITY ERROR
YES
PRTRP
SELIMA::
NO
MAKE KI STYLE PAGE FAIL WORD
USER REFERENCE
NO
YES
A OFF
YES
NO
W OFF
NO
YES
TRIED TO CHANGE WRITE PROTECTED PAGE
SET BIT T
SEILMB::
FAIL TYPE 23
IN EXEC MODE
YES
ADDRESS BREAK
YES
SEILM1
NO
NO
A
STOPCODE IME
MR-4663


USER MODE
A
SEILMO:
FAIL TYPE 23
ADDRESS BREAK
YES
SEILM1
NO
USER ENABLED
SUILM
FAIL TYPE 21
YES
PROPRIETARY VIOLATION
NO
USRFLT
CHECK PFH FOR VM USER
DON'T RETURN IF FOUND
ILL MEM REF
MR-4664


USRFLT
(VMSER)
ACCESS ALLOWED FAIL
YES
WRITE ATTEMPT
YES
USRFL9::
NO
PAGE IN WRITABLE HIGH SEGMENT
NO
ERROR
YES
TURN ON W BIT
USRFLX
GO TO USER PROGRAM
ENTER HERE
1. WHEN .UPTMC COUNTED DOWN FOR CURRENT JOB TO CAUSE TIME INTERRUPT (COMES FROM CLOCK 1 @INCTM4+7)
OR
2. UUOCON FINDS ARGUMENT OUT OF CORE (UUOCHK) AND CALLS WOFLT WHICH ENTERS HERE TO GET PAGE IN CORE
USRFL1::
YES
TIME INTERRUPT
NO
GET PAGE NUMBER AND MAP BITS
IS THE PAGE THERE
NO
USRFLB
YES
MAKE SURE PFH EXISTS
ESTIMATE ADDRESS TO STORE ARGUMENT BLOCK
TIME INTERRUPT
YES
SET REASON = TIME INTERRUPT
USRFLG
TIME INTERRUPT
NO
GET VIRTUAL PAGE
PAGE IN WORKING SET
NO
USRFL5
PAGE NOT IN CORE
YES
AABTAB SET
NO
A
YES
WAD STOPCODE
WSBTAB + AABTAB DO NOT AGREE
MR-4666

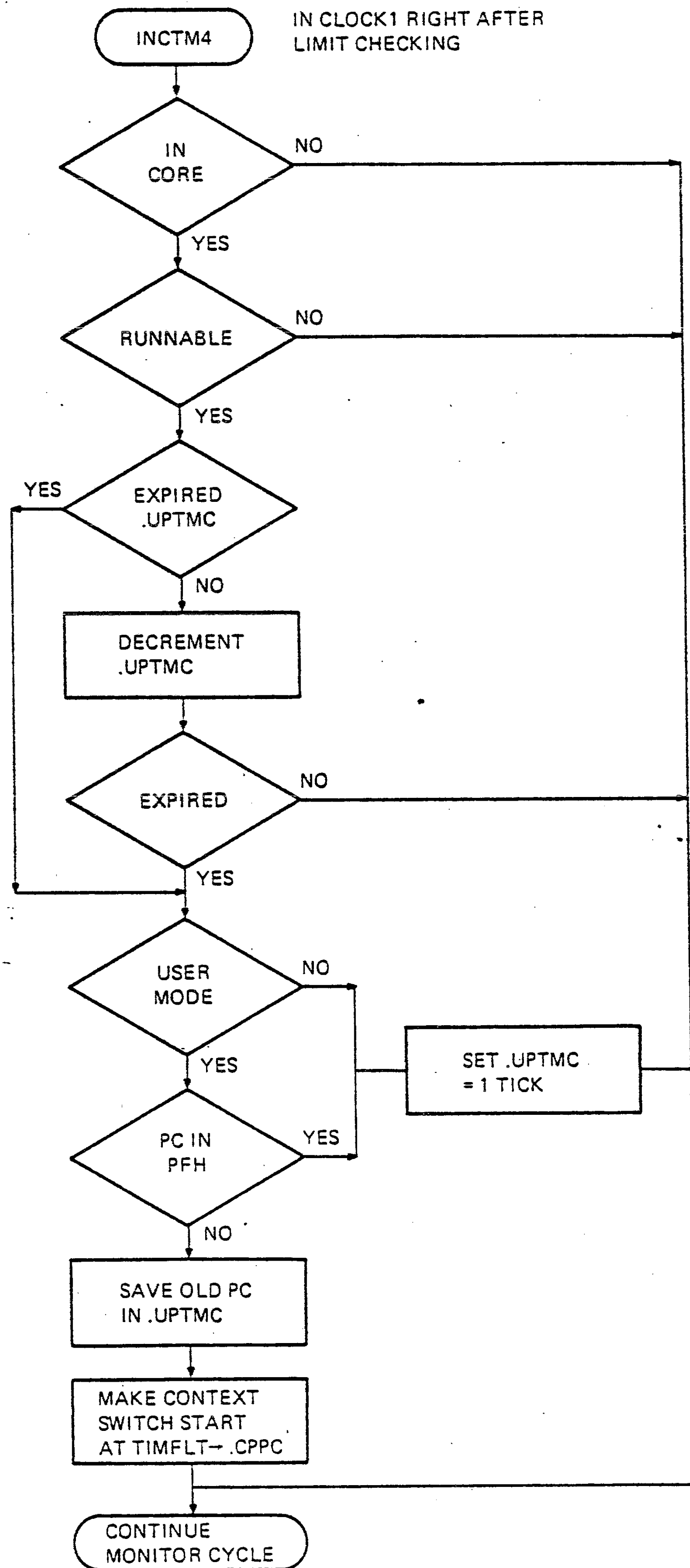

PAGE ACCESSIBLE
BUT NOT
IN CORE
A
SHOULD WE
SET A BIT
NO
USRFL6
LET PFH DO IT
YES
TURN ON
A BIT
CLEAR PAGE
TABLE
CONTINUE PROGRAM
GO TO USER
MR-4667

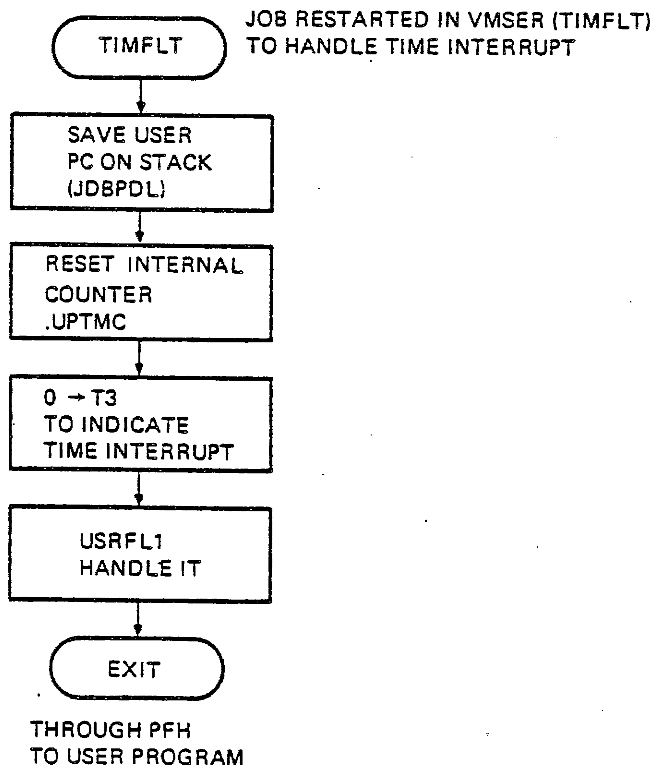


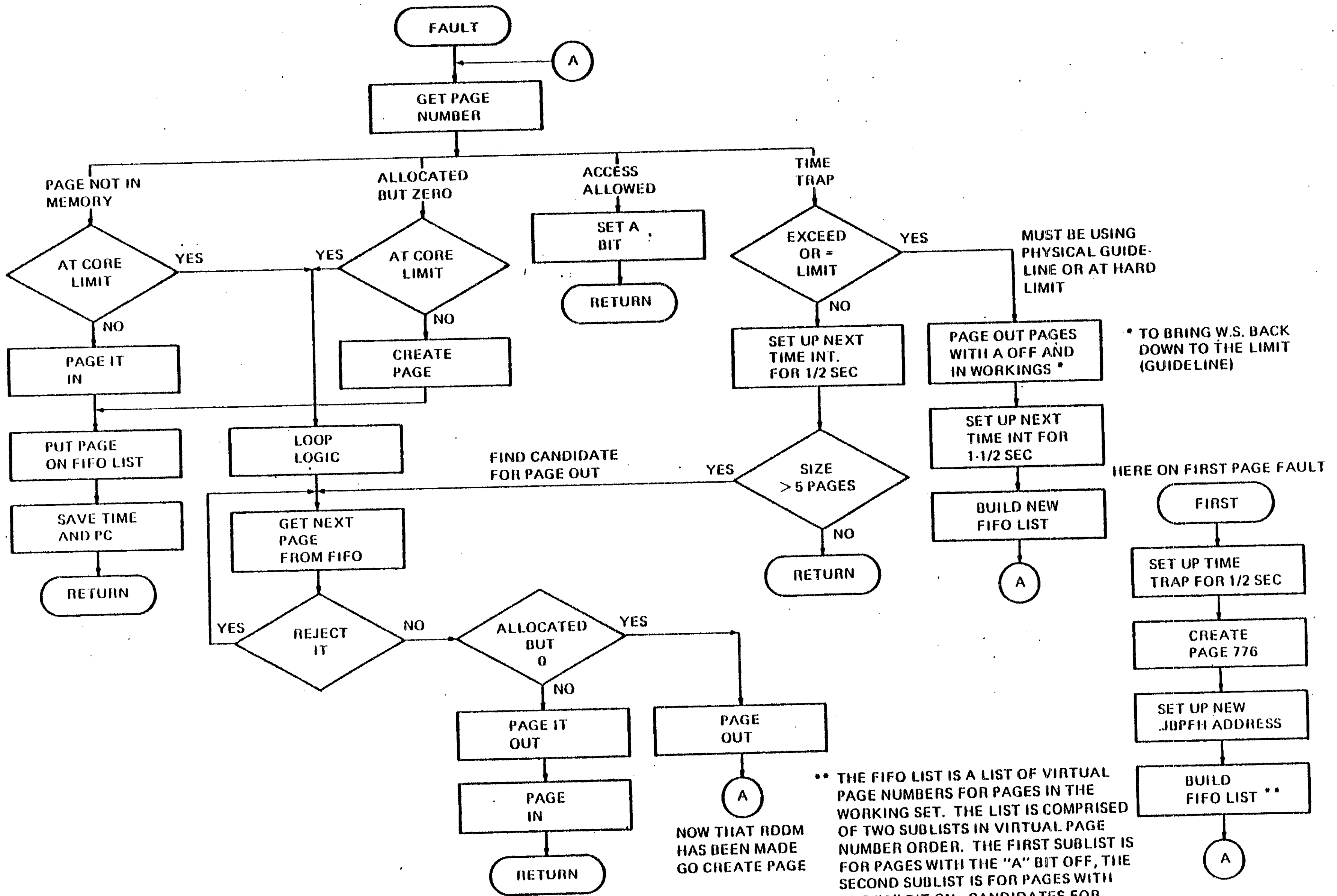
THE FOLLOWING INFORMATION IS RETURNED TO THE PFH:

1. OLD PC
2. PAGE FAULT WORD
3. VIRTUAL TIME
4. PAGE RATE

MR-4668

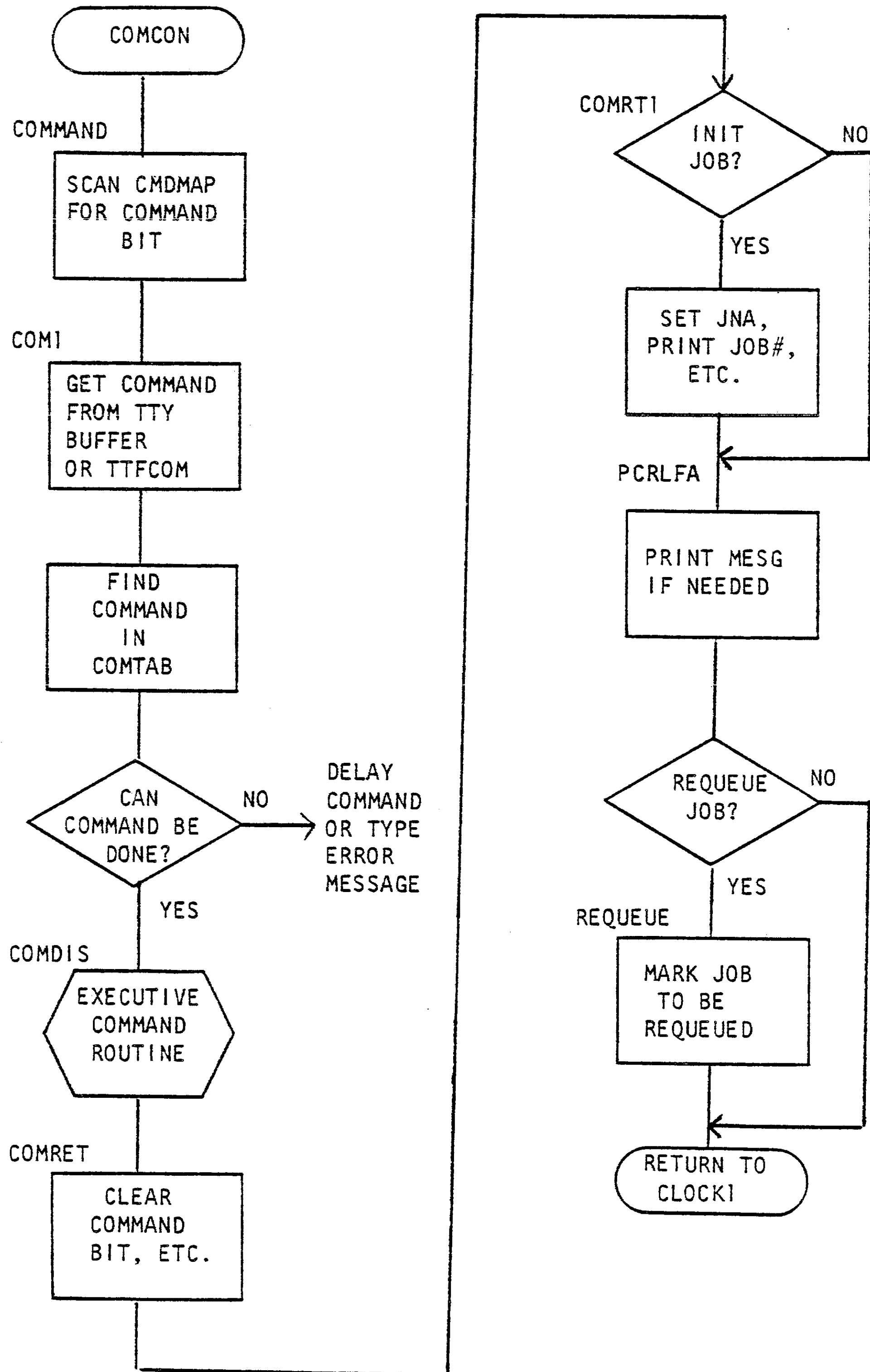

VIRTUAL TIME TRAPPING
INCTM4
IN CLOCK1 RIGHT AFTER LIMIT CHECKING
IN CORE
NO
YES
RUNNABLE
NO
YES
EXPIRED .UPTMC
YES
NO
DECREMENT .UPTMC
EXPIRED
NO
YES
USER MODE
NO
YES
SET .UPTMC = 1 TICK
PC IN PFH
YES
NO
SAVE OLD PC IN .UPTMC
MAKE CONTEXT SWITCH START AT TIMFLT→ .CPPC
CONTINUE MONITOR CYCLE
MR-4669


JOB RESTARTED IN VMSER (TIMFLT)
TO HANDLE TIME INTERRUPT
TIMFLT
SAVE USER
PC ON STACK
(JDBPDL)
RESET INTERNAL
COUNTER
.UPTMC
0 → T3
TO INDICATE
TIME INTERRUPT
USRFL1
HANDLE IT
EXIT
THROUGH PFH
TO USER PROGRAM
MR-4670

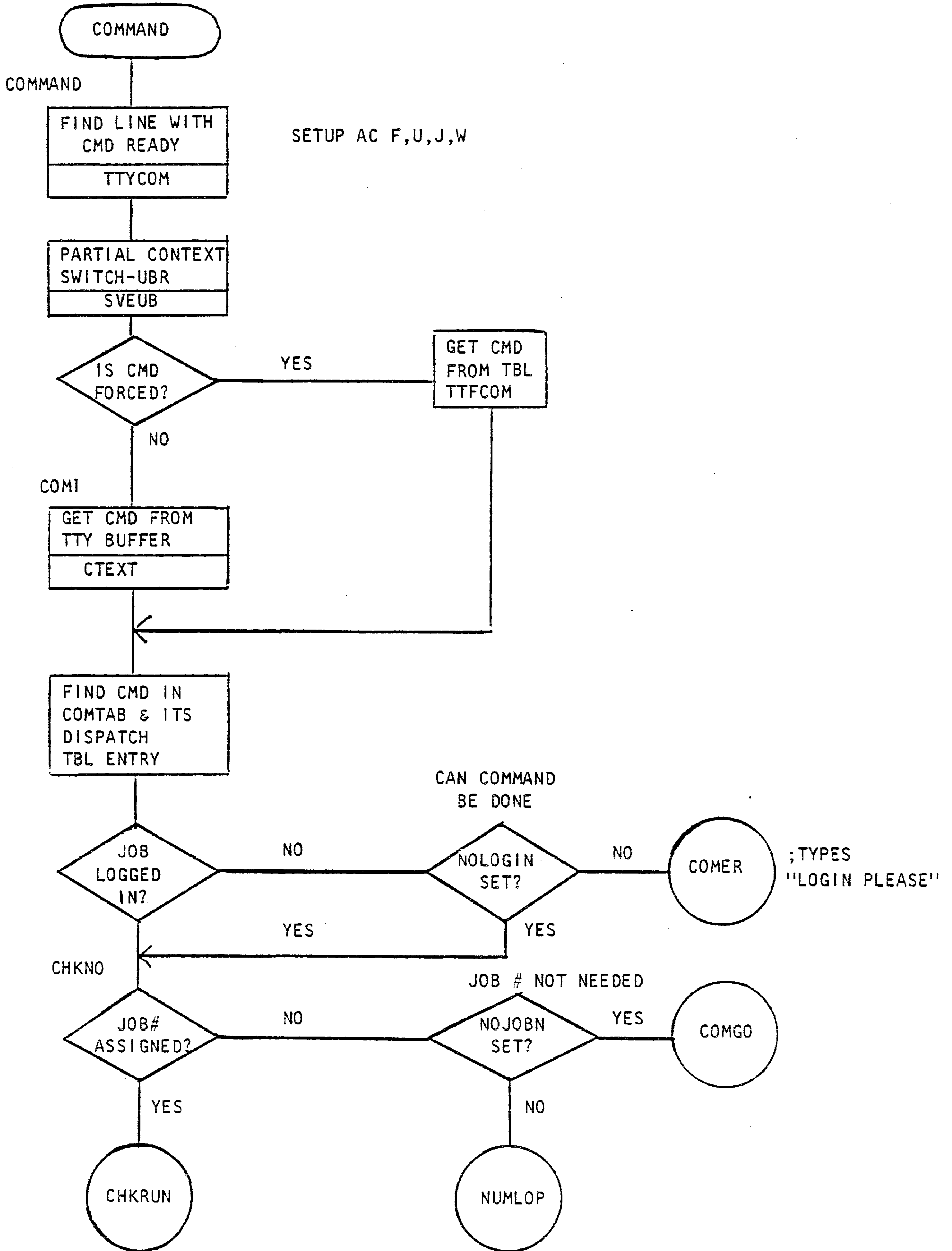


FAULT
A
GET PAGE NUMBER
PAGE NOT IN MEMORY
ALLOCATED BUT ZERO
ACCESS ALLOWED
TIME TRAP
AT CORE LIMIT
YES
NO
PAGE IT IN
PUT PAGE ON FIFO LIST
SAVE TIME AND PC
RETURN
AT CORE LIMIT
YES
NO
CREATE PAGE
LOOP LOGIC
SET A BIT
RETURN
EXCEED OR = LIMIT
YES
NO
MUST BE USING PHYSICAL GUIDE-LINE OR AT HARD LIMIT
SET UP NEXT TIME INT. FOR 1/2 SEC
PAGE OUT PAGES WITH A OFF AND IN WORKINGS *
* TO BRING W.S. BACK DOWN TO THE LIMIT (GUIDELINE)
SET UP NEXT TIME INT FOR 1-1/2 SEC
FIND CANDIDATE FOR PAGE OUT
SIZE > 5 PAGES
YES
NO
RETURN
BUILD NEW FIFO LIST
A
HERE ON FIRST PAGE FAULT
FIRST
SET UP TIME TRAP FOR 1/2 SEC
CREATE PAGE 776
SET UP NEW JBPFH ADDRESS
BUILD FIFO LIST **
A
GET NEXT PAGE FROM FIFO
REJECT IT
YES
NO
ALLOCATED BUT 0
YES
NO
PAGE IT OUT
PAGE IN
RETURN
PAGE OUT
A
NOW THAT ROOM HAS BEEN MADE GO CREATE PAGE
** THE FIFO LIST IS A LIST OF VIRTUAL PAGE NUMBERS FOR PAGES IN THE WORKING SET. THE LIST IS COMPRISED OF TWO SUBLISTS IN VIRTUAL PAGE NUMBER ORDER. THE FIRST SUBLIST IS FOR PAGES WITH THE "A" BIT OFF, THE SECOND SUBLIST IS FOR PAGES WITH THE "A" BIT ON. CANDIDATES FOR PAGE-OUT COME FROM THE FRONT OF THE LIST. THE NUMBER OF A PAGE PAGED IN IS PLACED AT THE END OF THE LIST.
MR 4071

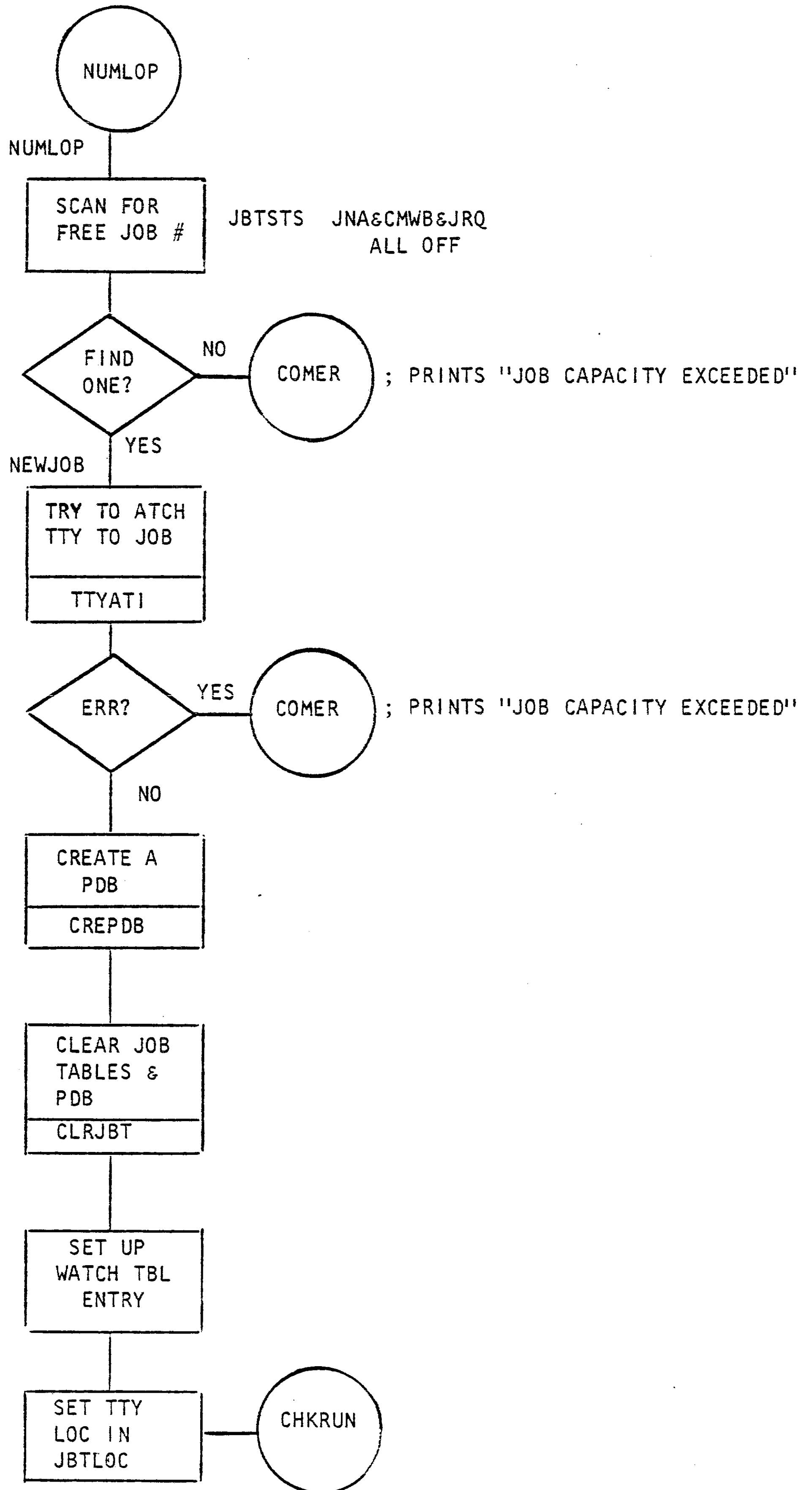
COMMAND PROCESSOR FLOW



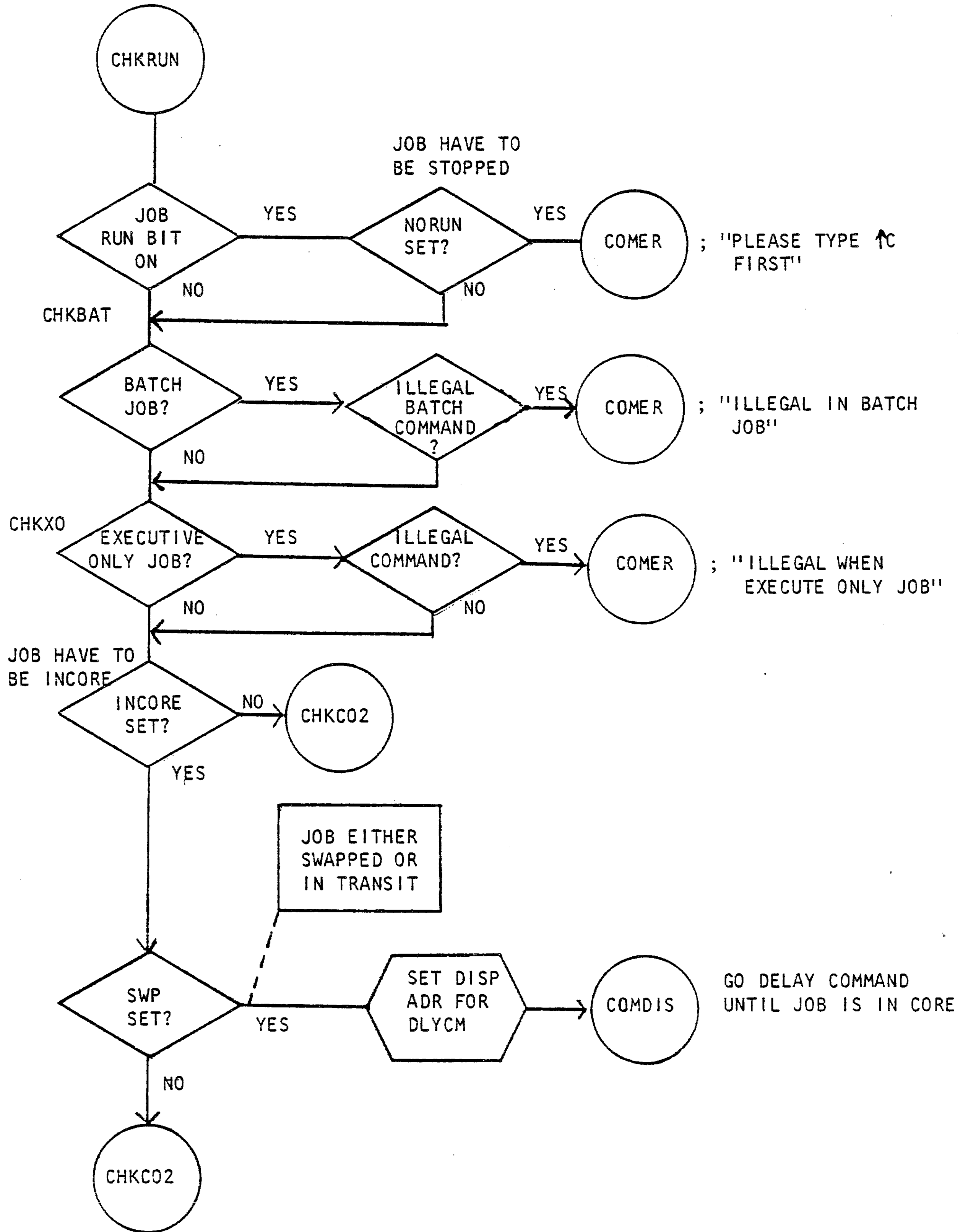
ENTER HERE FROM CLOCK1 BECAUSE COMCNT≠0

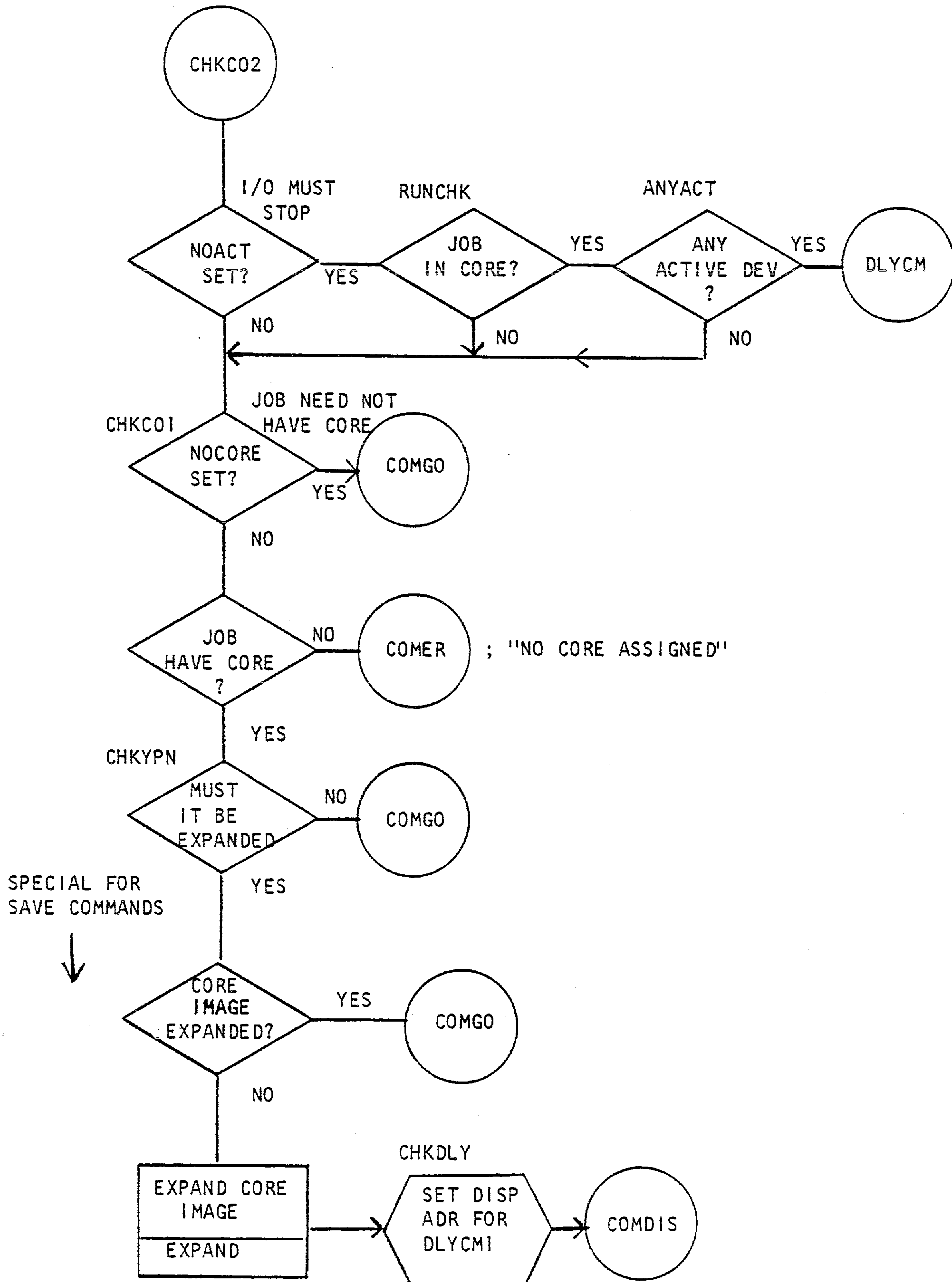


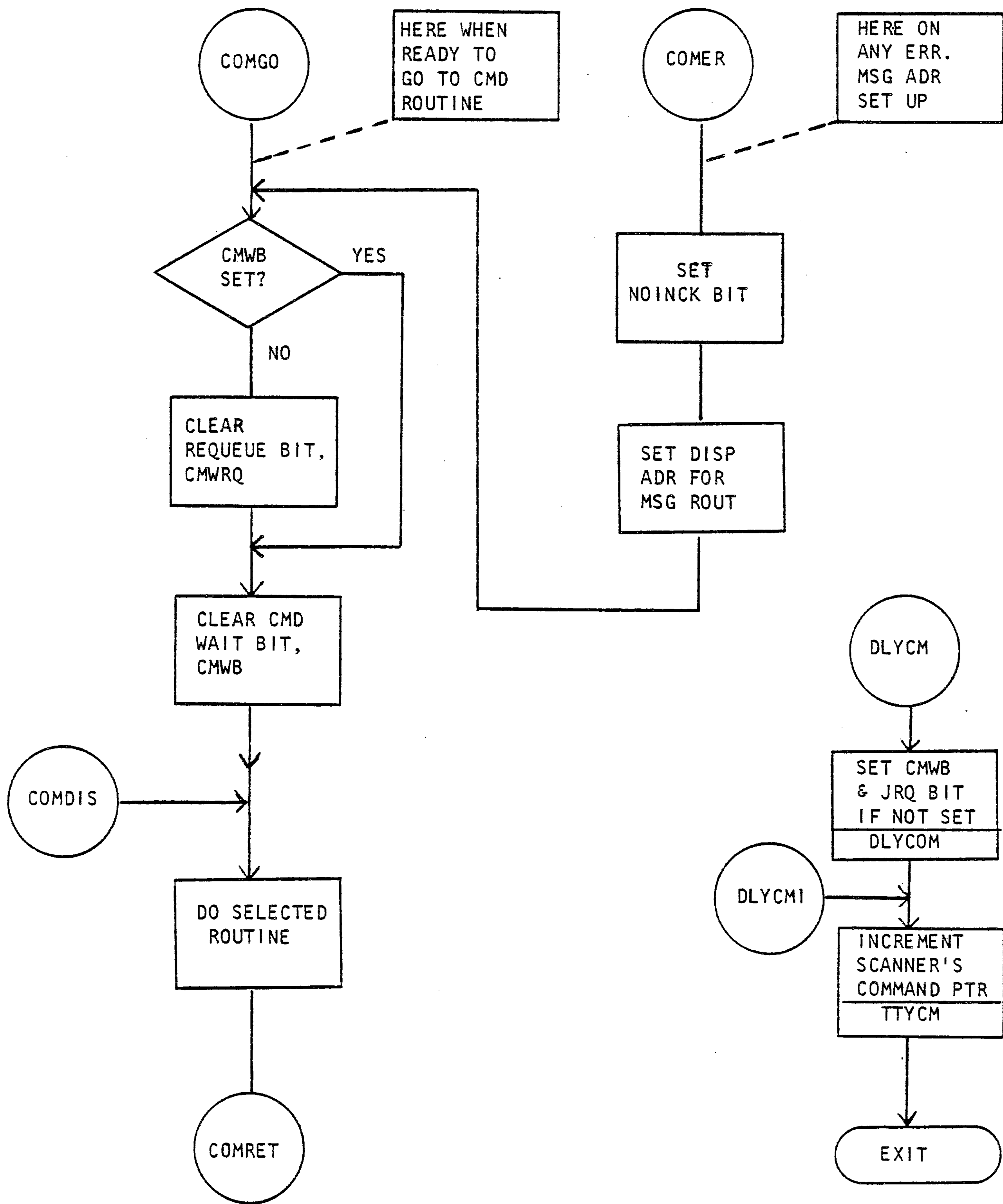
HERE TO FIND A FREE JOB NUMBER



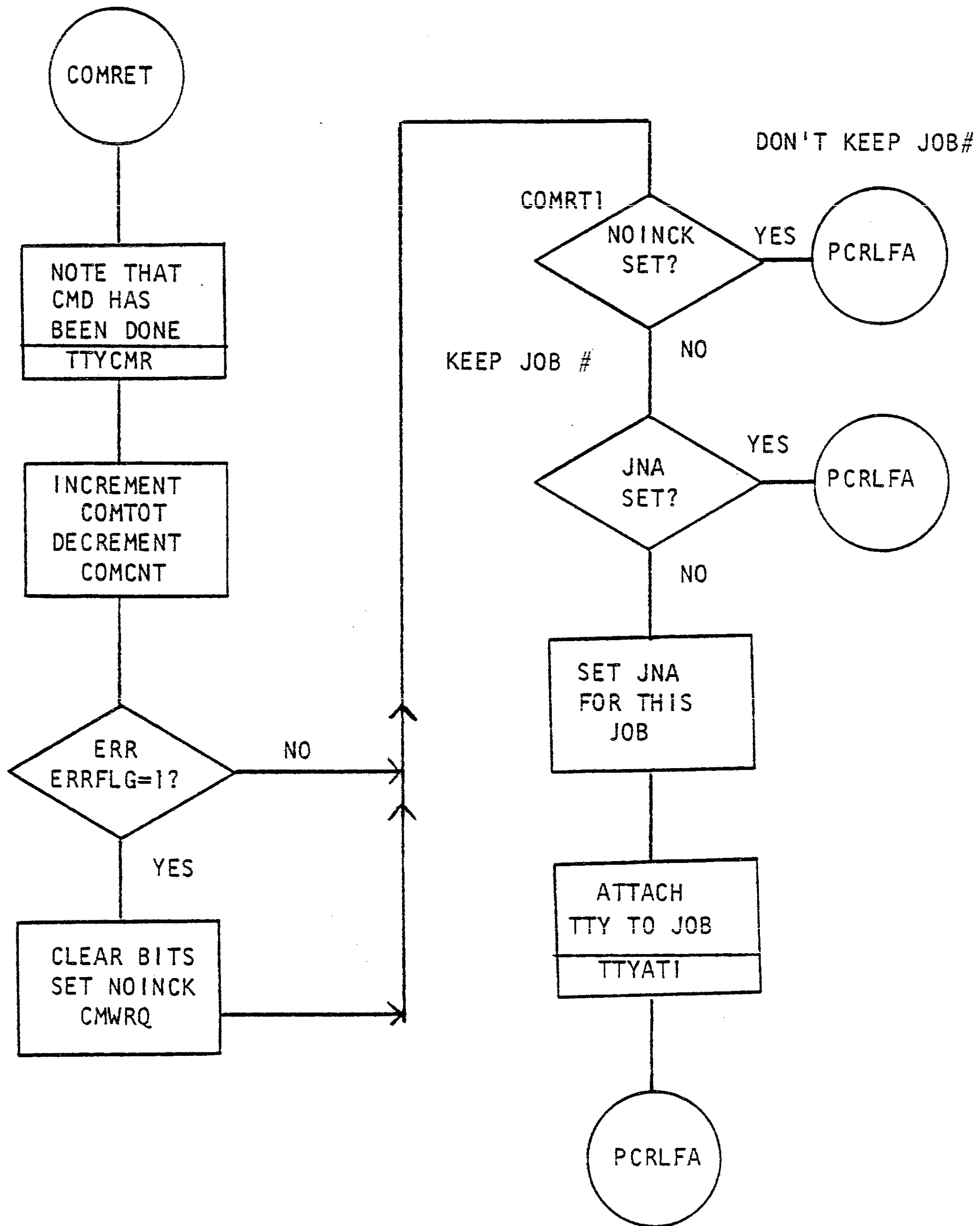
HERE WITH A JOB NUMBER

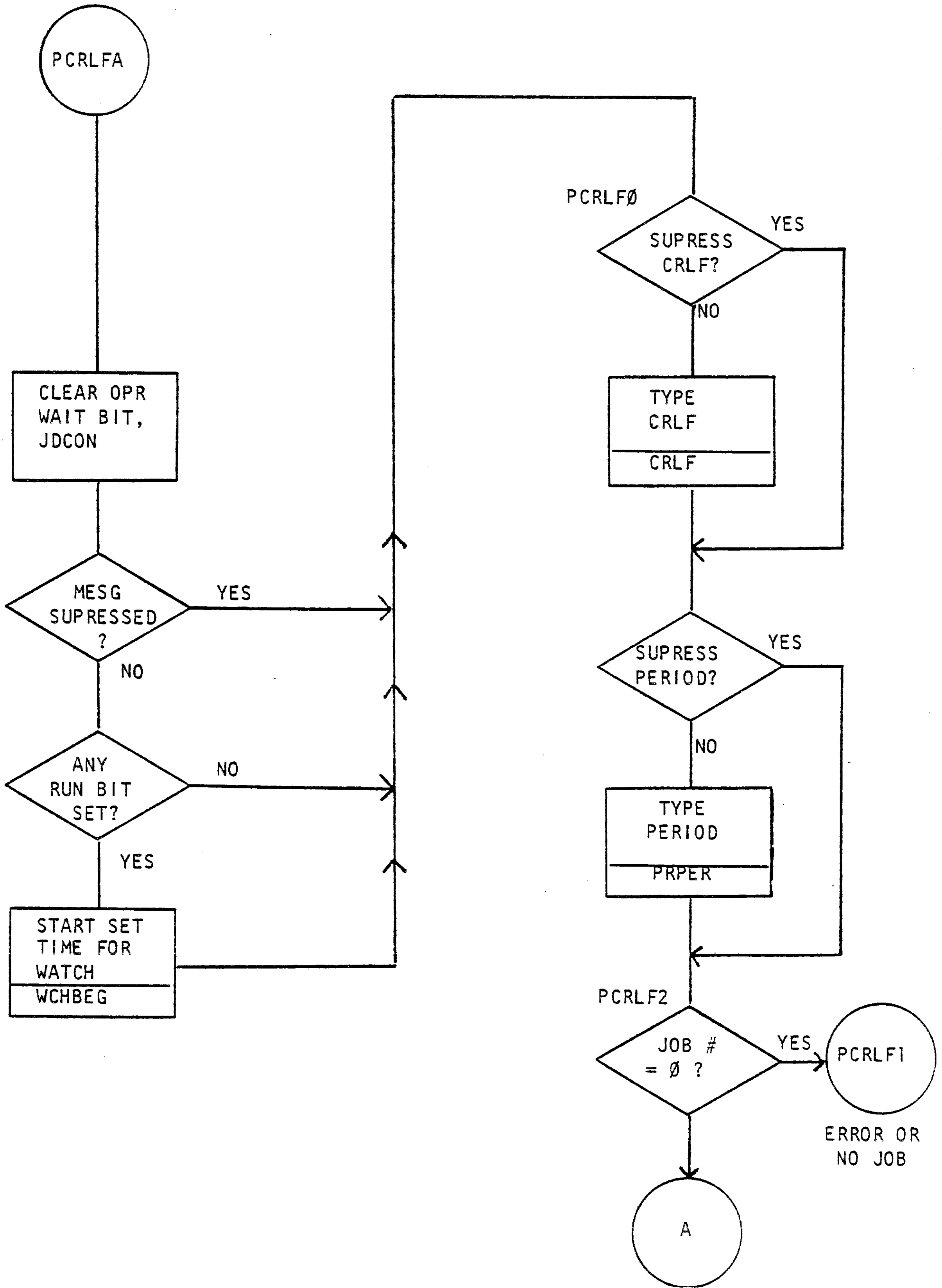


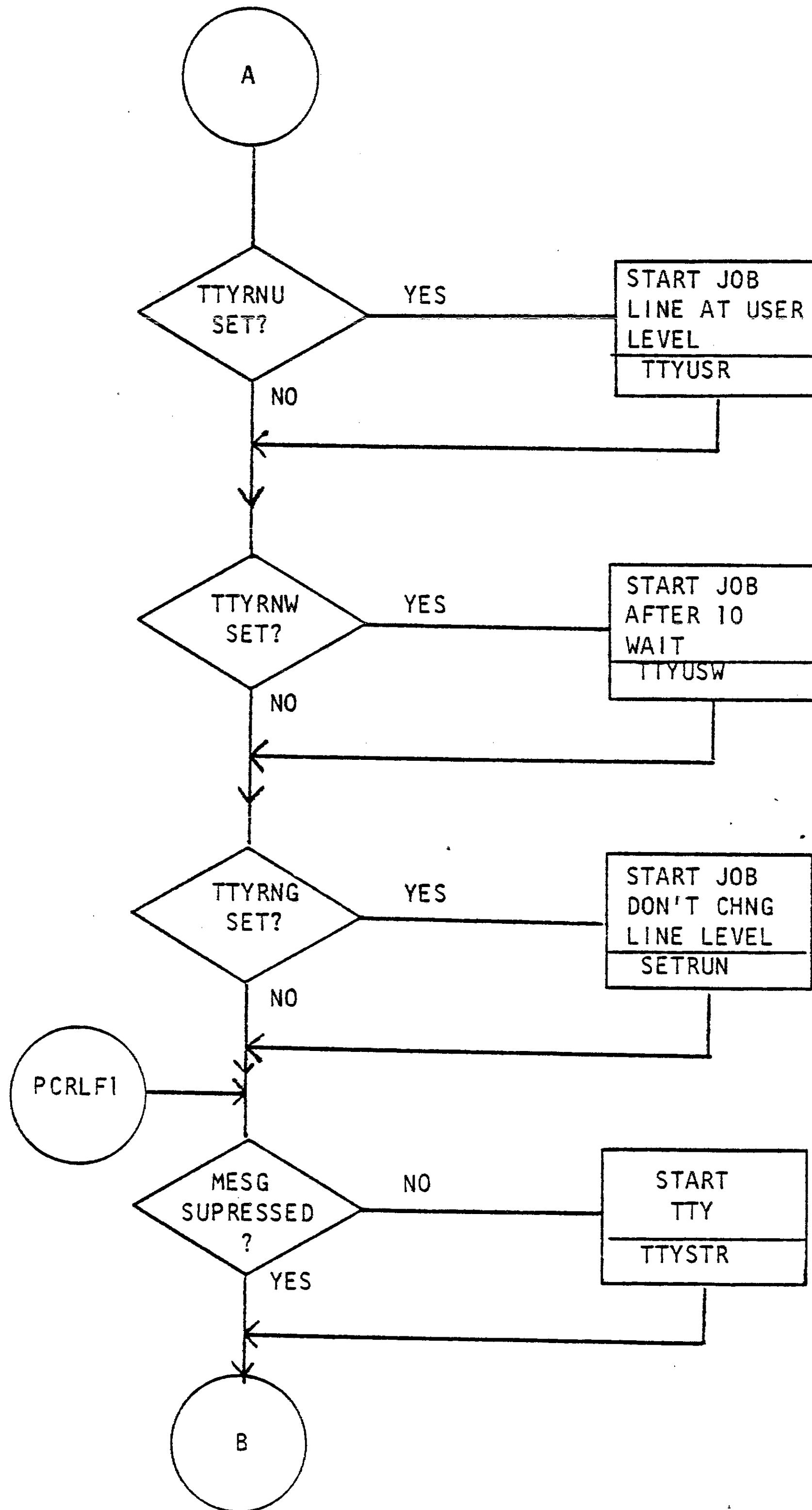

CHKCO2
I/O MUST
STOP
NOACT
SET?
YES
RUNCHK
JOB
IN CORE?
YES
ANYACT
ANY
ACTIVE DEV
?
YES
DLYCM
NO
NO
NO
CHKCO1
JOB NEED NOT
HAVE CORE
NOCORE
SET?
YES
COMGO
NO
JOB
HAVE CORE
?
NO
COMER
; "NO CORE ASSIGNED"
YES
CHKYPN
MUST
IT BE
EXPANDED
NO
COMGO
YES
SPECIAL FOR
SAVE COMMANDS
CORE
IMAGE
EXPANDED?
YES
COMGO
NO
CHKDLY
EXPAND CORE
IMAGE
EXPAND
SET DISP
ADR FOR
DLYCM1
COMDIS


COMGO
HERE WHEN READY TO GO TO CMD ROUTINE
COMER
HERE ON ANY ERR. MSG ADR SET UP
CMWB SET?
YES
NO
SET NOINCK BIT
CLEAR REQUEUE BIT, CMWRQ
SET DISP ADR FOR MSG ROUT
CLEAR CMD WAIT BIT, CMWB
DLYCM
COMDIS
SET CMWB & JRQ BIT IF NOT SET
DLYCOM
DLYCM1
DO SELECTED ROUTINE
INCREMENT SCANNER'S COMMAND PTR
TTYCM
COMRET
EXIT

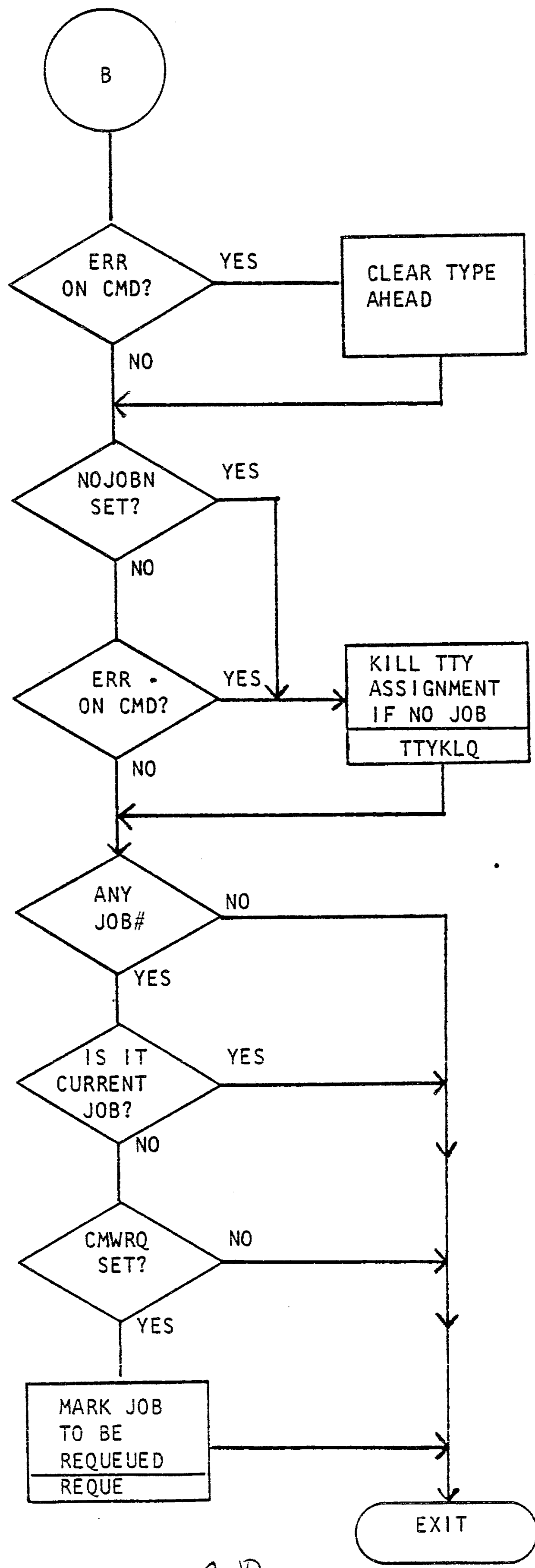
HERE AFTER THE COMMAND HAS BEEN "DONE"

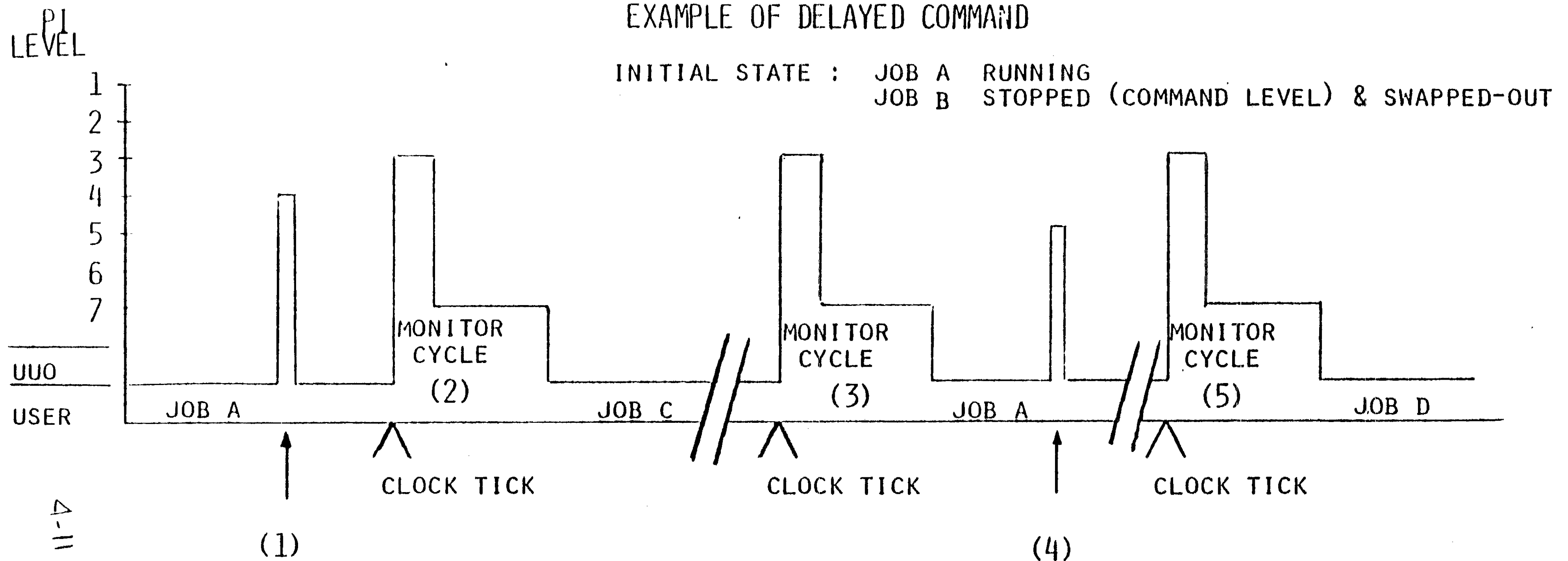



PCRLFA
CLEAR OPR
WAIT BIT,
JDCON
MESG
SUPRESSED
?
YES
NO
ANY
RUN BIT
SET?
NO
YES
START SET
TIME FOR
WATCH
WCHBEG
PCRLFØ
SUPRESS
CRLF?
YES
NO
TYPE
CRLF
CRLF
SUPRESS
PERIOD?
YES
NO
TYPE
PERIOD
PRPER
PCRLF2
JOB #
= Ø ?
YES
PCRLF1
ERROR OR
NO JOB
A



A
TTYRNU SET?
YES
START JOB LINE AT USER LEVEL
TTYUSR
NO
TTYRNW SET?
YES
START JOB AFTER 10 WAIT
TTYUSW
NO
TTYRNG SET?
YES
START JOB DON'T CHNG LINE LEVEL
SETRUN
NO
PCRLF1
MESG SUPRESSED ?
NO
START TTY
TTYSTR
YES
B


B
ERR
ON CMD?
YES
CLEAR TYPE
AHEAD
NO
NOJOBN
SET?
YES
NO
ERR •
ON CMD?
YES
KILL TTY
ASSIGNMENT
IF NO JOB
TTYKLQ
NO
ANY
JOB#
NO
YES
IS IT
CURRENT
JOB?
YES
NO
CMWRQ
SET?
NO
YES
MARK JOB
TO BE
REQUEUED
REQUE
EXIT



NOTES :

1. E COMMAND TYPED BY USER ASSOCIATED WITH JOB B.
   SCANNER SERVICE INTERRUPT CODE SETS CMDMAP BIT FOR THIS TTY LINE.

2. COMMAND PROCESSOR STARTS TO PROCESS THE E COMMAND , THIS COMMAND HAS THE "IN-CORE" BIT SET AND THE JOB IS SWAPPED-OUT, SO THE COMMAND MUST BE DELAYED UNTIL JOB B IS IN-CORE. COMMAND PROCESSOR SETS JOB'S JRQ BIT AND CMWB BIT (LEAVING THIS TTY LINE'S CMDMAP BIT SET).

   REQUEUING ROUTINE OF SCHEDULER PUTS JOB B INTO COMMAND WAIT QUEUE. (HIGH PRIORITY FOR SWAP-IN)

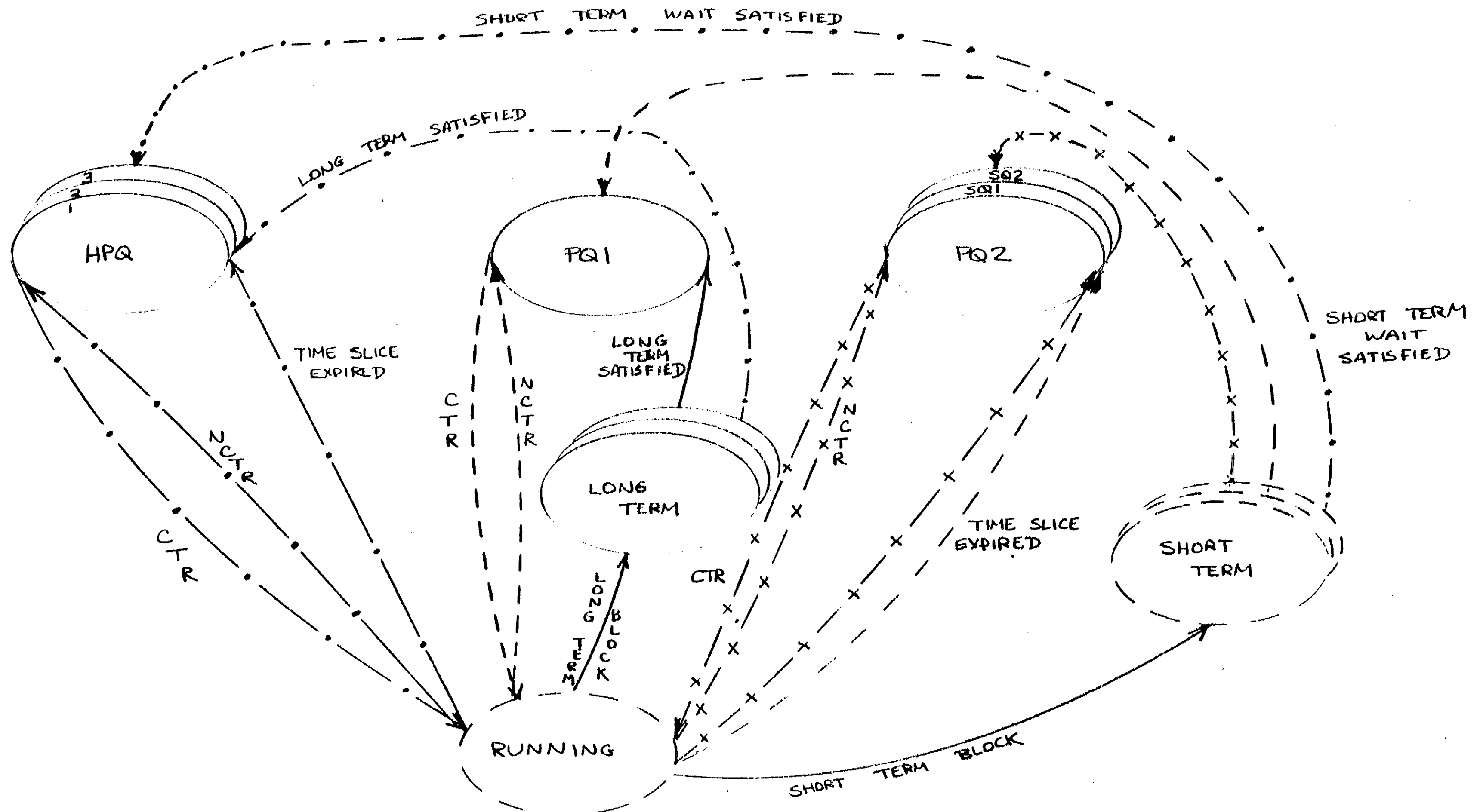
   SWAPPER PICKS JOB B FOR SWAP-IN.

3. THE DELAYED COMMAND WILL BE PROCESSED AGAIN HERE IF NO OTHER COMMANDS ARE PENDING; IF SO, THE COMMAND WILL BE DELAYED AGAIN BECAUSE THE JOB IS STILL SWAPPED-OUT.

4. SWAPPER I/O COMPLETE INTERRUPT, SWAPPER CLEARS JOB B'S SWAP BIT INDICATING JOB B NOW IN-CORE.

5. COMMAND PROCESSOR EVENTUALLY PICKS THE DELAYED COMMAND'S TTY LINE AGAIN AND SEES ORIGINAL COMMAND AGAIN (CMDMAP BIT STILL SET). THE COMMAND CAN NOW BE PROCESSED SINCE JOB IS NOW IN-CORE. AFTER PROCESSING THE COMMAND JOB MUST BE REQUEUED BACK TO ORIGINAL QUEUE (STOP QUEUE) THIS IS INDICATED BY THE CMWRQ BIT FOR THIS COMMAND.

TOPS-10 STATE TRANSITIONS

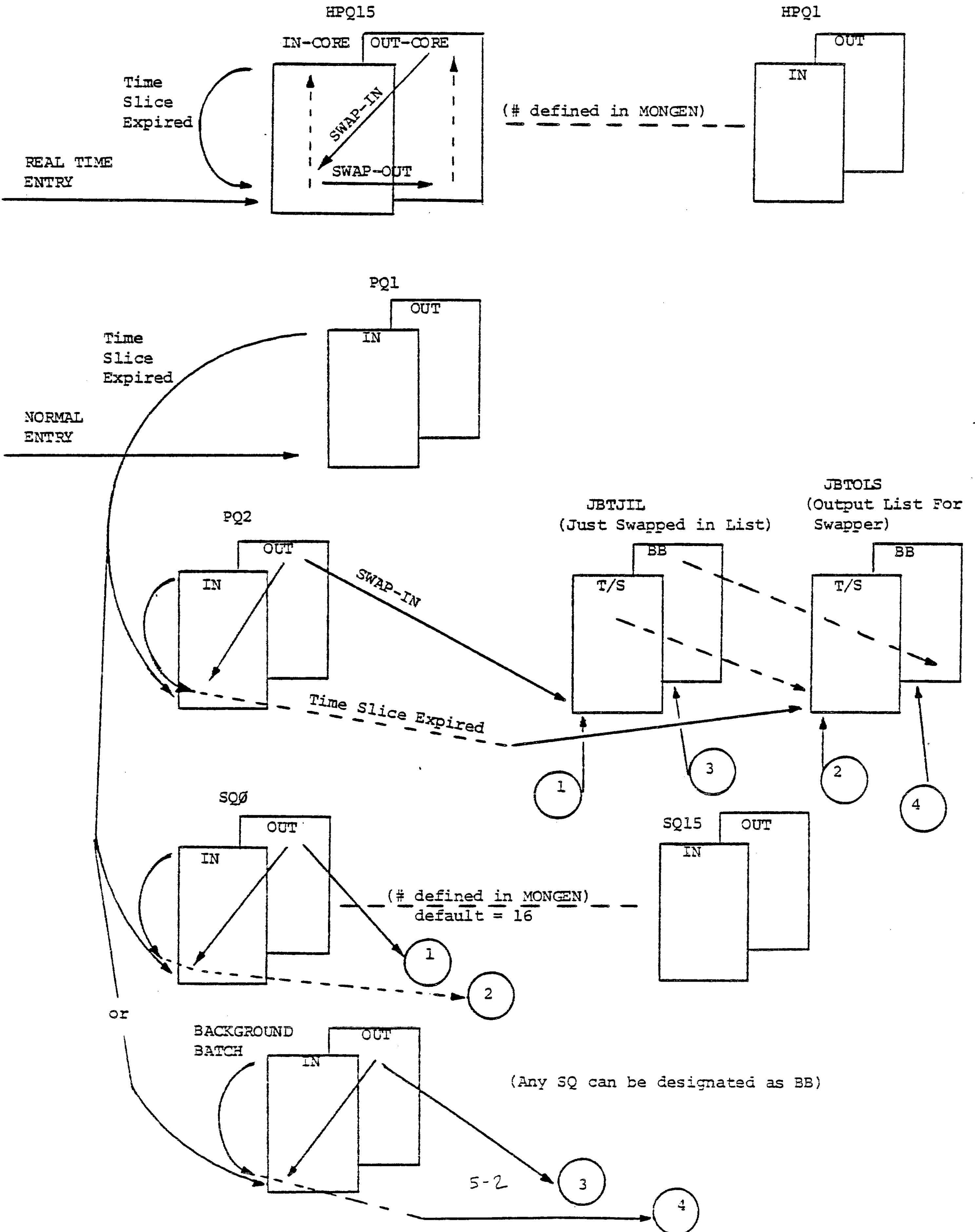


EVENTS

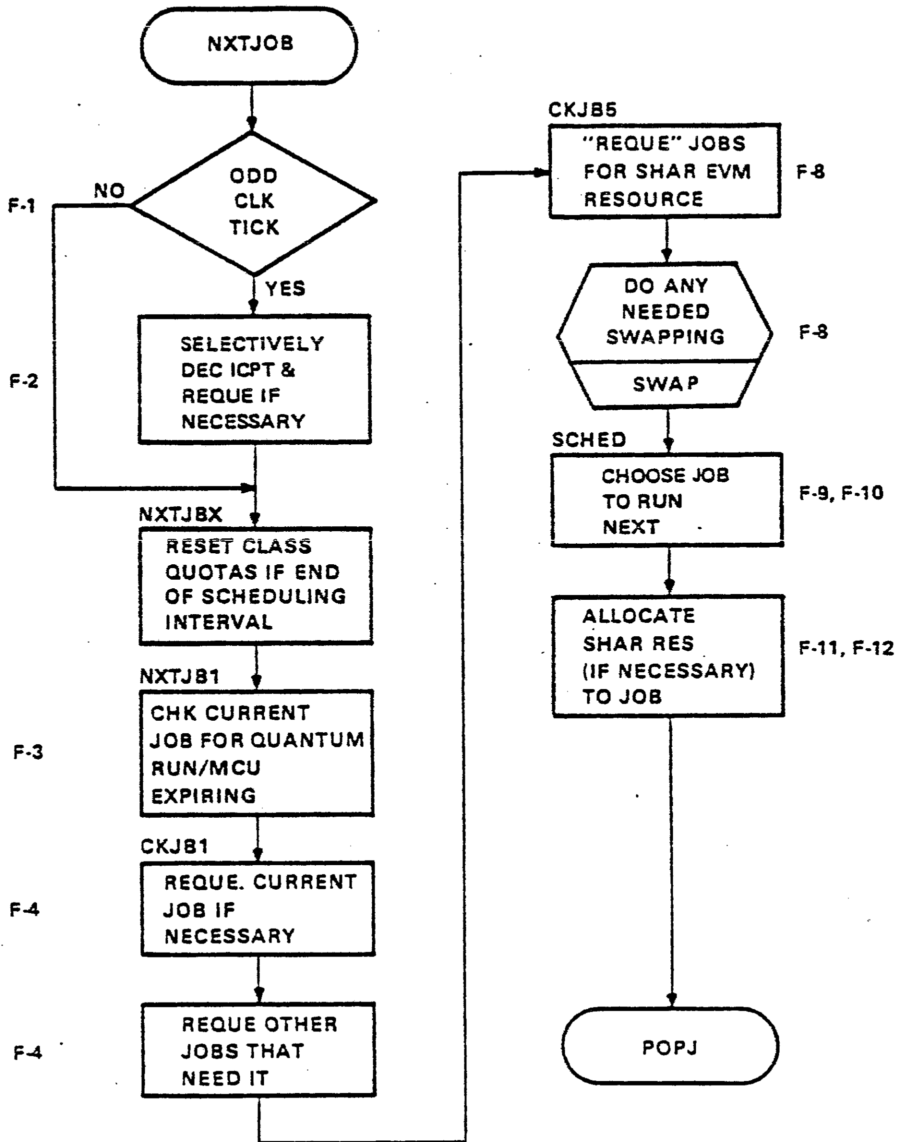
CTR - CHOSEN TO RUN
NCTR - NOT CHOSEN TO RUN

KEY

— • — • HPQ JOB STATE TRANSITIONS
— — — PQ1 JOB STATE TRANSITIONS
— X — X PQ2 JOB STATE TRANSITIONS
———— ANY PREVIOUS STATE

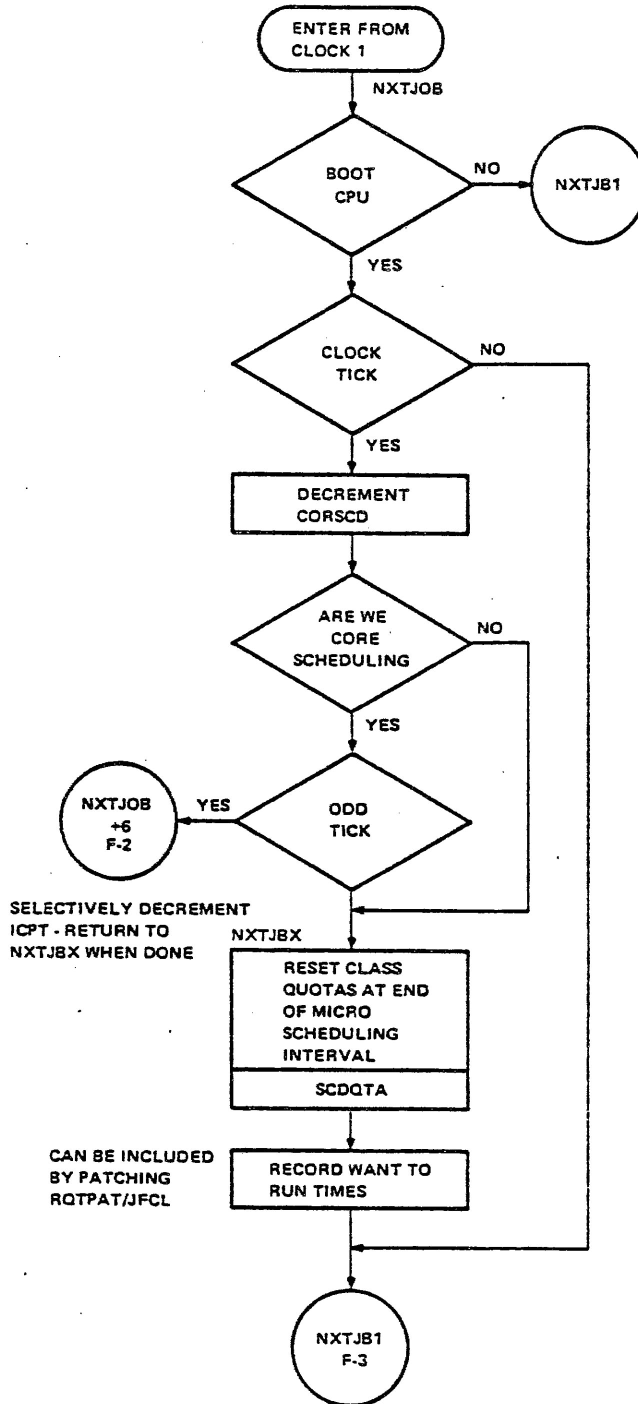

SCHEDULER QUEUES
HPQ15
IN-CORE
OUT-CORE
Time
Slice
Expired
SWAP-IN
SWAP-OUT
REAL TIME
ENTRY
(# defined in MONGEN)
HPQ1
OUT
IN
PQ1
OUT
IN
Time
Slice
Expired
NORMAL
ENTRY
PQ2
OUT
IN
SWAP-IN
JBTJIL
(Just Swapped in List)
BB
T/S
JBTOLS
(Output List For
Swapper)
BB
T/S
Time Slice Expired
1
3
2
4
SQØ
OUT
IN
SQ15
OUT
IN
(# defined in MONGEN)
default = 16
1
2
or
BACKGROUND
BATCH
OUT
IN
(Any SQ can be designated as BB)
3
4

7.01 SCHEDULER FLOWCHART

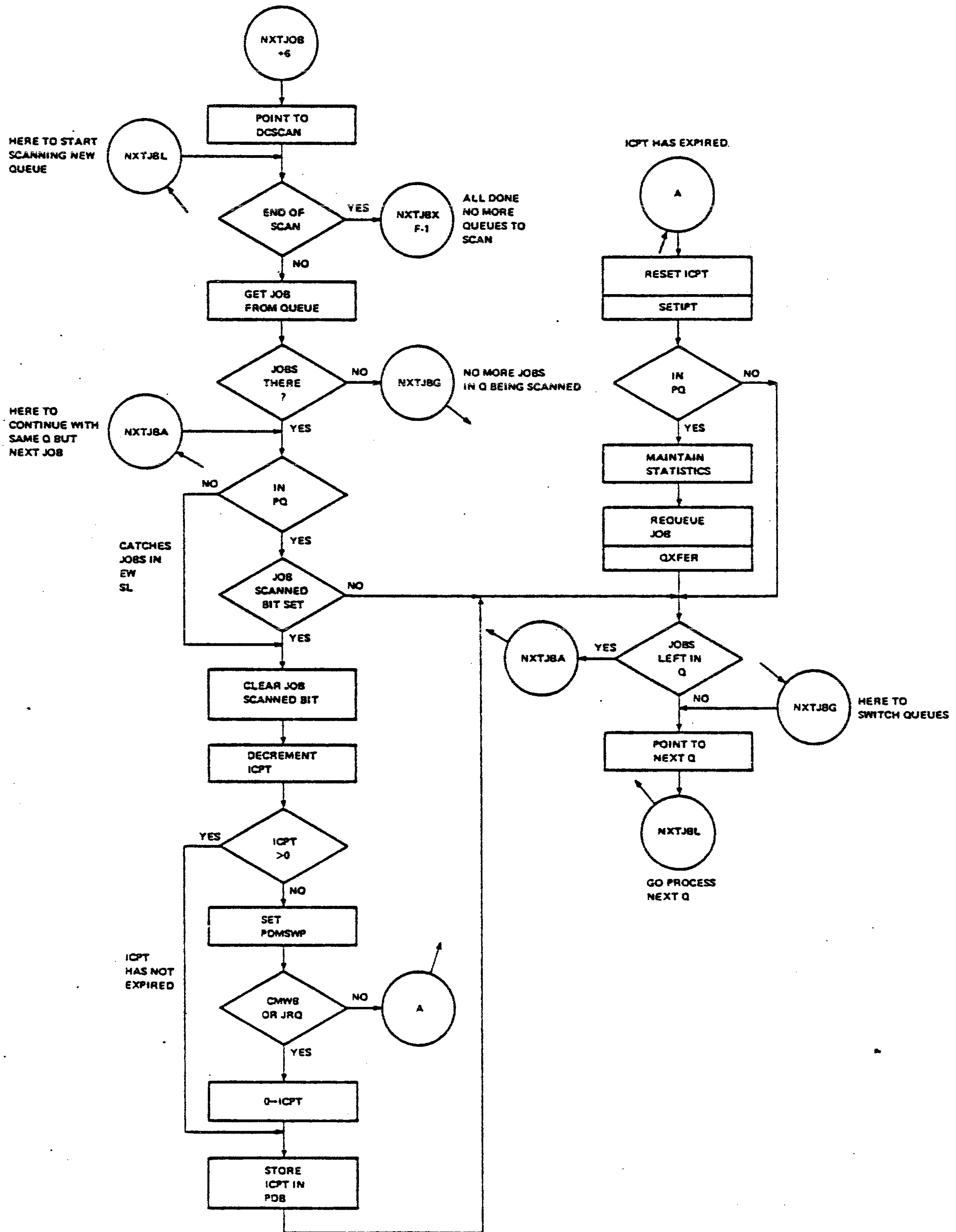


7.01 SCHEDULER

THE SCHEDULER IS CALLED FROM CLOCK1 AT CIP6 + 1

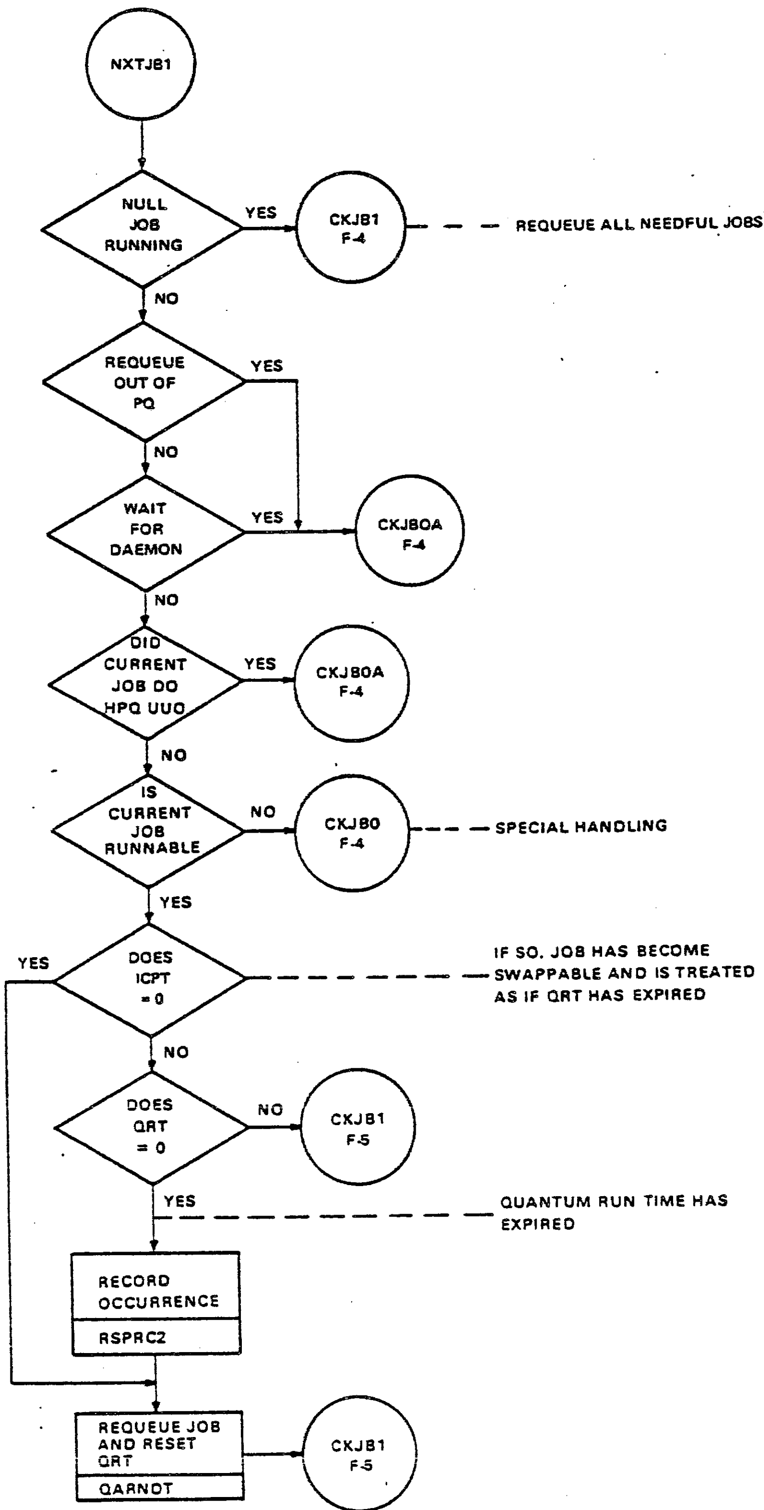


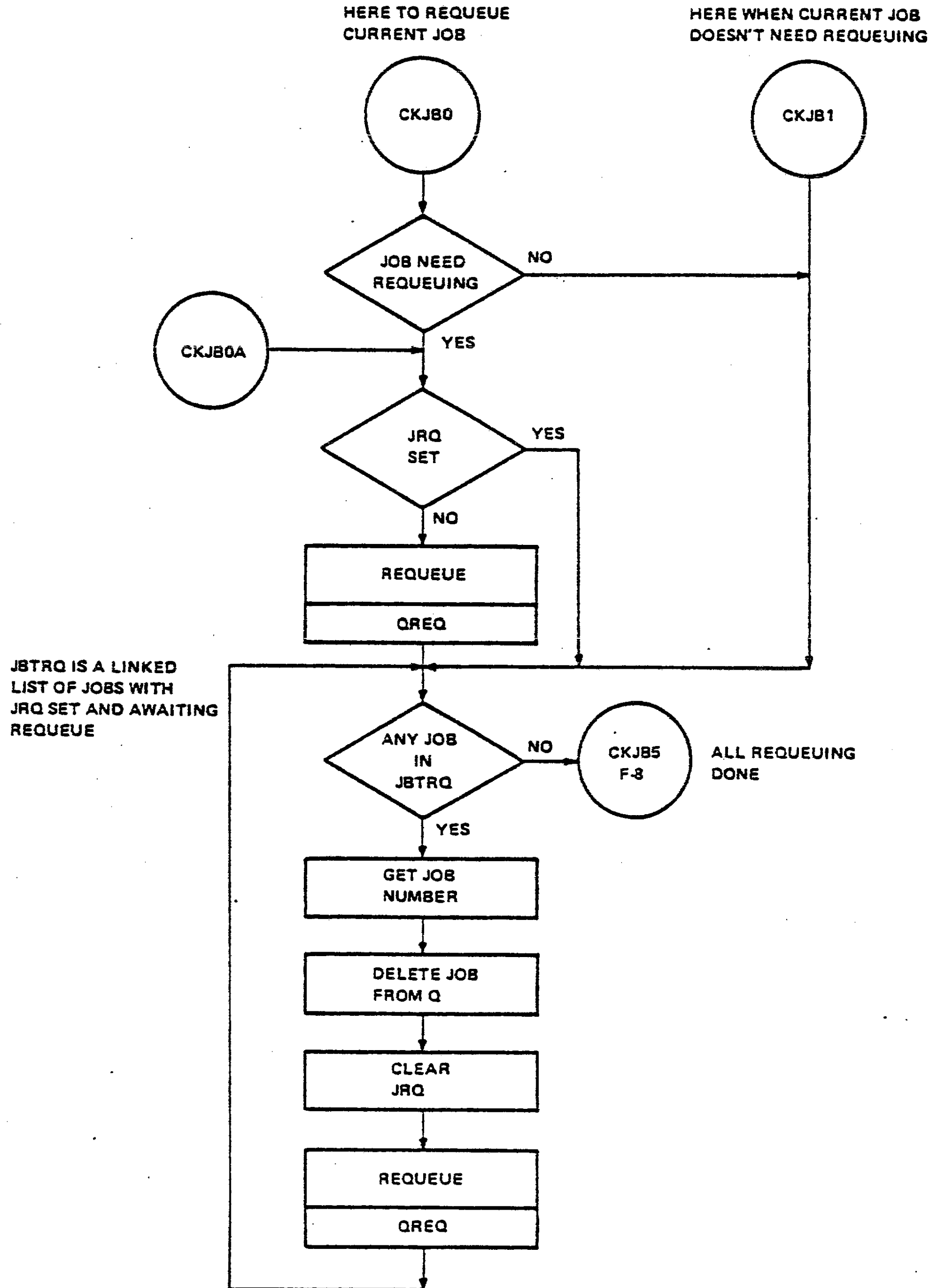
F-2
ICPT MAINTENANCE



F-3

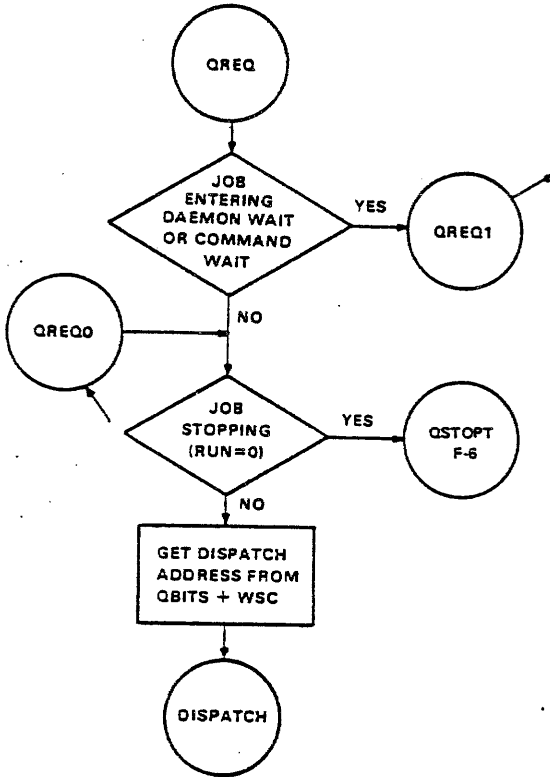
THIS PAGES REQUEUES CURRENT JOB IF ICPT OR QRT EXPIRED AND THEN GOES TO REQUEUE ALL JOBS



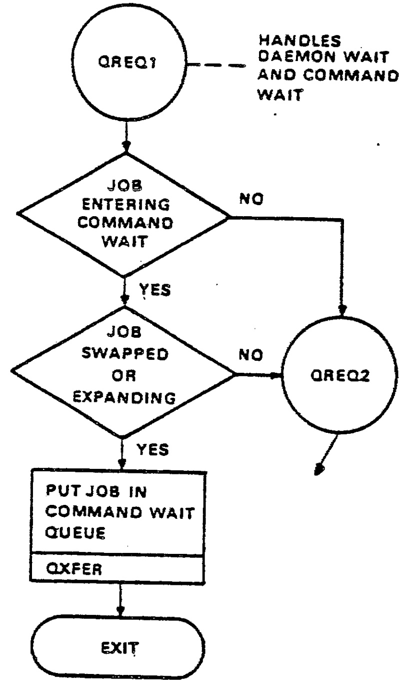

F-4
HERE TO REQUEUE CURRENT JOB IF NECESSARY
AS WELL AS ALL JOBS IN JBTRQ
HERE TO REQUEUE
CURRENT JOB
HERE WHEN CURRENT JOB
DOESN'T NEED REQUEUING
CKJB0
CKJB1
JOB NEED
REQUEUING
NO
YES
CKJB0A
JRQ
SET
YES
NO
REQUEUE
QREQ
JBTRQ IS A LINKED
LIST OF JOBS WITH
JRQ SET AND AWAITING
REQUEUE
ANY JOB
IN
JBTRQ
NO
CKJB5
F-8
ALL REQUEUING
DONE
YES
GET JOB
NUMBER
DELETE JOB
FROM Q
CLEAR
JRQ
REQUEUE
QREQ
MR-5005

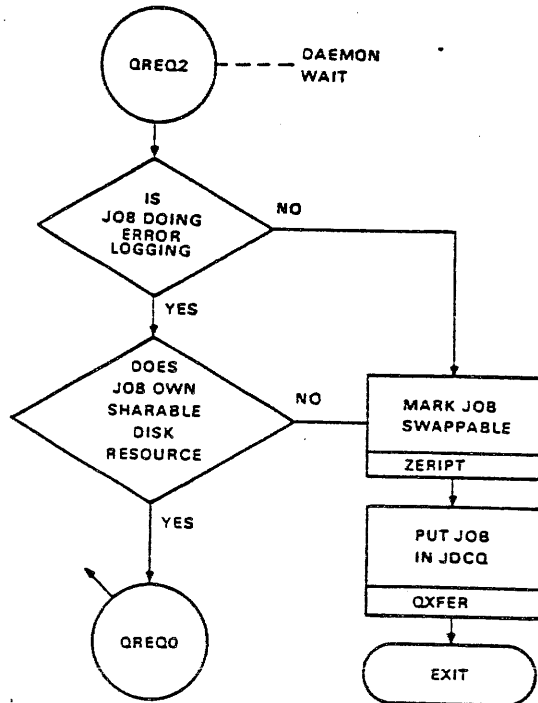
F-5

THIS ROUTINE REQUEUES A SINGLE JOB



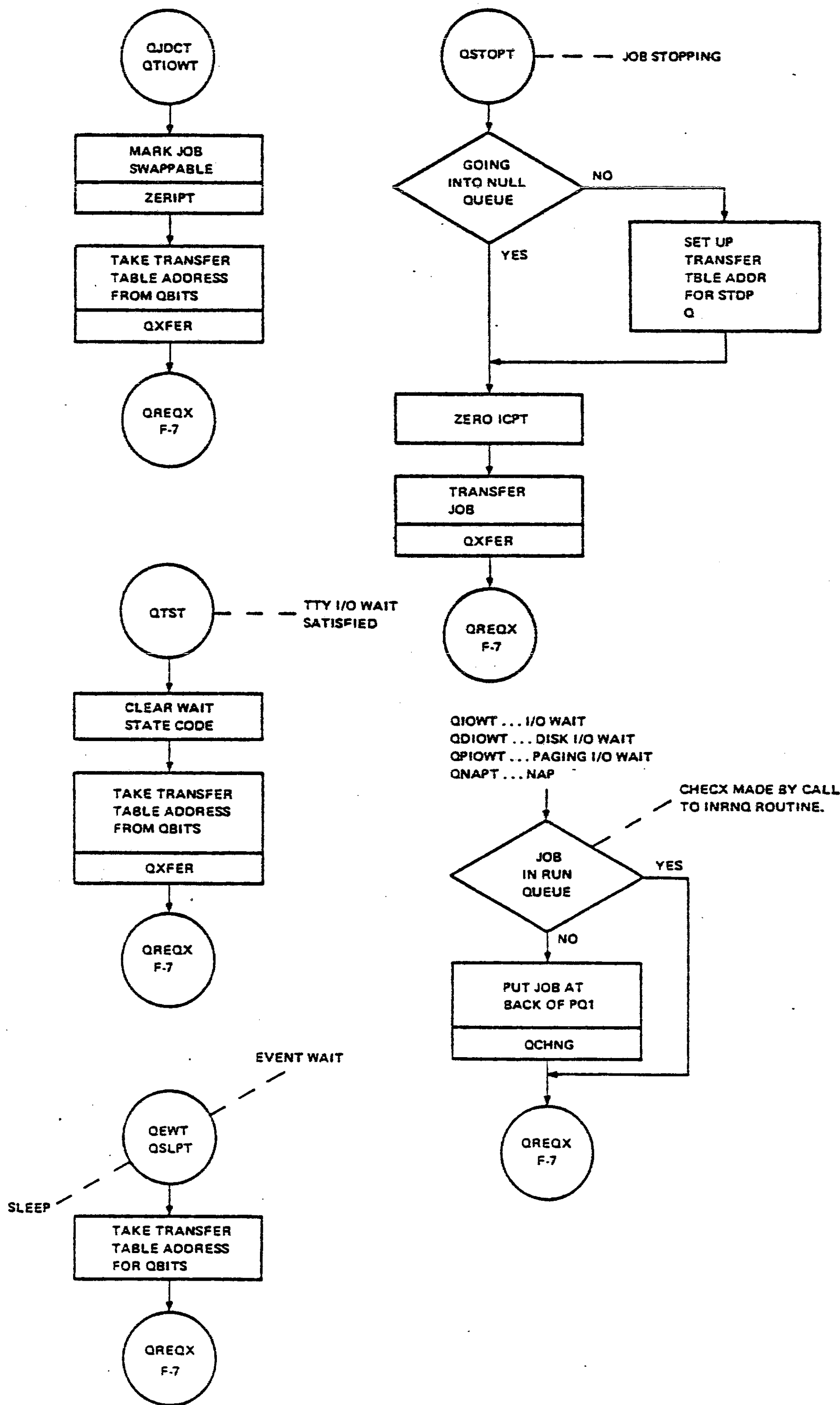
THESE DISPATCHES ARE SHOWN ON THIS AND THE FOLLOWING TWO PAGES

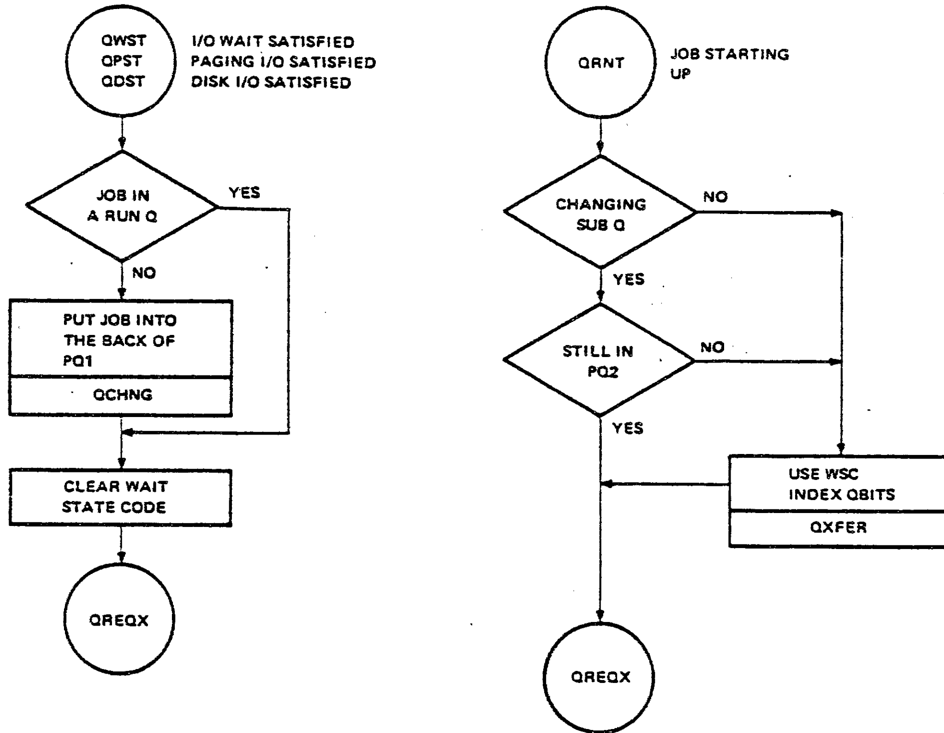




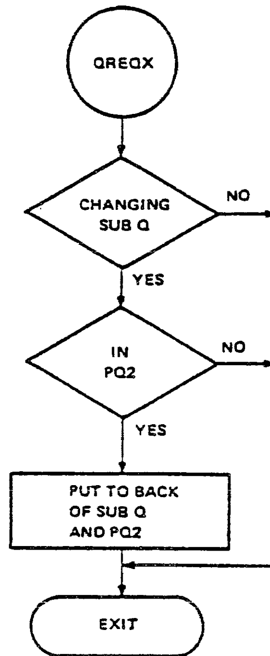
MR-5006

F-6
QREQ DISPATCHES

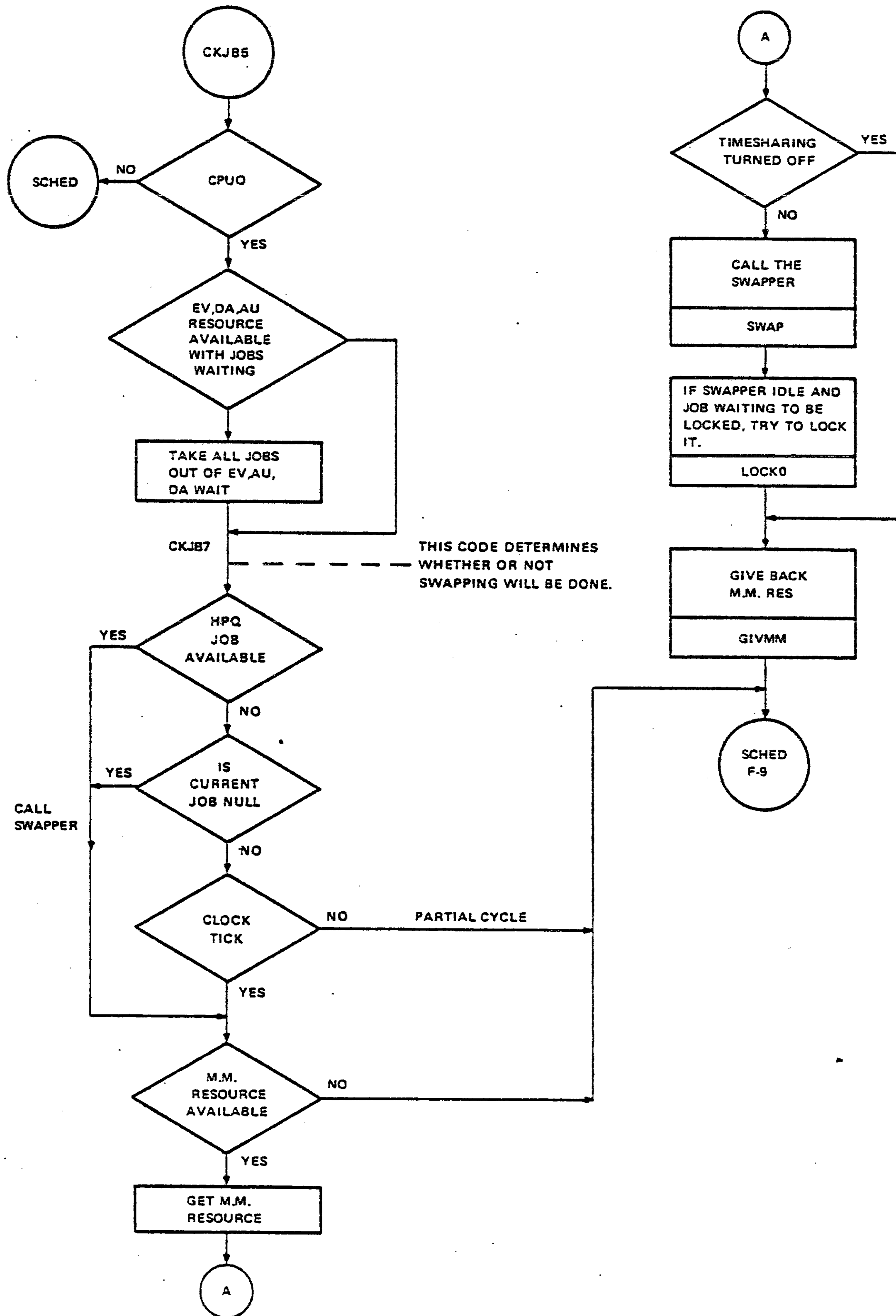


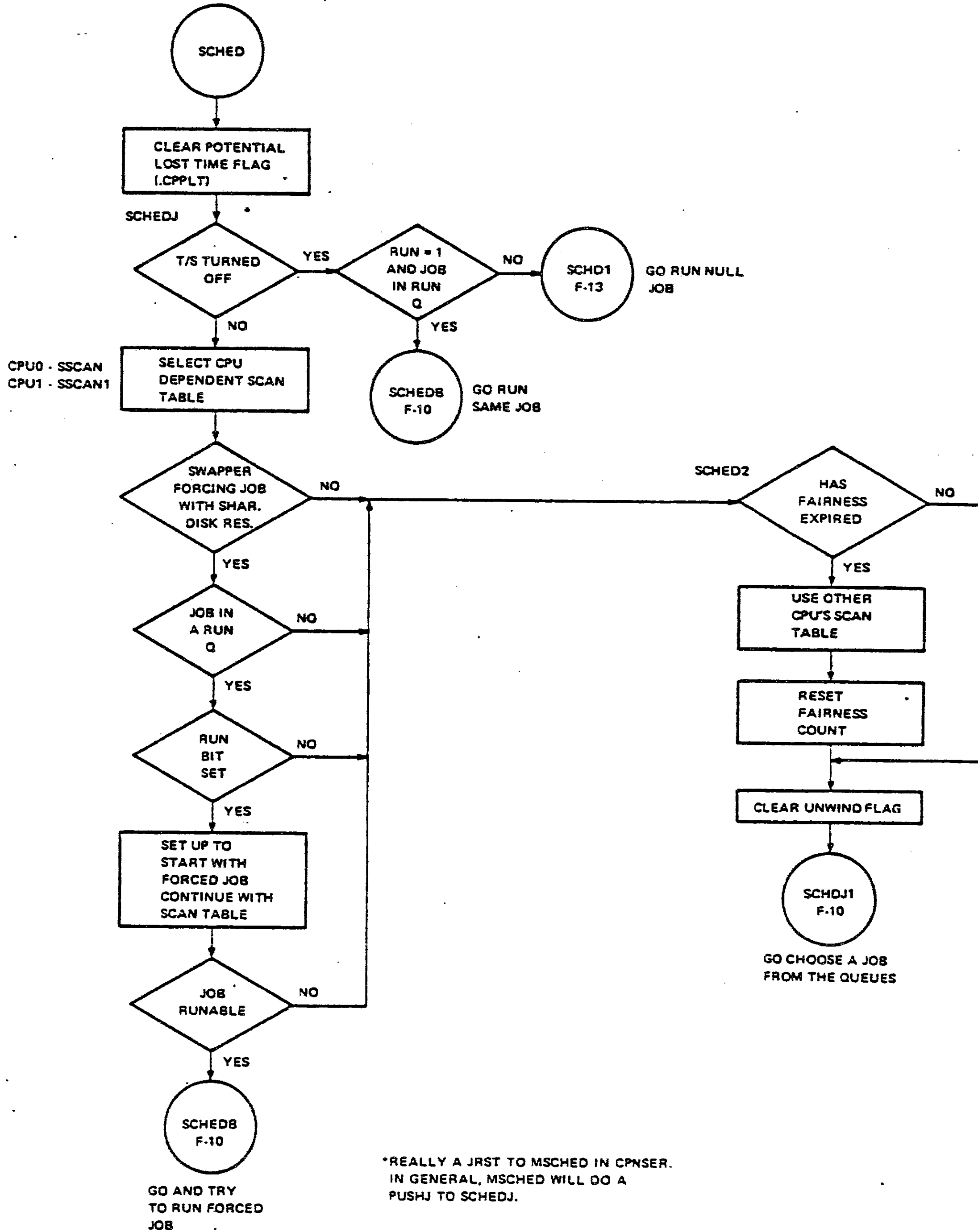


ALL REQUEUING EXITS THROUGH HERE



MR-5008


F-8
HERE AFTER JOB REQUEUING TO MANAGE EVM RESOURCE AND TO CALL SWAPPER IF APPROPRIATE
CKJB5
SCHED
NO
CPUO
YES
EV,DA,AU RESOURCE AVAILABLE WITH JOBS WAITING
TAKE ALL JOBS OUT OF EV,AU, DA WAIT
CKJB7
THIS CODE DETERMINES WHETHER OR NOT SWAPPING WILL BE DONE.
YES
HPQ JOB AVAILABLE
NO
YES
IS CURRENT JOB NULL
CALL SWAPPER
NO
CLOCK TICK
NO
PARTIAL CYCLE
YES
M.M. RESOURCE AVAILABLE
NO
YES
GET M.M. RESOURCE
A
A
TIMESHARING TURNED OFF
YES
NO
CALL THE SWAPPER
SWAP
IF SWAPPER IDLE AND JOB WAITING TO BE LOCKED, TRY TO LOCK IT.
LOCK0
GIVE BACK M.M. RES
GIVMM
SCHED F-9
MR-5009


F-9
HERE TO CHOOSE A SCAN TABLE
SCHED
CLEAR POTENTIAL LOST TIME FLAG (.CPPLT)
SCHEDJ
T/S TURNED OFF
YES
RUN = 1 AND JOB IN RUN Q
NO
SCHD1 F-13
GO RUN NULL JOB
YES
SCHEDB F-10
GO RUN SAME JOB
NO
CPU0 - SSCAN
CPU1 - SSCAN1
SELECT CPU DEPENDENT SCAN TABLE
SWAPPER FORCING JOB WITH SHAR. DISK RES.
NO
YES
JOB IN A RUN Q
NO
YES
RUN BIT SET
NO
YES
SET UP TO START WITH FORCED JOB CONTINUE WITH SCAN TABLE
JOB RUNABLE
NO
YES
SCHEDB F-10
GO AND TRY TO RUN FORCED JOB
SCHED2
HAS FAIRNESS EXPIRED
NO
YES
USE OTHER CPU'S SCAN TABLE
RESET FAIRNESS COUNT
CLEAR UNWIND FLAG
SCHDJ1 F-10
GO CHOOSE A JOB FROM THE QUEUES
*REALLY A JRST TO MSCHED IN CPNSER. IN GENERAL, MSCHED WILL DO A PUSHJ TO SCHEDJ.
MR-5010

F-10

SCAN THE QUEUES CHOOSING A JOB TO RUN

SCHDJ1 — FIND A RUNNABLE JOB IN THE QUEUES

SELECT A JOB FROM THE QUEUES
QSCAN

ANY MORE TO CHECK — NO → SCHD1 F-13 (GO RUN NULL JOB)

YES

SCHEDB — HERE WITH FORCEI JOB OR WHEN T/S TURNED .OFF AND JOB STILL RUNNABLE

JOB OKAY FOR THIS CPU — NO — CHECKED BY DXRUN

YES

SET JS.SCN FOR JOB — CAUSING ICPT TO BE DECREMENTED THE NEXT CYCLE

WAIT STATE CODE = 0 — NO → E F-11

YES

SCHEDC

SET .CPPLT

FORCED JOB WITH NO RESOURCE — CJFWRX ROUTINE — YES

NO

D

THIS PATH IS TAKEN WHENEVER A JOB IS REJECTED

D

JOB BEING LOCKED — YES

NO

REQUEUEING, SWAPPED, SHUFFLING OR EXPANDING — YES

NO

JOB RUNNABLE WITH RESPECT TO CACHE — SCDCSH ROUTINE — NO

YES — WE'LL USE THIS JOB

IF A FORCED JOB, INCREMENT FORCFC

JOB FROM SUBQUEUES — NO

YES

SET .CPSQF (TELL CLOCK1 SO IT CAN ADJUST CLASS QUOTAS)

MAINTAIN RESPONSE DATA, CLEAR .CPPLT AND .CPCLF

REACH FAIR TERRITORY — NO → INCREMENT FAIRNESS COUNT

YES — RESET FAIRNESS COUNT

EXIT — LEAVING JOB # IN J

MR-5011

F-11

HERE WHEN WSC ≠ FOR CANDIDATE JOB TO RUN – EITHER GIVE HIM RESOURCE OR GO UNWIND

E

HERE WHEN WSC ≠ 0

IF FORCE JOB WITH JXPN=1 LET IT RUN

CJFRCX

CLEAR JXPN TEMPORARILY

JOB IN EV,AU,DA WAIT, SWP,SHF,JXPN JRQ

YES

NO

JOB IN SHARABLE RESOURCE WAIT

NO

SCHDJ1 F-10

YES

IS RESOURCE AVAILABLE

NO

HAS SCHED. UNWOUND THIS CALL

YES

NO

SET UNWIND FLAG

UNWND1 F12

NOW WE UNWIND

YES

IS IT THE FORCE JOB

YES

STILL HAVE RESOURCE

NO

SCHDJ1 F-10

YES

SCHEDE

NO

GIVE RESOURCE TO JOB AND UNWIND

MAINTAIN SHARABLE RESOURCE DATA BASE

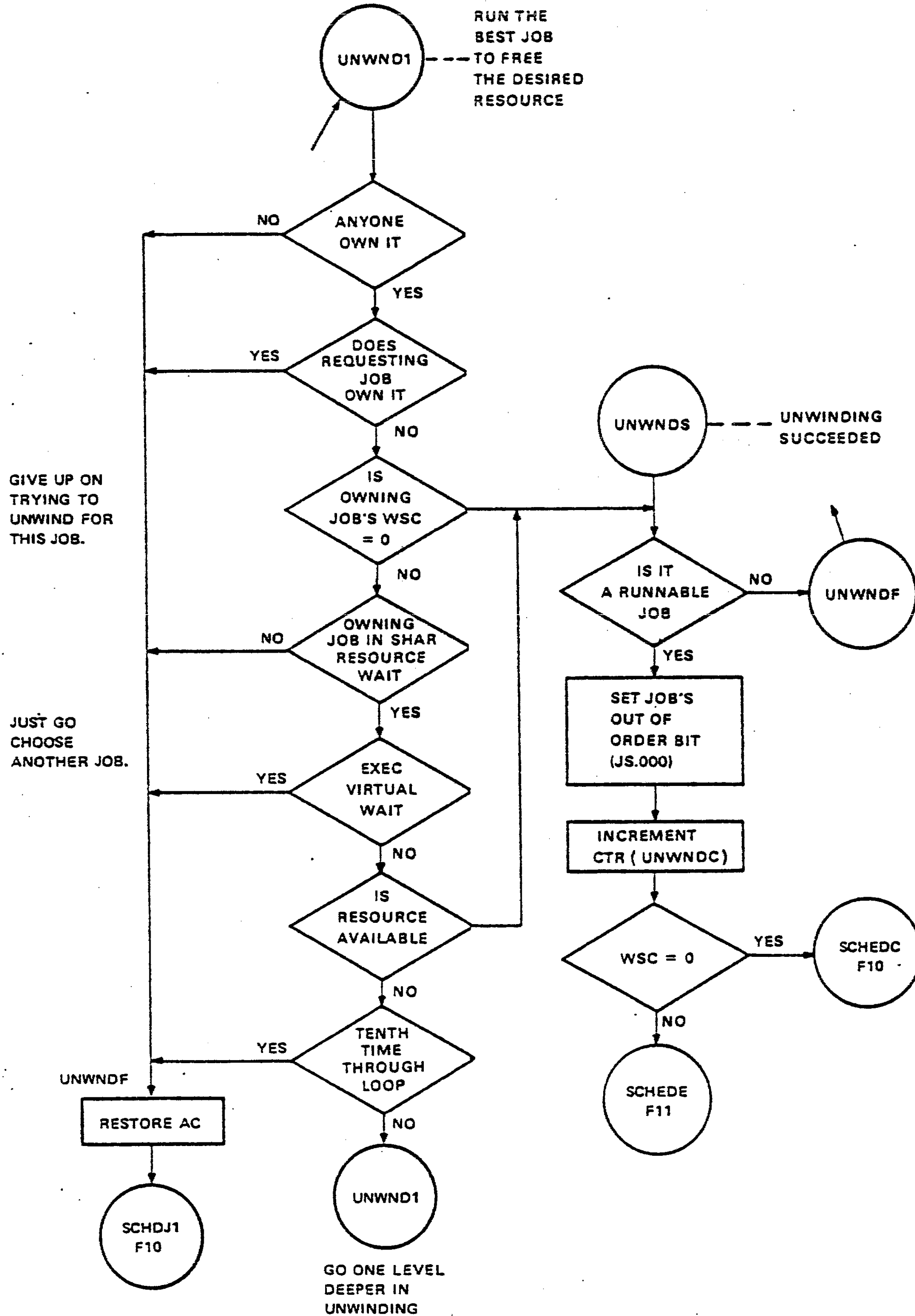
UPDATE AVTBMQ, USTBMQ

CLEAR JOB'S WAIT STATE CODE

SCHEDC F-10

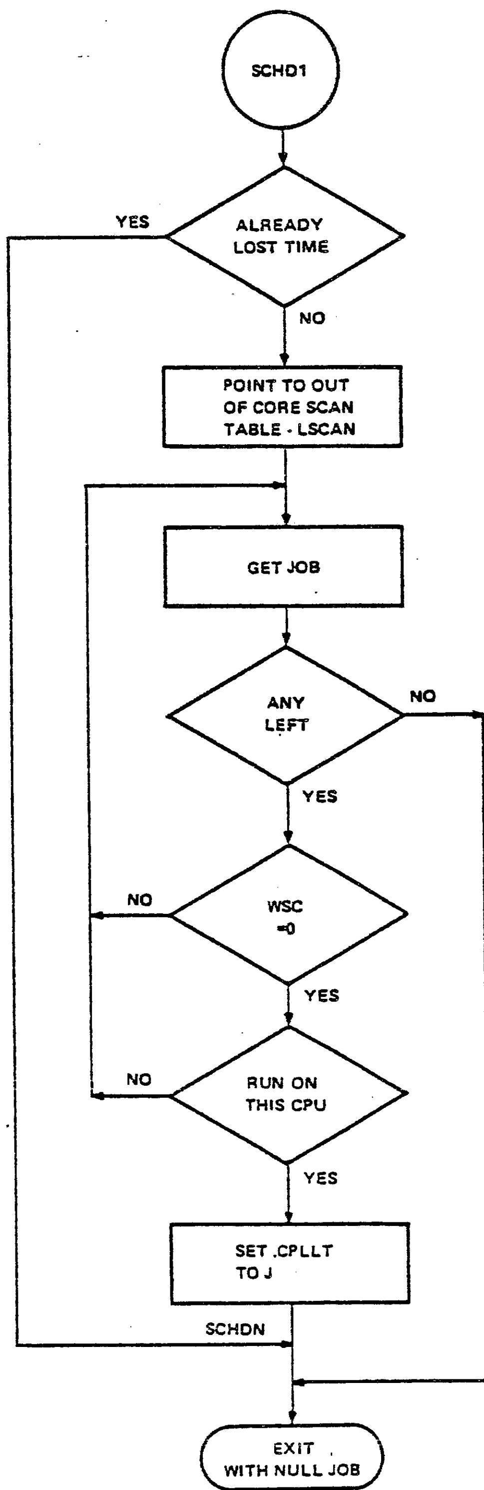
RUN THE JOB

MR-5012


F12
HERE TO UNWIND RESOURSE EITHER
UNWIND UP TO 10 LEVELS DEEP
OR GIVE UP JOB
UNWND1
RUN THE
BEST JOB
TO FREE
THE DESIRED
RESOURCE
ANYONE
OWN IT
NO
YES
DOES
REQUESTING
JOB
OWN IT
YES
NO
IS
OWNING
JOB'S WSC
= 0
NO
GIVE UP ON
TRYING TO
UNWIND FOR
THIS JOB.
UNWNDS
UNWINDING
SUCCEEDED
OWNING
JOB IN SHAR
RESOURCE
WAIT
NO
YES
IS IT
A RUNNABLE
JOB
NO
UNWNDF
YES
SET JOB'S
OUT OF
ORDER BIT
(JS.000)
JUST GO
CHOOSE
ANOTHER JOB.
EXEC
VIRTUAL
WAIT
YES
NO
INCREMENT
CTR (UNWNDC)
IS
RESOURCE
AVAILABLE
NO
WSC = 0
YES
SCHEDC
F10
NO
TENTH
TIME
THROUGH
LOOP
YES
UNWNDF
NO
SCHEDE
F11
RESTORE AC
SCHDJ1
F10
UNWND1
GO ONE LEVEL
DEEPER IN
UNWINDING
MR-5013

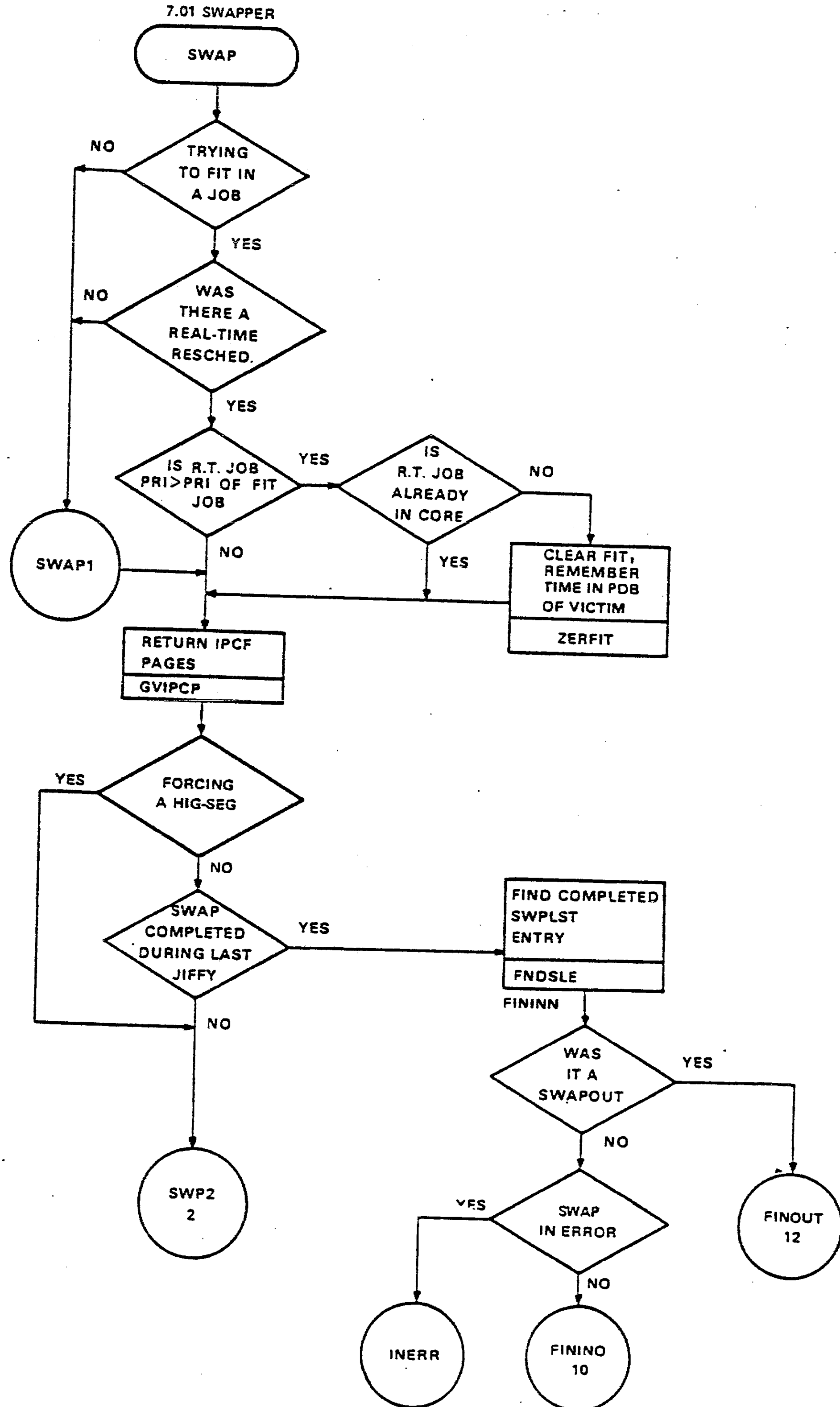
F-13

HERE WHEN NO JOB CAN BE FOUND TO RUN. DETERMINE IF LOST TIME FLAG SHOULD BE SET THEN RETURN JOB 0



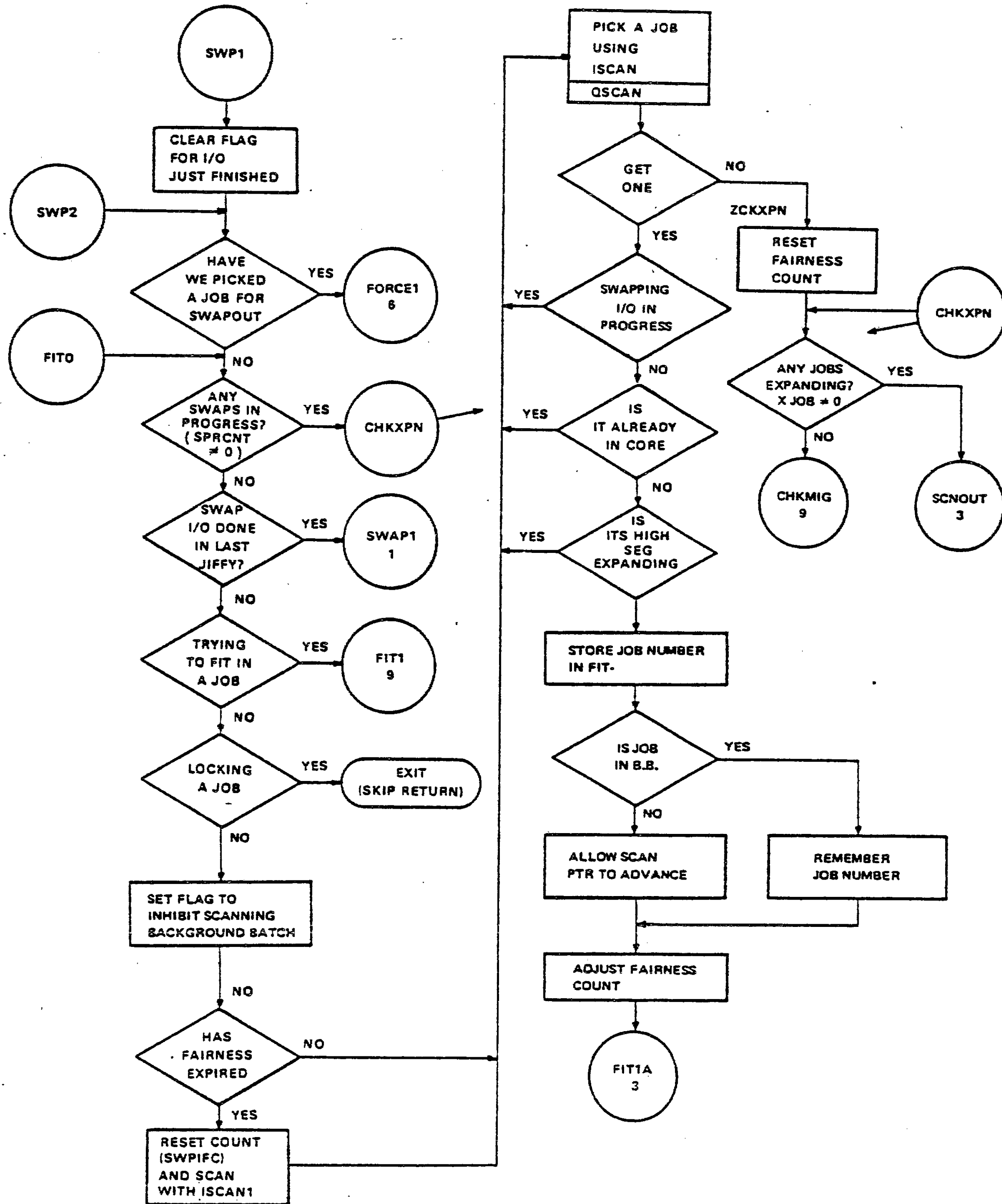
F-1

DETERMINE WHERE WE LEFT OFF ON THE LAST PASS AND WHAT TO DO NOW



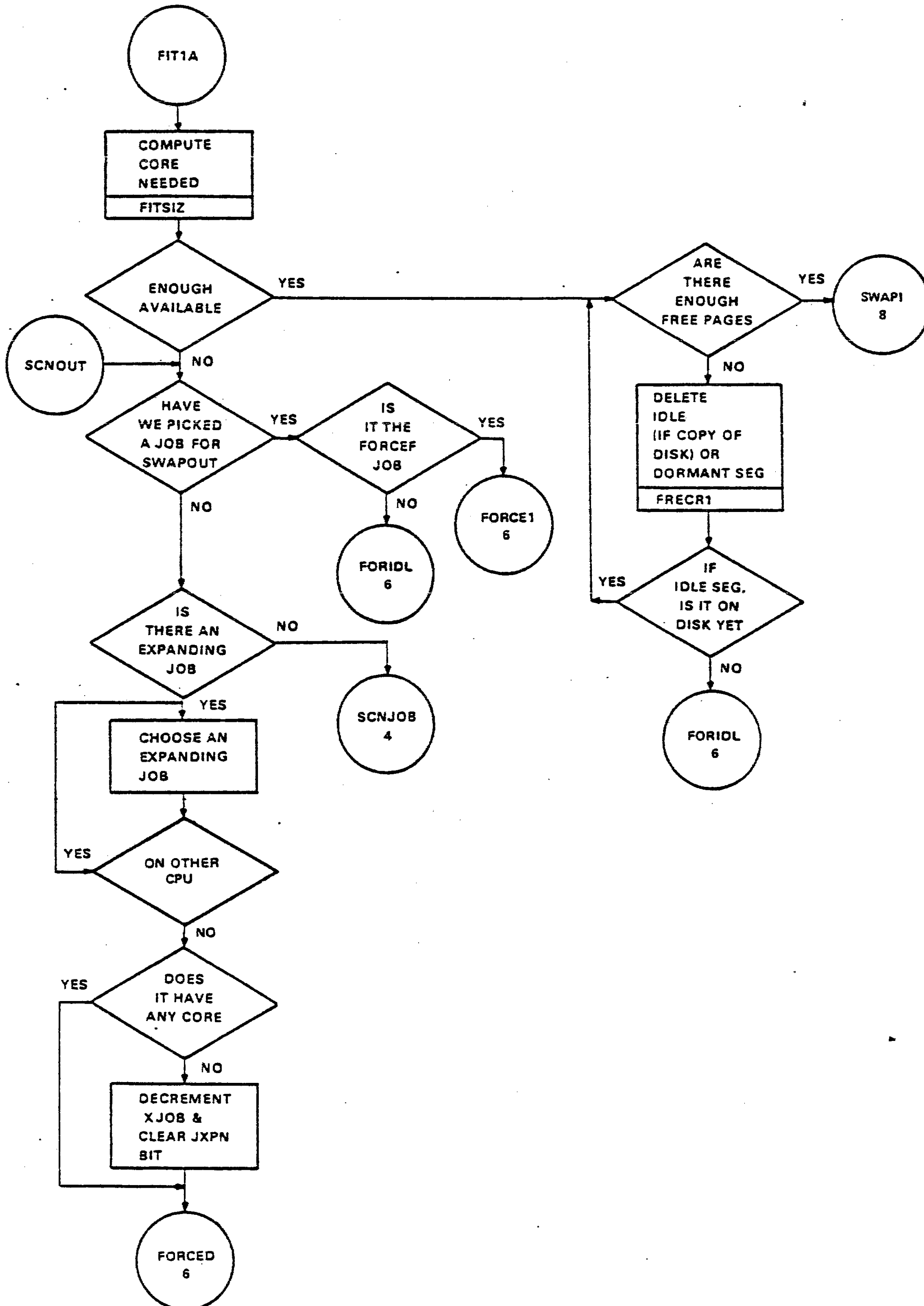
F-2

PICK A JOB TO SWAP IN



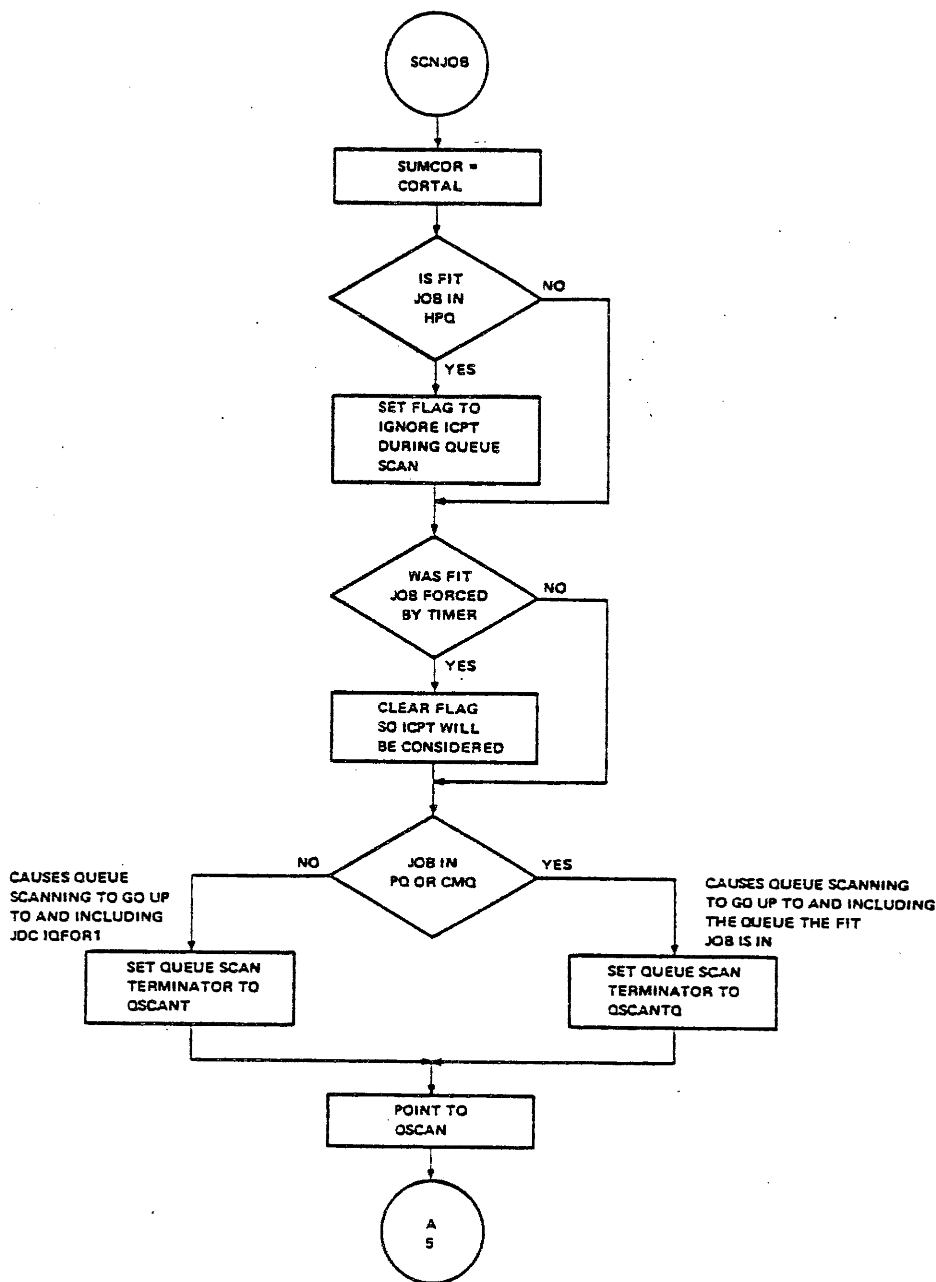
F-3

SWAP IN JOB CHOOSEN
NOW BRING IT IN



F-4

ESTABLISH THE PROPER SCAN TABLE FOR SWAP OUT JOB SELECTION AND DECIDE HOW MUCH OF THE TABLE TO SCAN



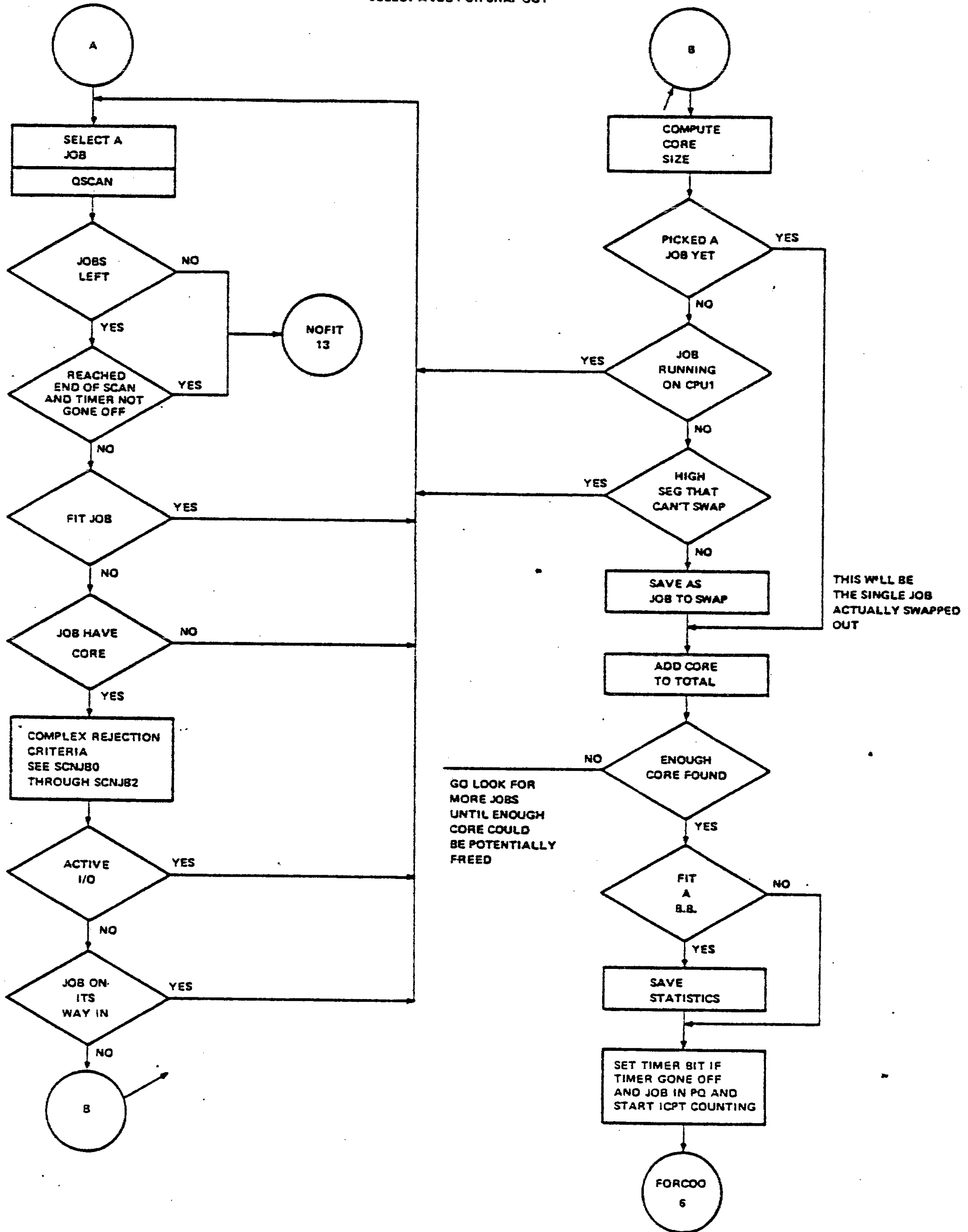
OSCAN HAS THE FOLLOWING ENTRIES

SEARCH LABEL	QUEUE	SEARCH CRITERIA
OSCAN	STOP	IQFOR
	SLP	IQFOR
	EW	IQFOR
	JDC	IQBAK1
	TI	IQFOR
QSCANT	JDC	IQFOR1
QSCANTQ	PQ2	OLFOR (INCLUDES IQBAK)
	PQ1	IQBAK
	CMQ	IQBAK
	HPQ?	IQBAK

MR-5018

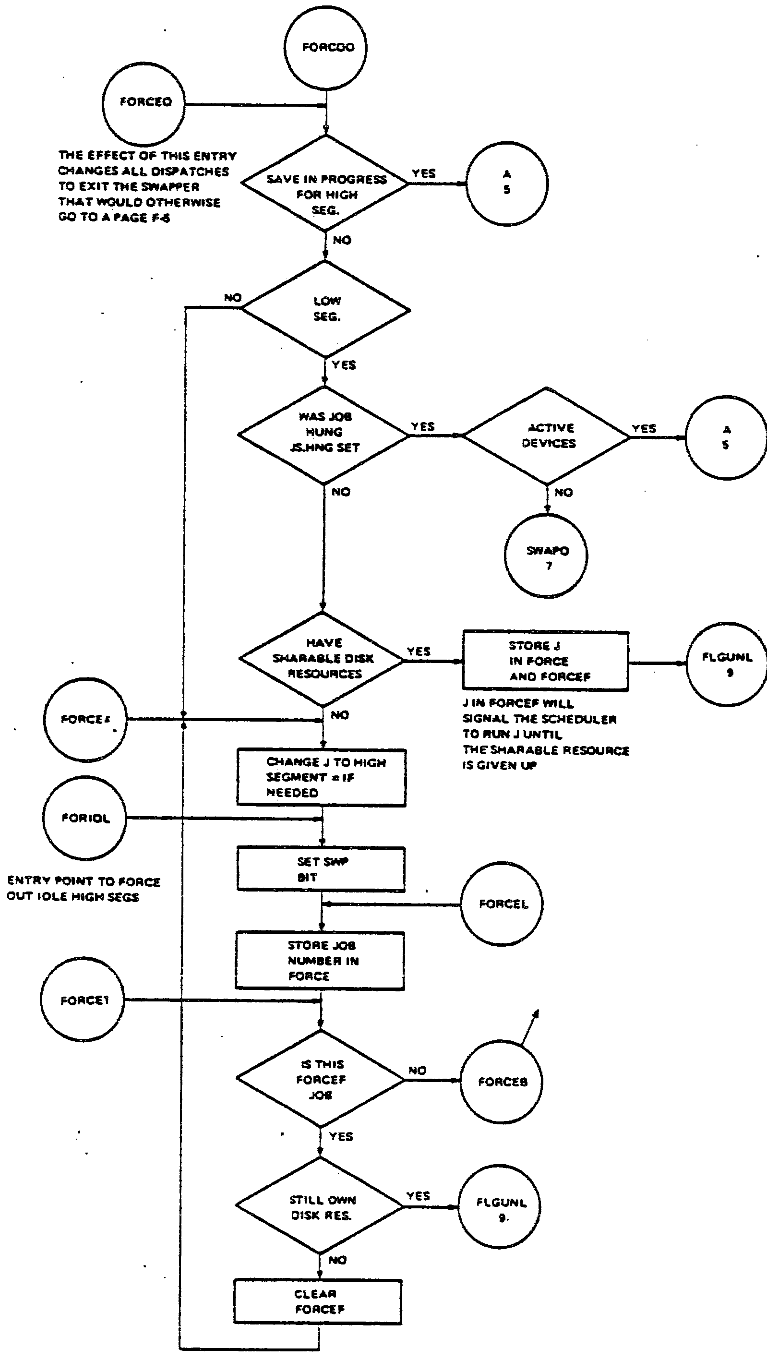
F-5

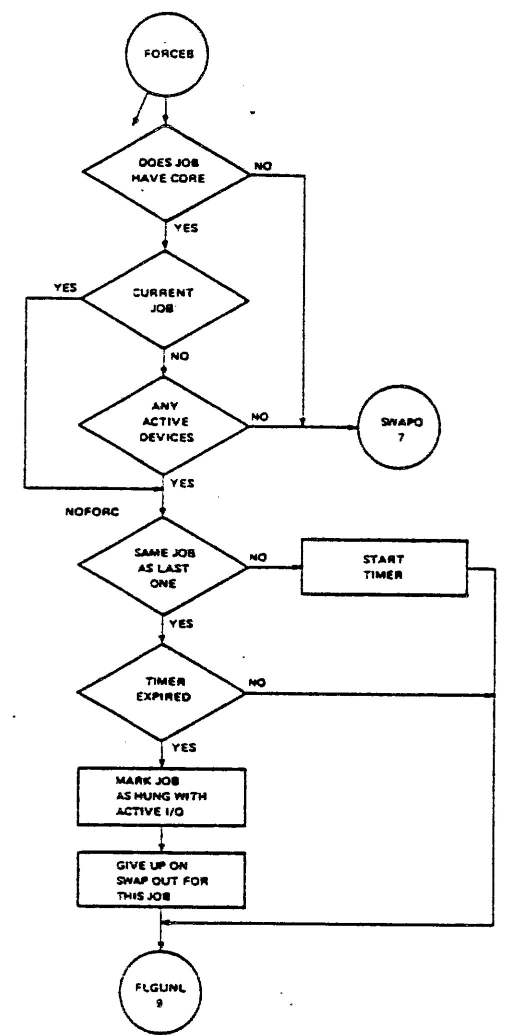
SELECT A JOB FOR SWAP OUT



F-6

DETERMINE IF JOB SELECTED FOR SWAP OUT CAN BE SWAPPED OR MUST IT WAIT FOR I/O TO STOP OR SHARABLE RESOURCES TO BE GIVEN UP

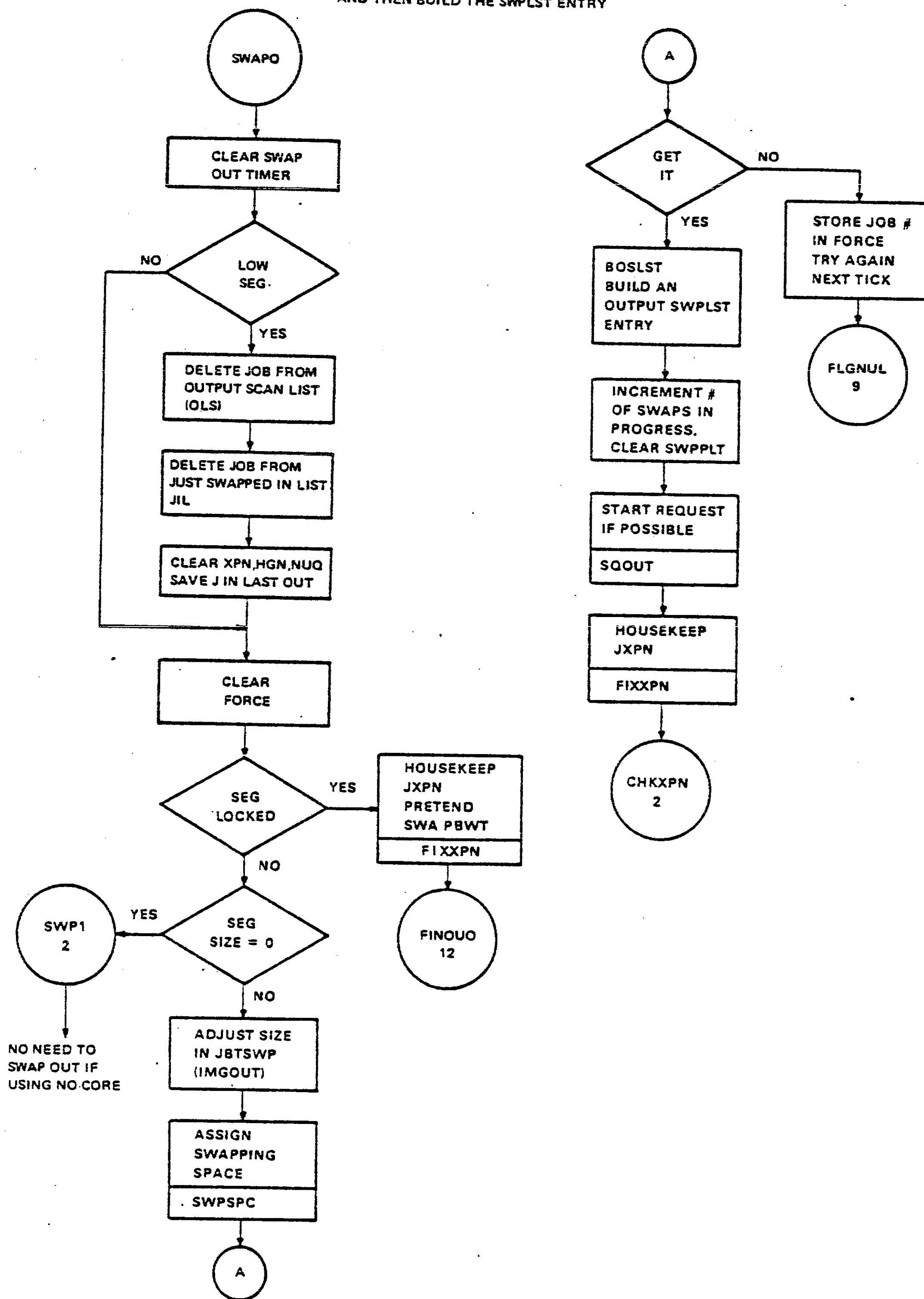




MR-5020

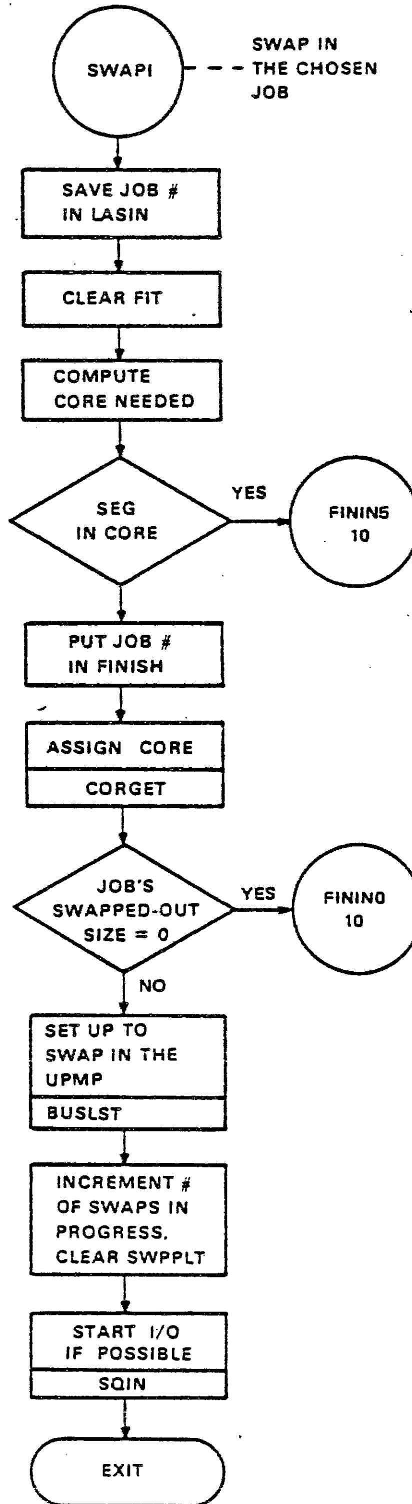
F7

DO SOME PRE-SWAPOUT HOUSEKEPPING AND THEN BUILD THE SWPLST ENTRY

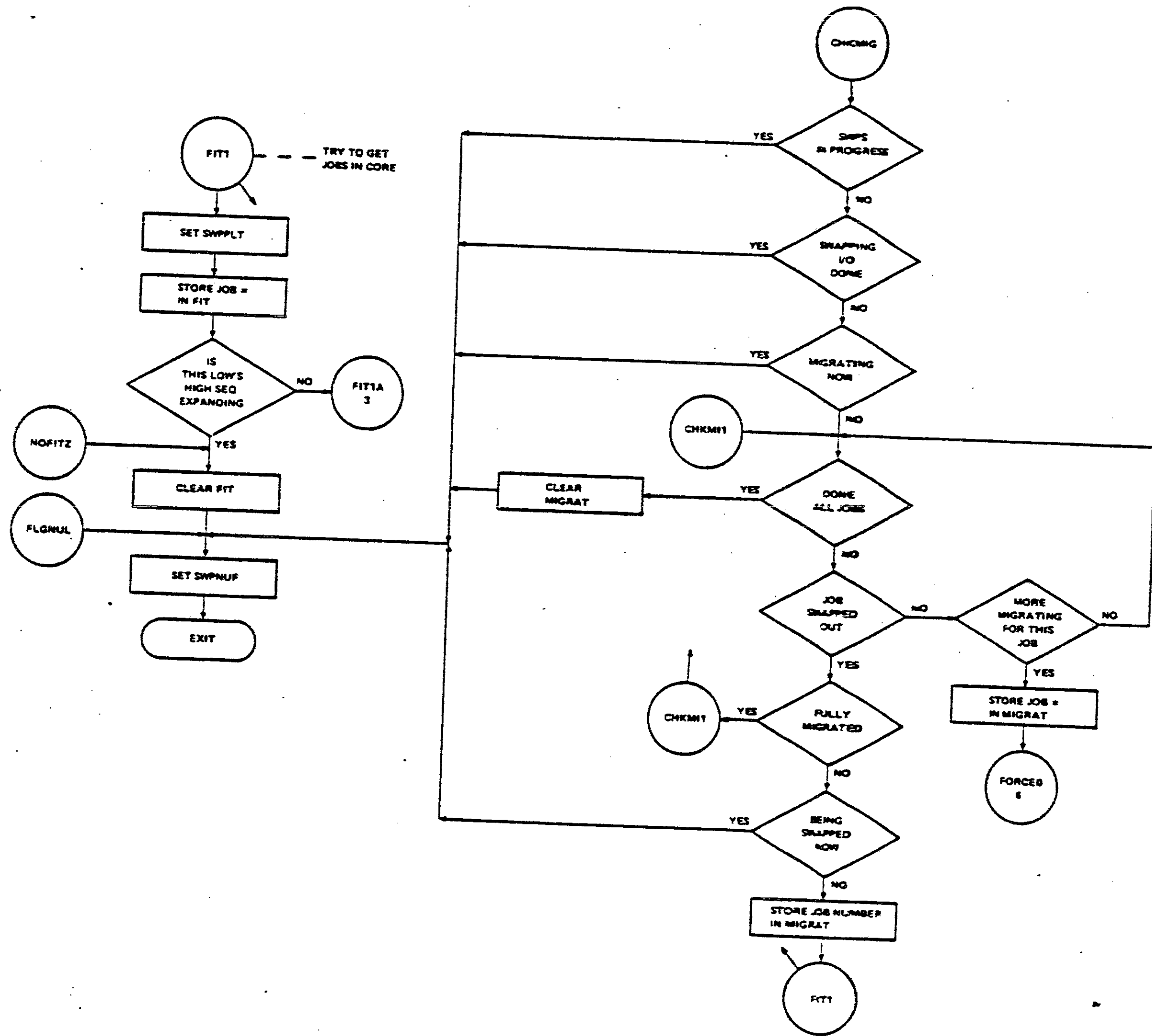


F-8

GET CORE FOR SWAP IN JOB AND MAKE SWPLST ENTRY

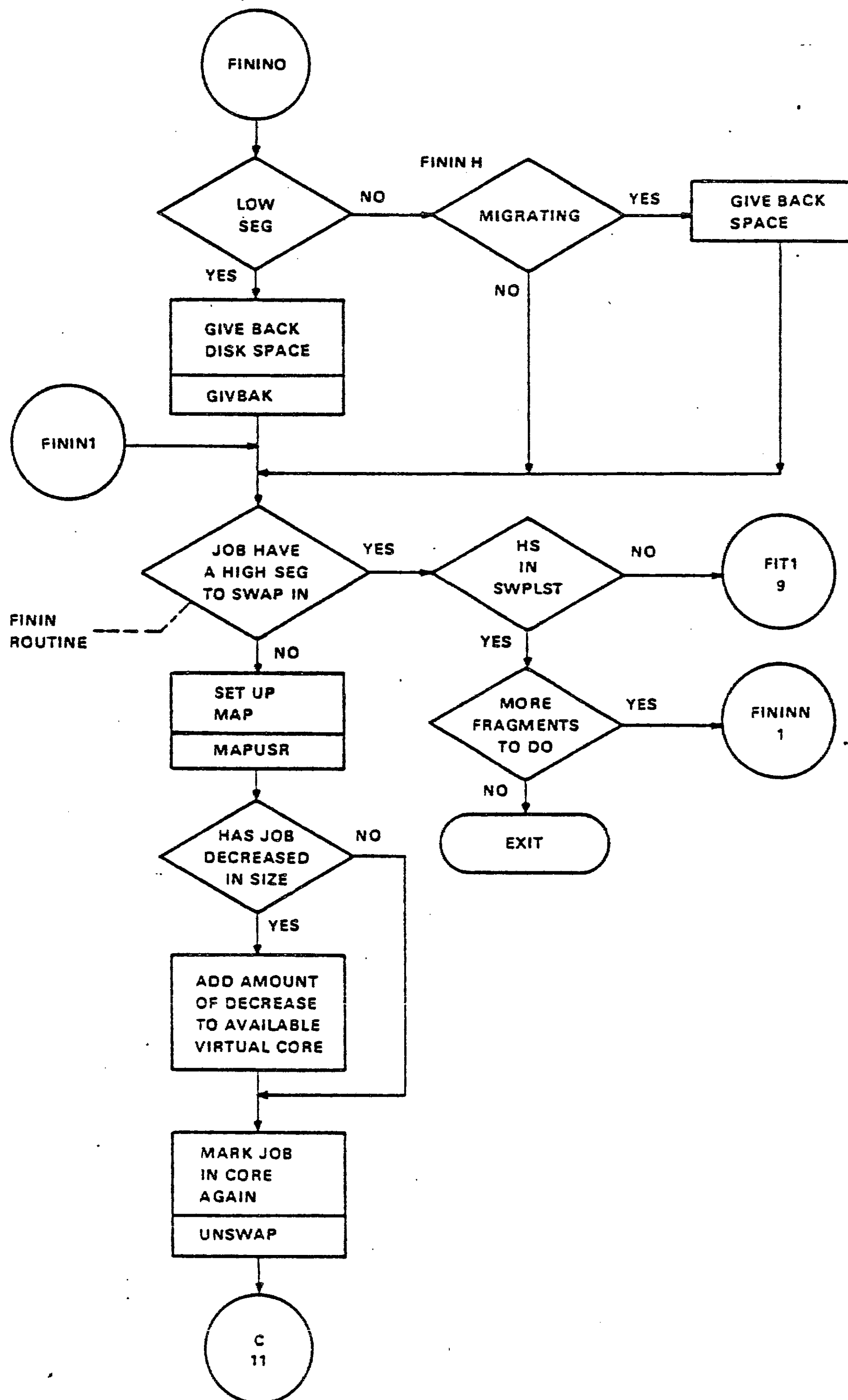


MR-5022


FIT1
TRY TO GET JOBS IN CORE
SET SWPPLT
STORE JOB = IN FIT
IS THIS LOW'S HIGH SEQ EXPANDING
NO
FIT1A 3
YES
NOFITZ
CLEAR FIT
FLGNUL
SET SWPNUF
EXIT
CHKMIG
YES
SWPS IN PROGRESS
NO
YES
SWAPPING I/O DONE
NO
YES
MIGRATING NOW
NO
CHKMI1
CLEAR MIGRAT
YES
DONE ALL JOBS
NO
JOB SWAPPED OUT
NO
MORE MIGRATING FOR THIS JOB
NO
YES
YES
STORE JOB = IN MIGRAT
FORCE0 6
CHKMI1
YES
FULLY MIGRATED
NO
YES
BEING SWAPPED NOW
NO
STORE JOB NUMBER IN MIGRAT
FIT1
MR-5023

F-10

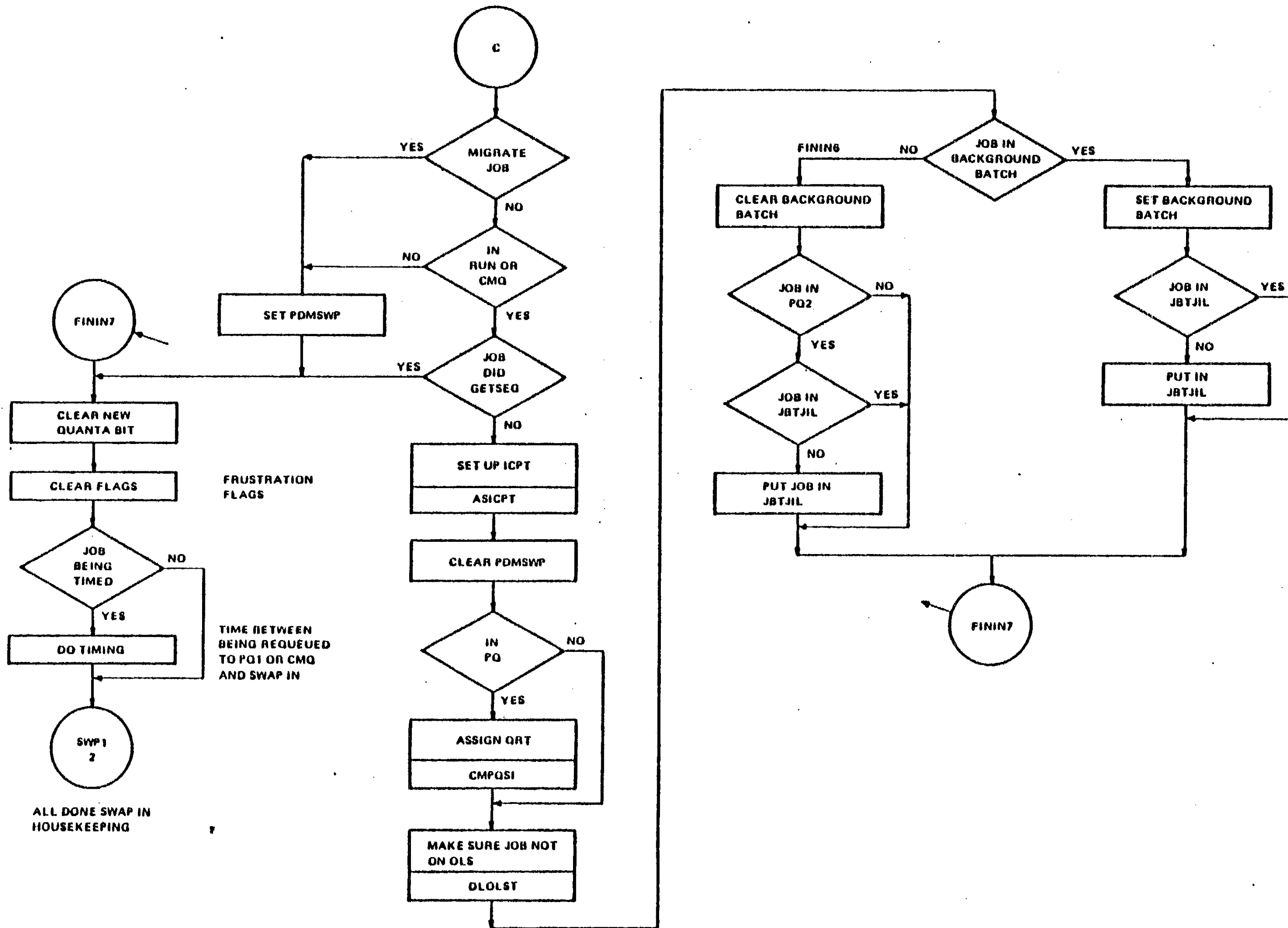
SWAP IN HOUSEKEEPING



MR-5024

F-11

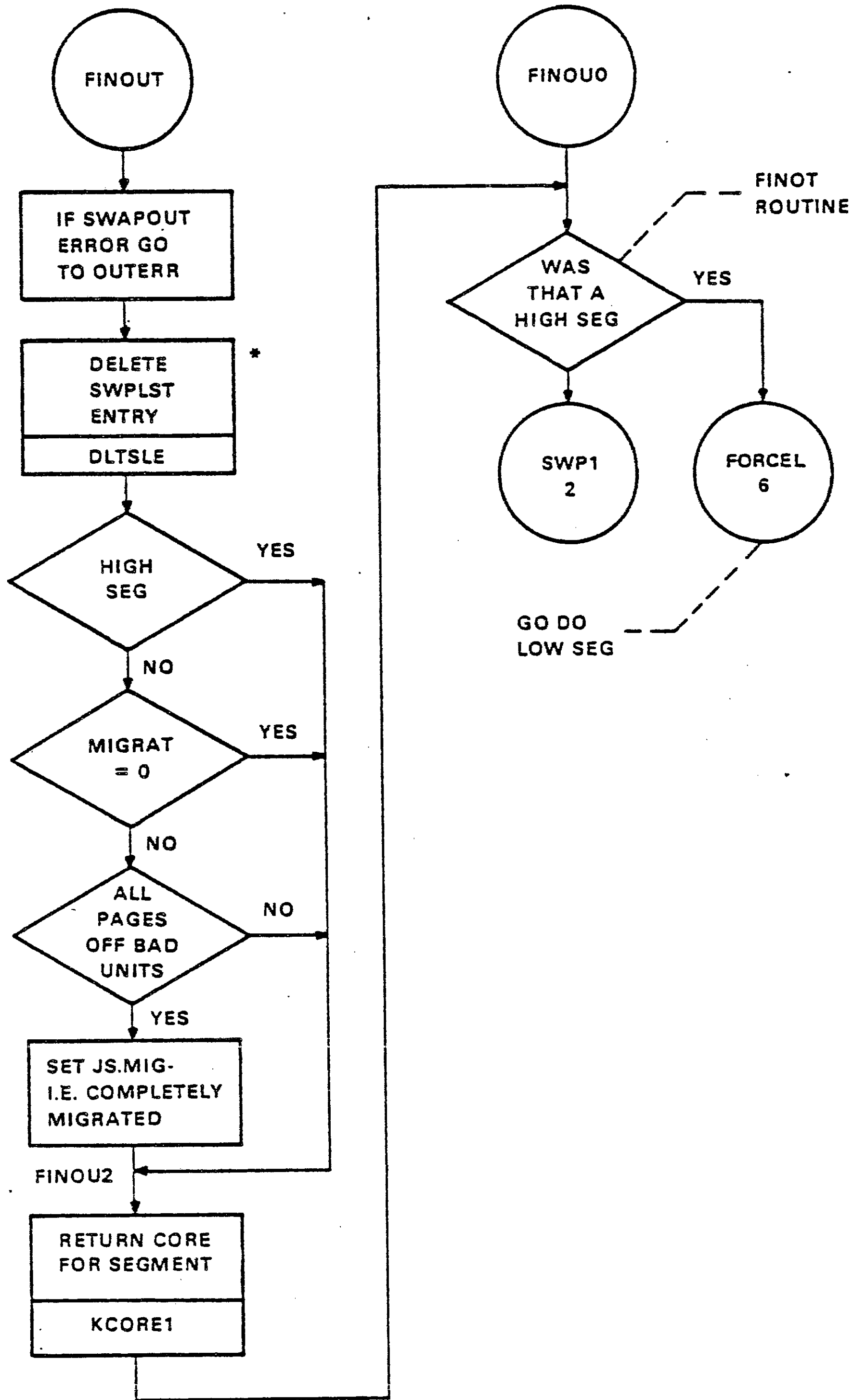
CONTINUATION OF SWAP IN HOUSEKEEPING



6-11

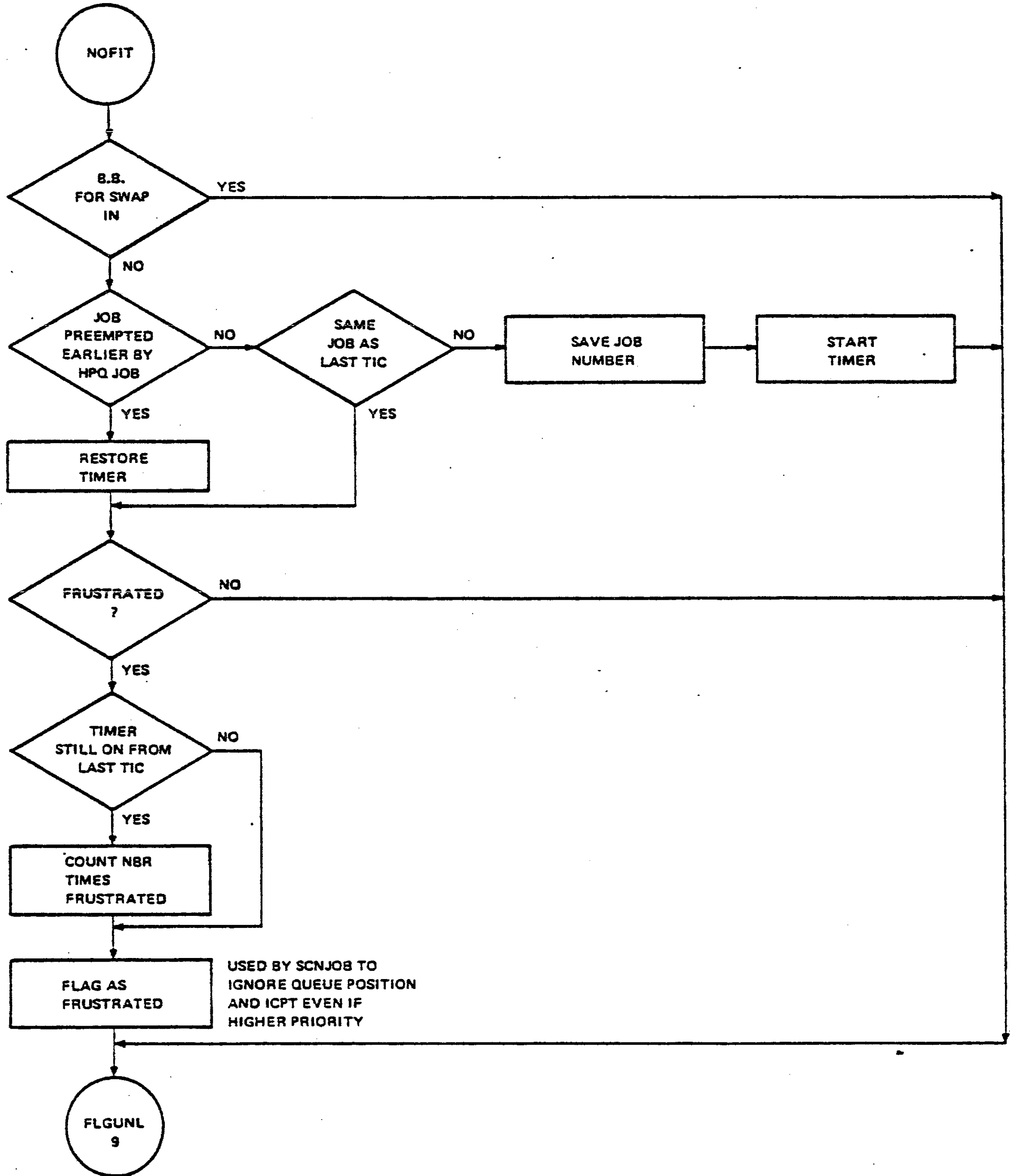
F-12

SWAP OUT HOUSEKEEPING

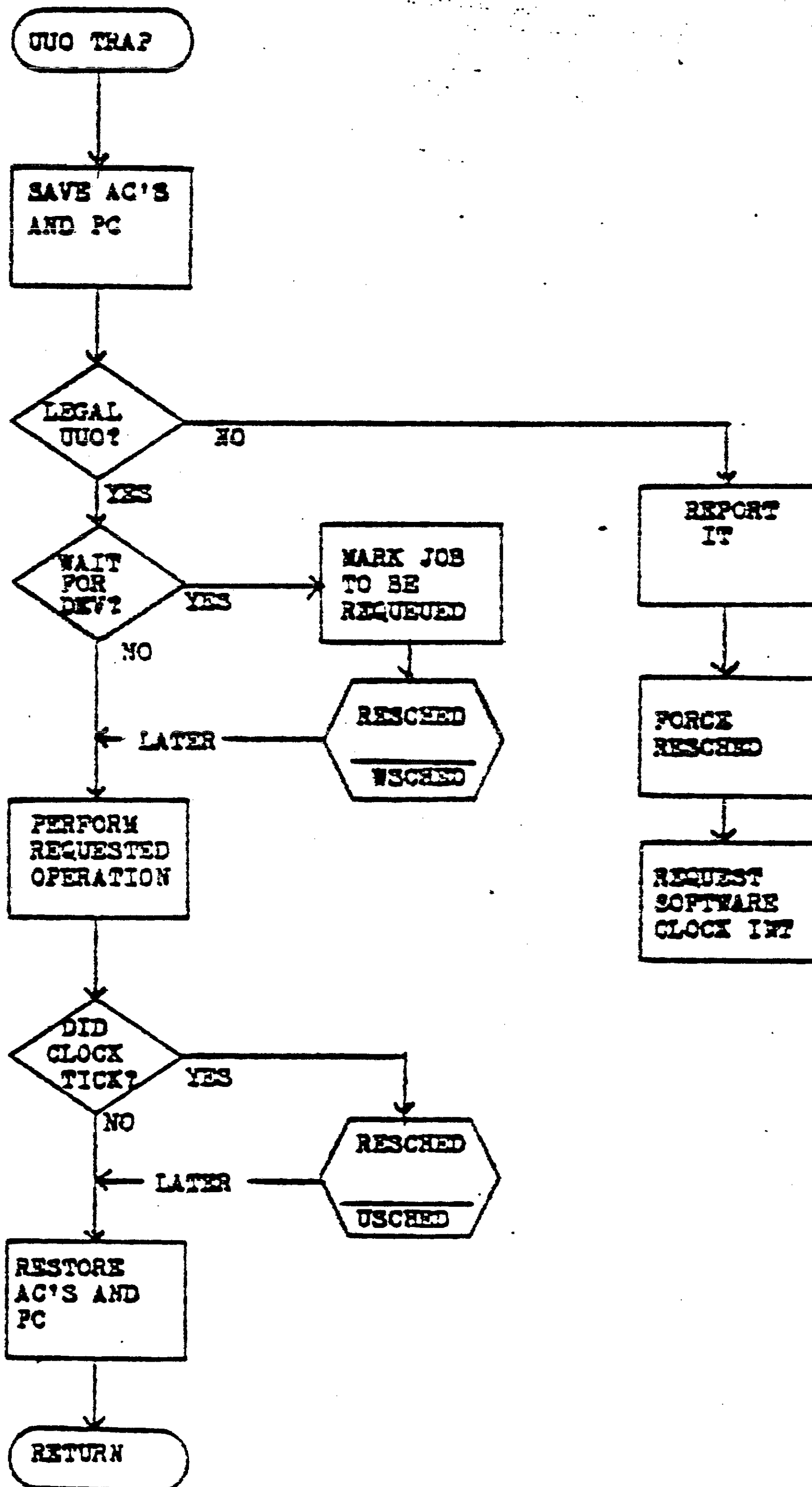


F-13

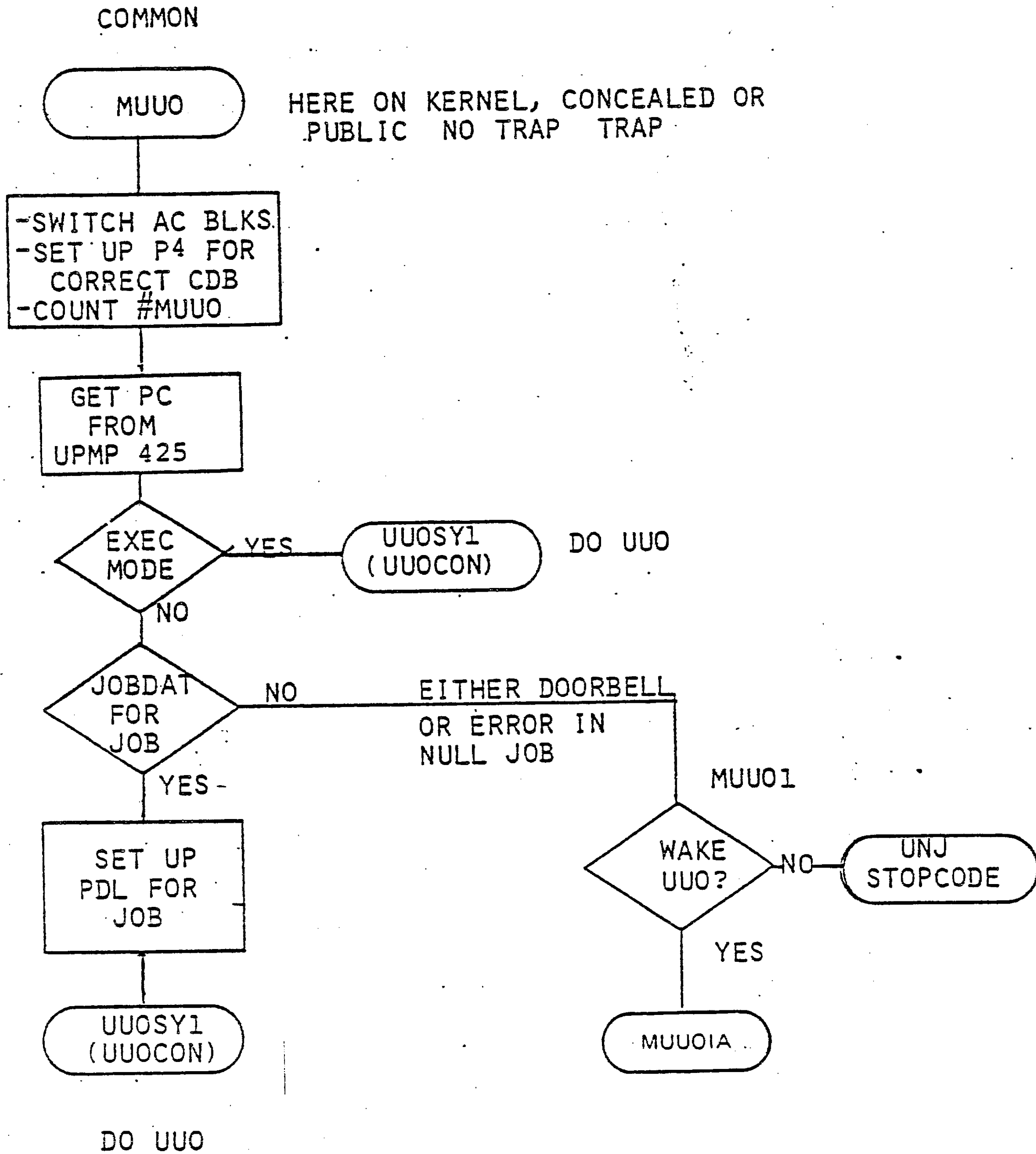
HERE WHEN ENOUGH CORE CANNOT BE FREED BY DELETING IDLE & DORMANT AND ENOUGH ELIGIBLE JOBS CANNOT BE FOUND TO SWAP OUT



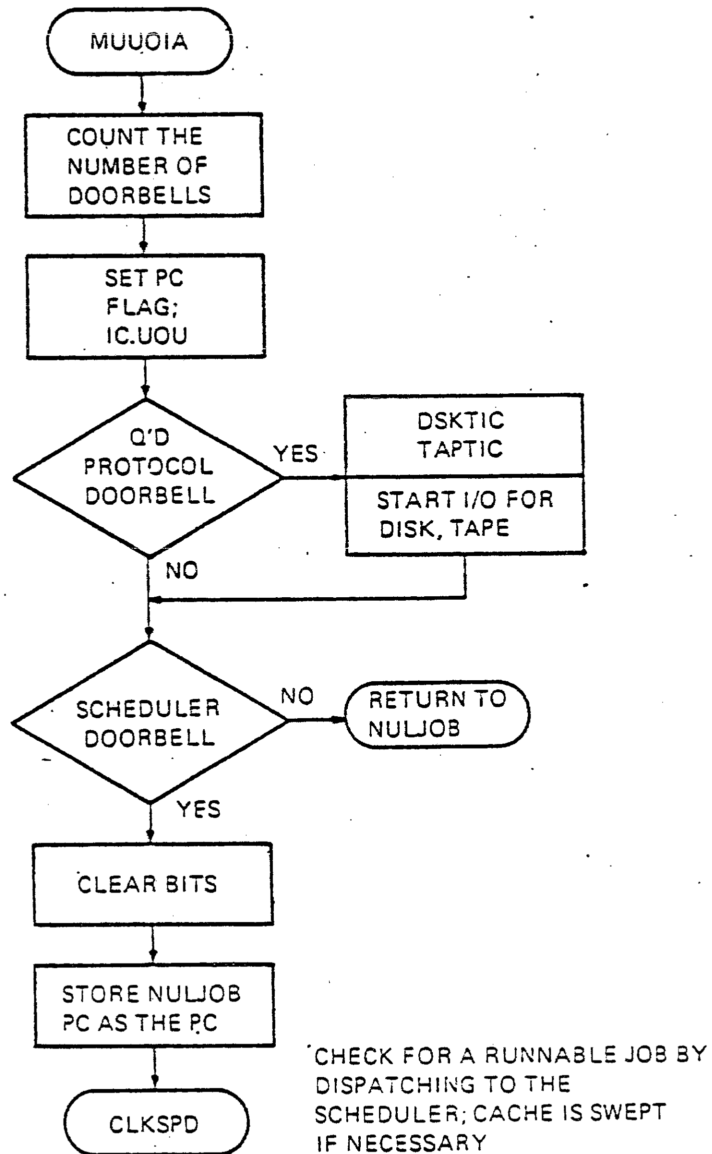
UUO FLOW

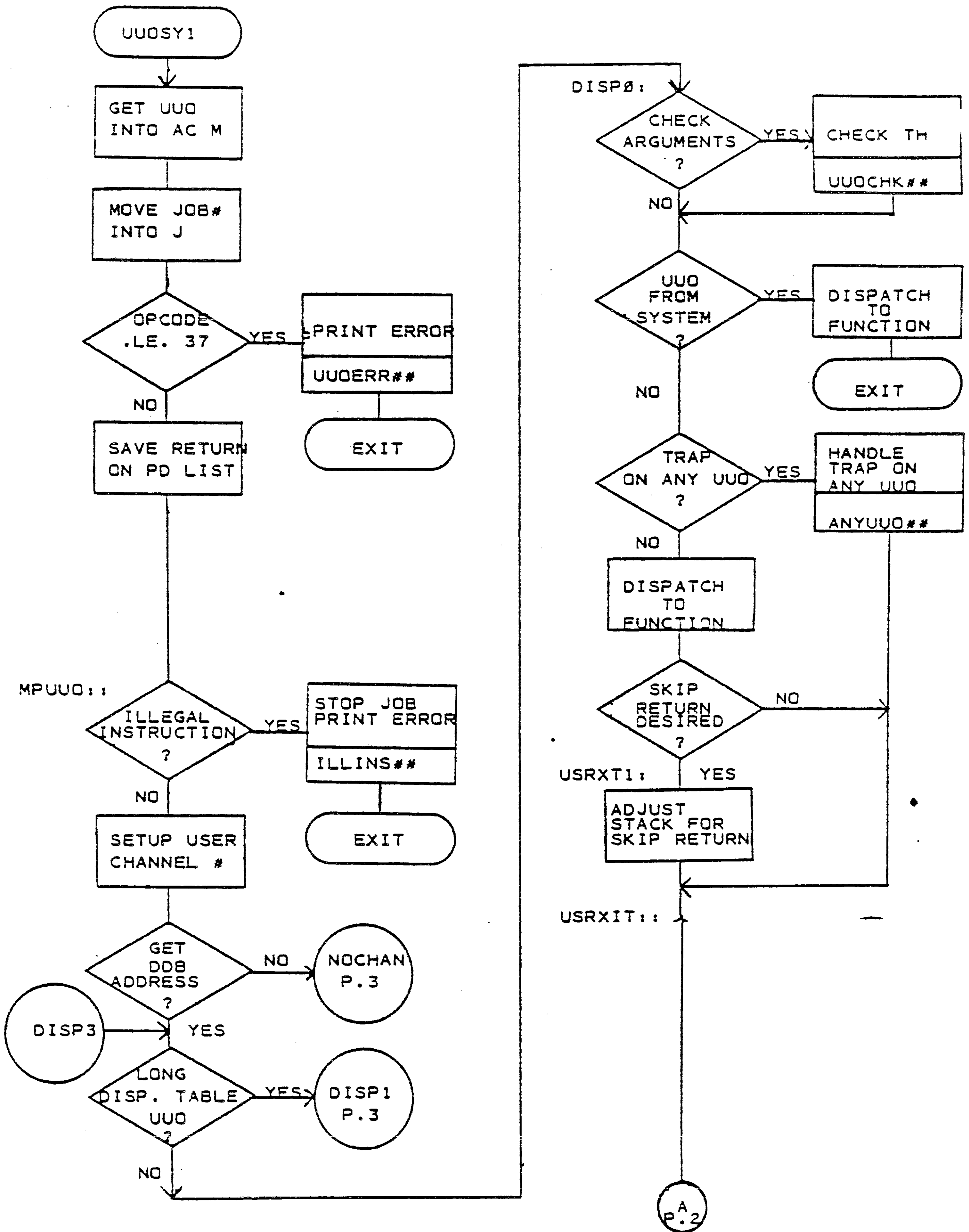


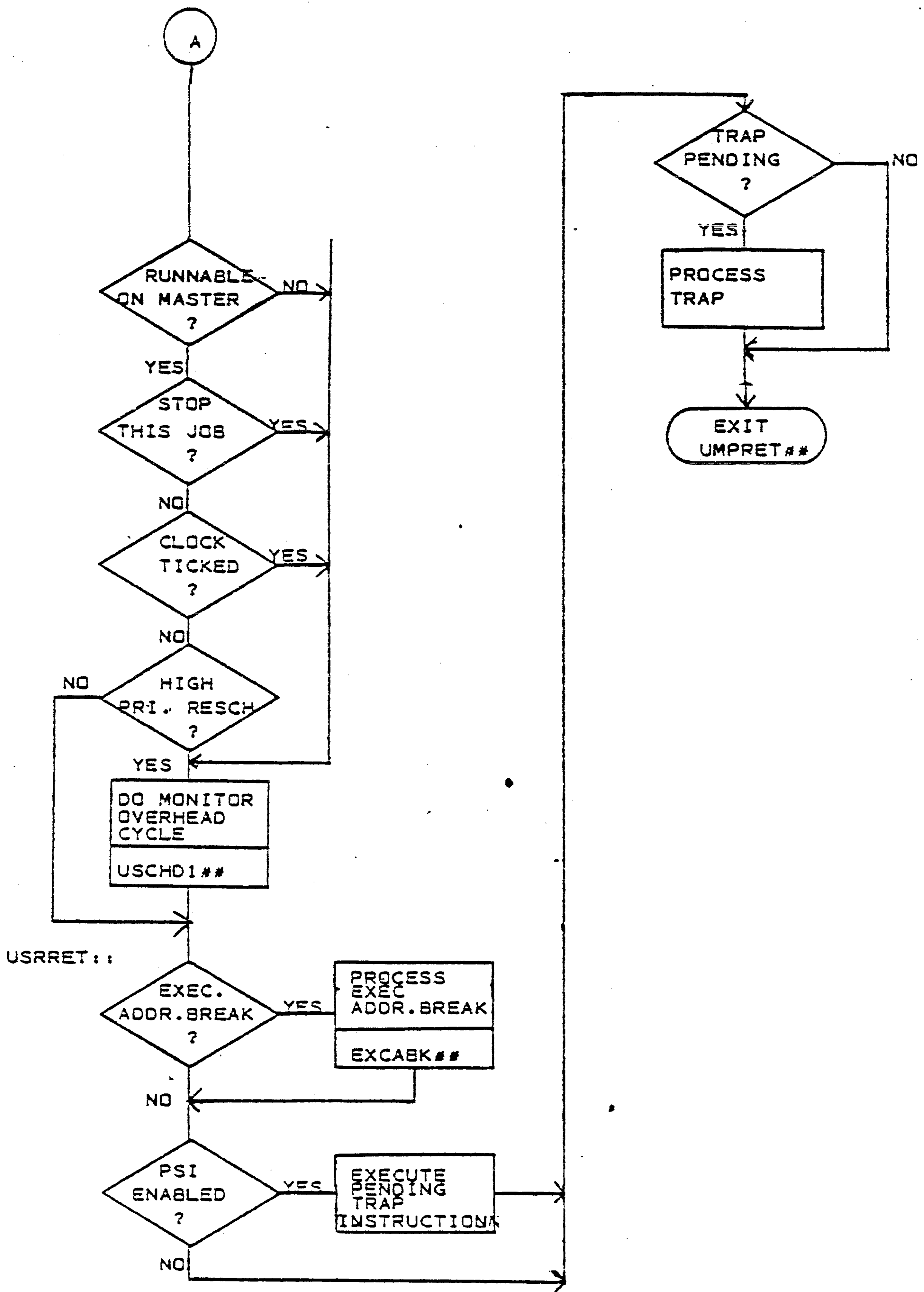
PRELIMINARY MUUO TRAP CODE

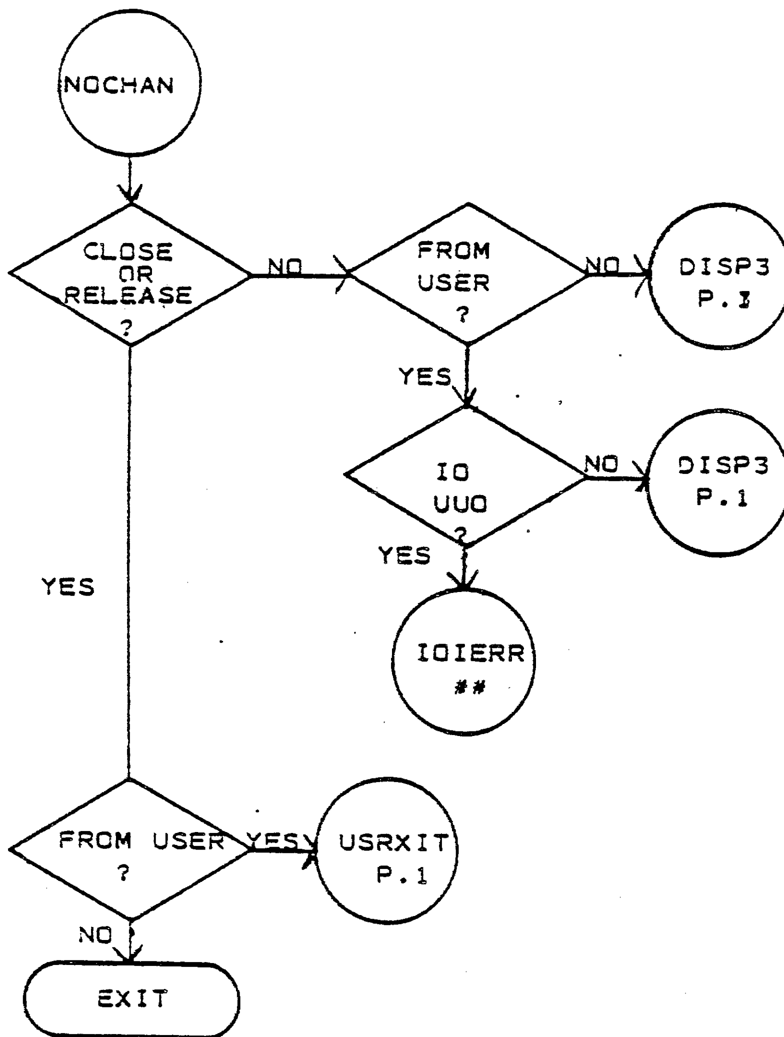


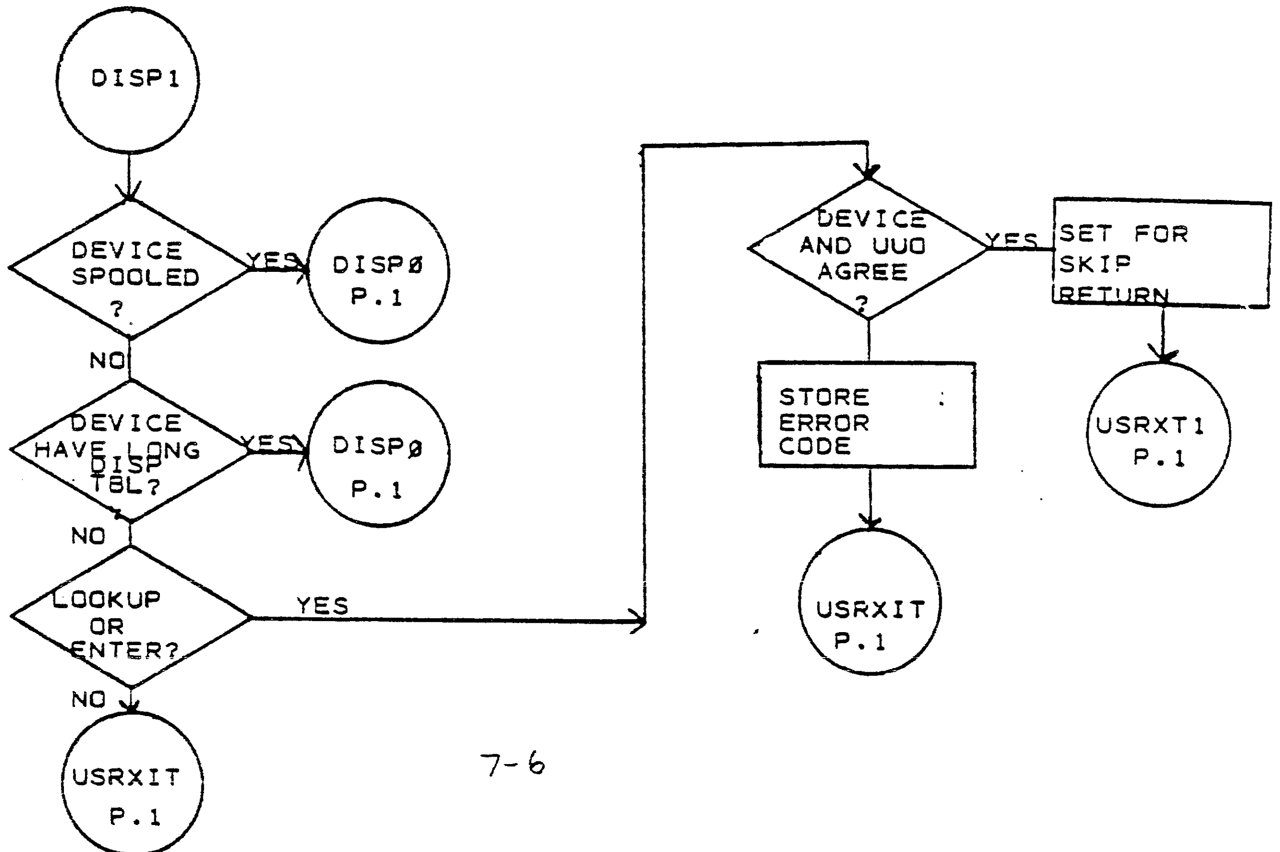
Detecting the CPU Doorbell




UUOSY1
GET UUO INTO AC M
MOVE JOB# INTO J
OPCODE .LE. 37
YES
PRINT ERROR
UUOERR##
EXIT
NO
SAVE RETURN ON PD LIST
MPUUO::
ILLEGAL INSTRUCTION ?
YES
STOP JOB PRINT ERROR
ILLINS##
EXIT
NO
SETUP USER CHANNEL #
GET DDB ADDRESS ?
NO
NOCHAN P.3
DISP3
YES
LONG DISP. TABLE UUO ?
YES
DISP1 P.3
NO
DISPØ:
CHECK ARGUMENTS ?
YES
CHECK TH
UUOCHK##
NO
UUO FROM SYSTEM ?
YES
DISPATCH TO FUNCTION
EXIT
NO
TRAP ON ANY UUO ?
YES
HANDLE TRAP ON ANY UUO
ANYUUO##
NO
DISPATCH TO FUNCTION
SKIP RETURN DESIRED ?
NO
USRXT1:
YES
ADJUST STACK FOR SKIP RETURN
USRXIT::
A P.2


A
RUNNABLE ON MASTER ?
NO
YES
STOP THIS JOB ?
YES
NO
CLOCK TICKED ?
YES
NO
HIGH PRI. RESCH ?
NO
YES
DO MONITOR OVERHEAD CYCLE
USCHD1##
USRRET::
EXEC. ADDR.BREAK ?
YES
PROCESS EXEC ADDR.BREAK
EXCABK##
NO
PSI ENABLED ?
YES
EXECUTE PENDING TRAP INSTRUCTION
NO
TRAP PENDING ?
NO
YES
PROCESS TRAP
EXIT UMPRET##


NOCHAN
CLOSE
OR
RELEASE
?
NO
FROM
USER
?
NO
DISP3
P.1
YES
IO
UUO
?
NO
DISP3
P.1
YES
IOIERR
##
YES
FROM USER
?
YES
USRXIT
P.1
NO
EXIT


DISP1
DEVICE
SPOOLED
?
YES
DISPØ
P.1
NO
DEVICE
HAVE LONG
DISP
TBL?
YES
DISPØ
P.1
NO
LOOKUP
OR
ENTER?
YES
NO
USRXIT
P.1
DEVICE
AND UUO
AGREE
?
YES
SET FOR
SKIP
RETURN
STORE
ERROR
CODE
USRXT1
P.1
USRXIT
P.1

STOPCD MACRO

STOPCD CONT, TYPE, NAME, DISP

CONT - LOCATION TO JUMP TO AFTER PROCESSING ERROR

TYPE - TYPE OF FAILURE, USED TO DETERMINE NEXT COURSE OF ACTION

HALT
STOP
JOB
DEBUG
CPU

NAME - UNIQUE THREE LETTER NAME, WILL BE EXPANDED TO FORM GLOBAL LABEL

S..NAME

DISP - ADDRESS OF ROUTINE TO TYPE ADDITIONAL INFORMATION (USUALLY NOT SPECIFIED)

CODE GENERATED VIA

STOPCD MACRO

HALT TYPE

```
        STOPCD      CONT,   TYPE,   NAME
                ⇓
S..NAME ::    HALT      CONT
```

CODE GENERATED VIA
STOPCD MACRO

DEBUG, JOB, STOP TYPES

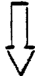
- IF CONT IS A SYMBOLIC ADDRESS

```
STOPCD      CONT,   TYPE,   NAME
```

```
S.. NAME :: PUSHJ    P,DIE
            CAIA     TYPE, (SIXBIT/NAME/)(17)
            JRST     CONT
```

- IF CONT IS . OR .+1 OR CPOPJ OR CPOPJ1

```
STOPCD  .,   TYPE,   NAME
```



```
S..NAME :: PUSHJ  P,DIE
           CAI    TYPE, (SIXBIT/NAME/)(CONTINUE TYPE)
```

CODE GENERATED VIA
STOPCD MACRO

RECOVERABLE STOPCD

SOURCE CODE

```
ROUT:          ⋮
               CONDITIONAL TEST                ; EVERYTHING OK ?
               STOPCD  CONT, TYPE, NAME        ; NO,
                 ⋮                             ; YES.
               POPJ  P,

CONT:            ⋮                             ; SYSTEM CONTINUE
                                               ; ROUTINE
               POPJ  P,
```

EXPANSION

```
ROUT:            ⋮
               CONDITIONAL TEST
S..NAME::      PUSHJ    P,DIE
               CAIA   TYPE, (SIXBIT/NAME/)(17)  ; PARAM FOR DIE
               JRST   CONT                      ; WHERE TO GO
                 ⋮                                IF WE COME BACK
               POPJ  P,

CONT:            ⋮
               POPJ  P,
```

CODE GENERATED VIA STOPCD MACRO

NONRECOVERABLE STOPCD

SOURCE

```
ROUT:
           ~
        CONDITIONAL TEST
        STOPCD  ., TYPE, NAME     ; DON'T COME BACK
           ~
        POPJ  P,
```

EXPANSION

```
ROUT:
           ~
          CONDITIONAL TEST              ; EVERYTHING OK?
S..NAME:: PUSHJ   P,DIE                 ; NO, DIE
          CAI  TYPE,(SIXBIT/NAME/)(1);  YES, NO-OP
           ~                            ; CONTINUE
          POPJ  P,
```

EFFECT OF STOPCD TYPES

TYPE / LEVEL	DEBUG	JOB	STOP	HALT
PI	CONTINUE SYSTEM	RELOAD	RELOAD	HALT
NON-PI	JOB ABORTED ⟹		RELOAD	HALT

CPU

SINGLE CPU OR LAST CPU OF MULTI CPU	RELOAD
MULTI-CPU	JUMP INTO AC

FINDING THE FAILING CYCLE

PISTS: CONI PI, PISTS — BEFORE THE CRASH

PISTS: 010000..150377 — AFTER THE CRASH

377 → PI ACTIVE
ALL CHANNELS
ON

150 → INTERRUPTS IN PROGRESS
ON CH1 AND CH3

PUSHDOWN LISTS

UUO CYCLE ⇨ 340510

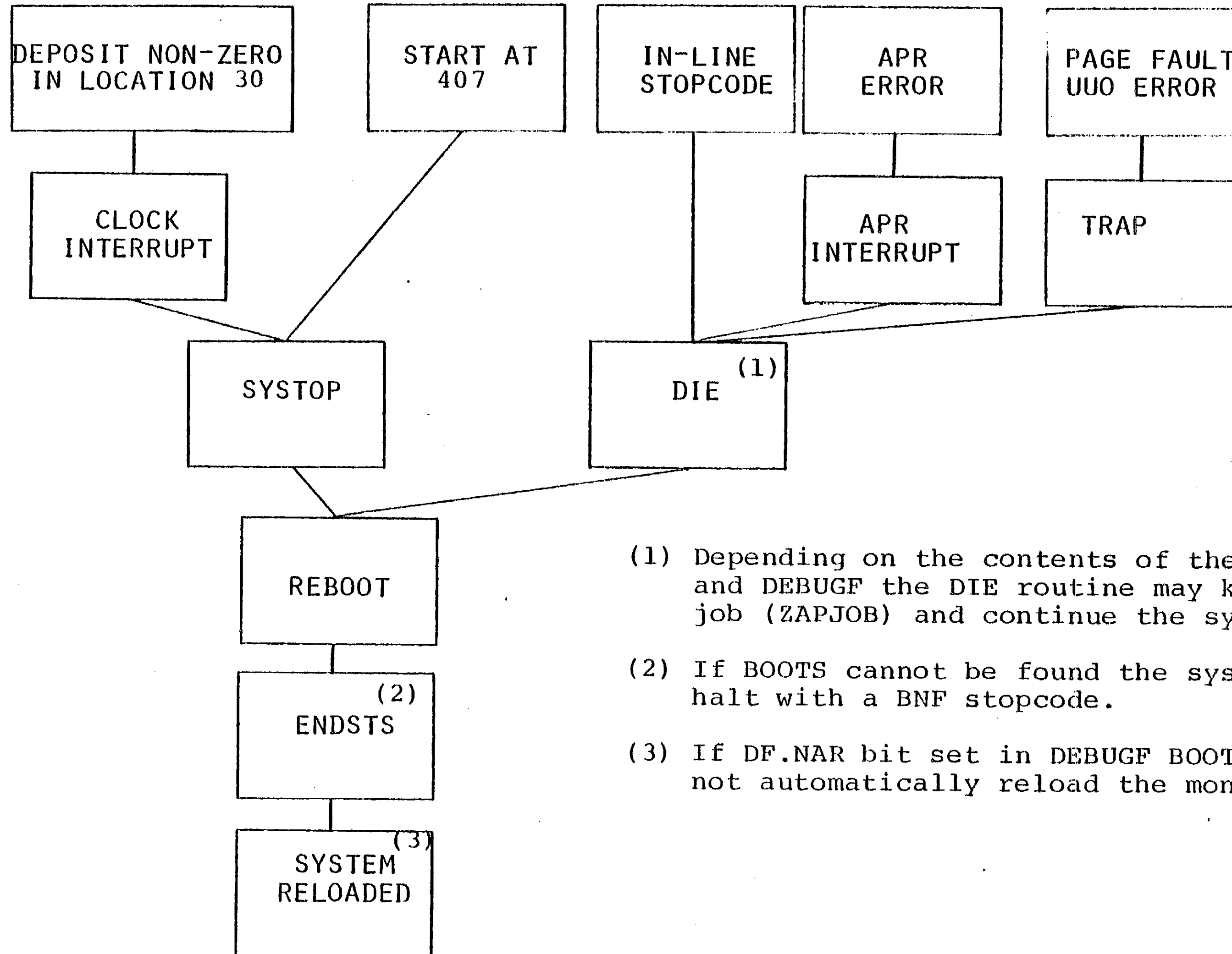
MONITOR CYCLE ⇨ NULPDL *

DEVICE INTERRUPT CYCLE ⇨ C1PD1
C2PD1
C3PD1
C4PD1
C5PD1
C6PD1

INTERRUPT LEVEL ↑ (points to the digit in C6PD1)

* Typed by FILDDT as ONCPDL + n

HOW TOPS10 DIES



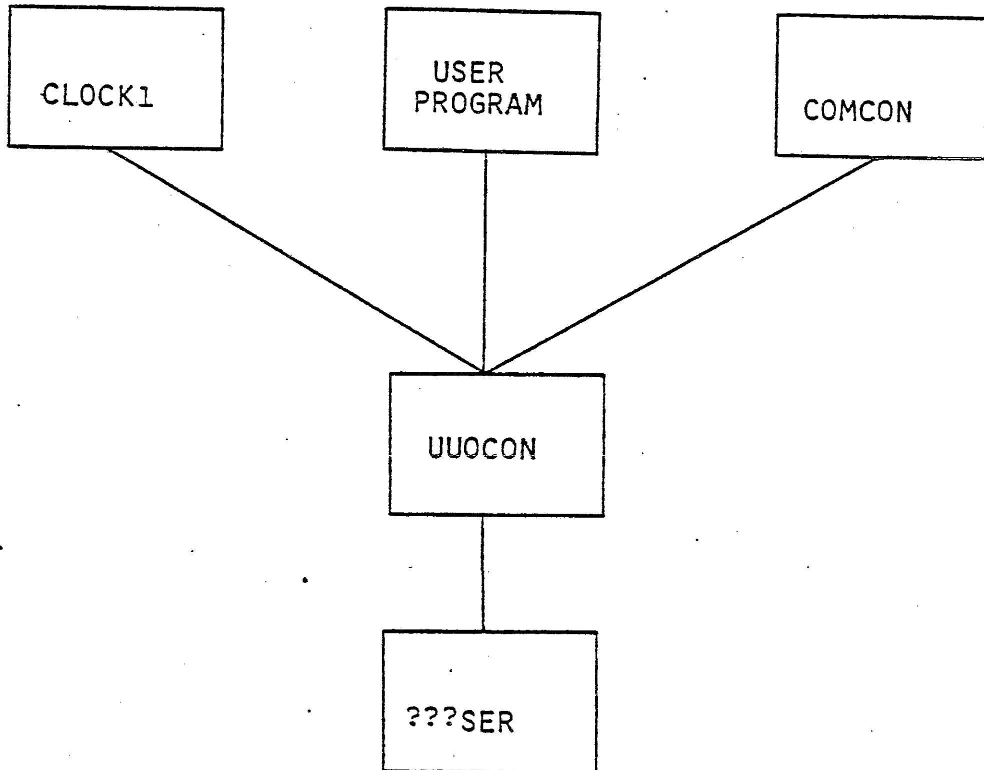
(1) Depending on the contents of the STATES and DEBUGF the DIE routine may kill the job (ZAPJOB) and continue the system.

(2) If BOOTS cannot be found the system will halt with a BNF stopcode.

(3) If DF.NAR bit set in DEBUGF BOOTS will not automatically reload the monitor.

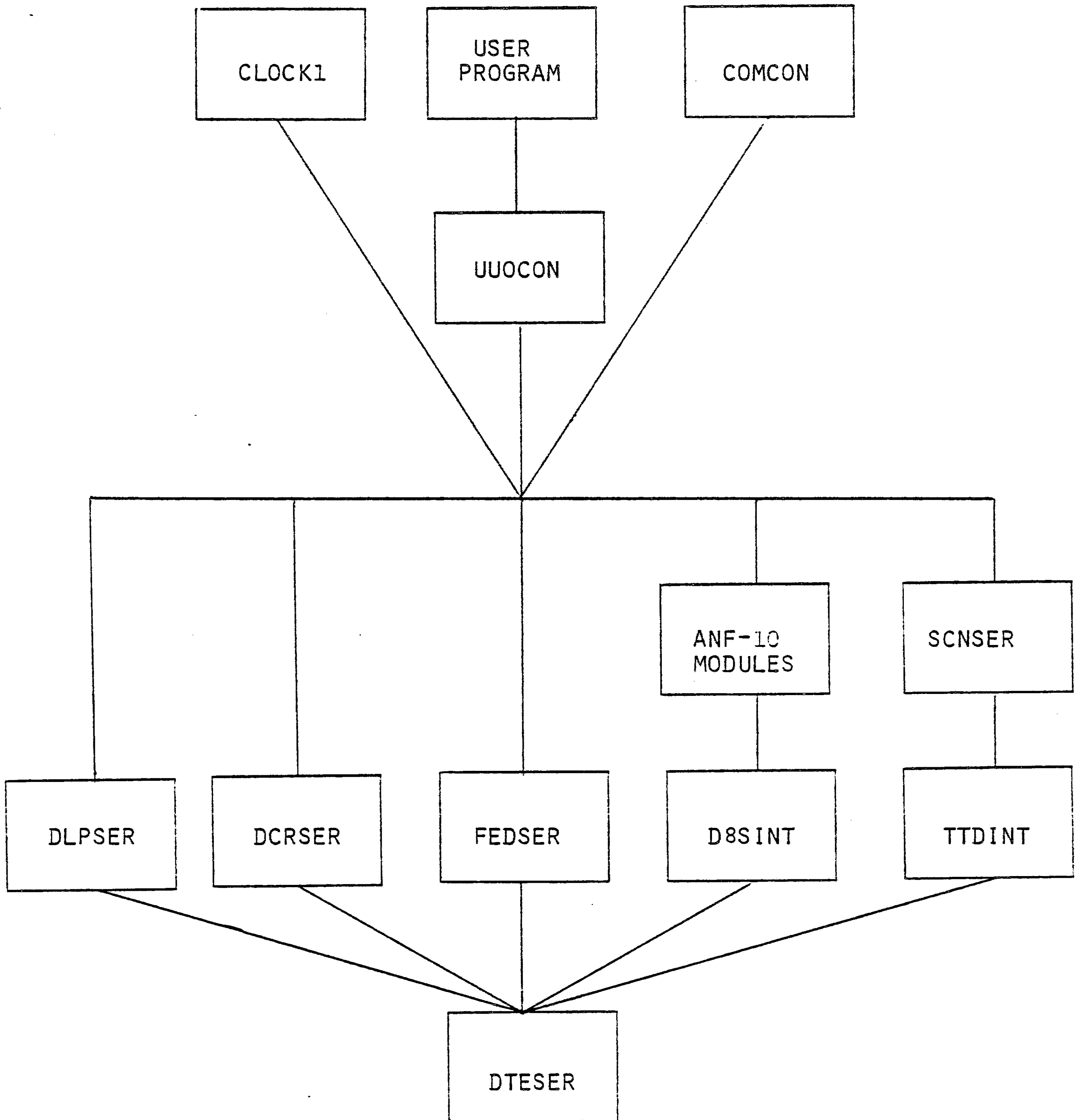
I/O BUS I/O MODULE

ARCHITECTURE

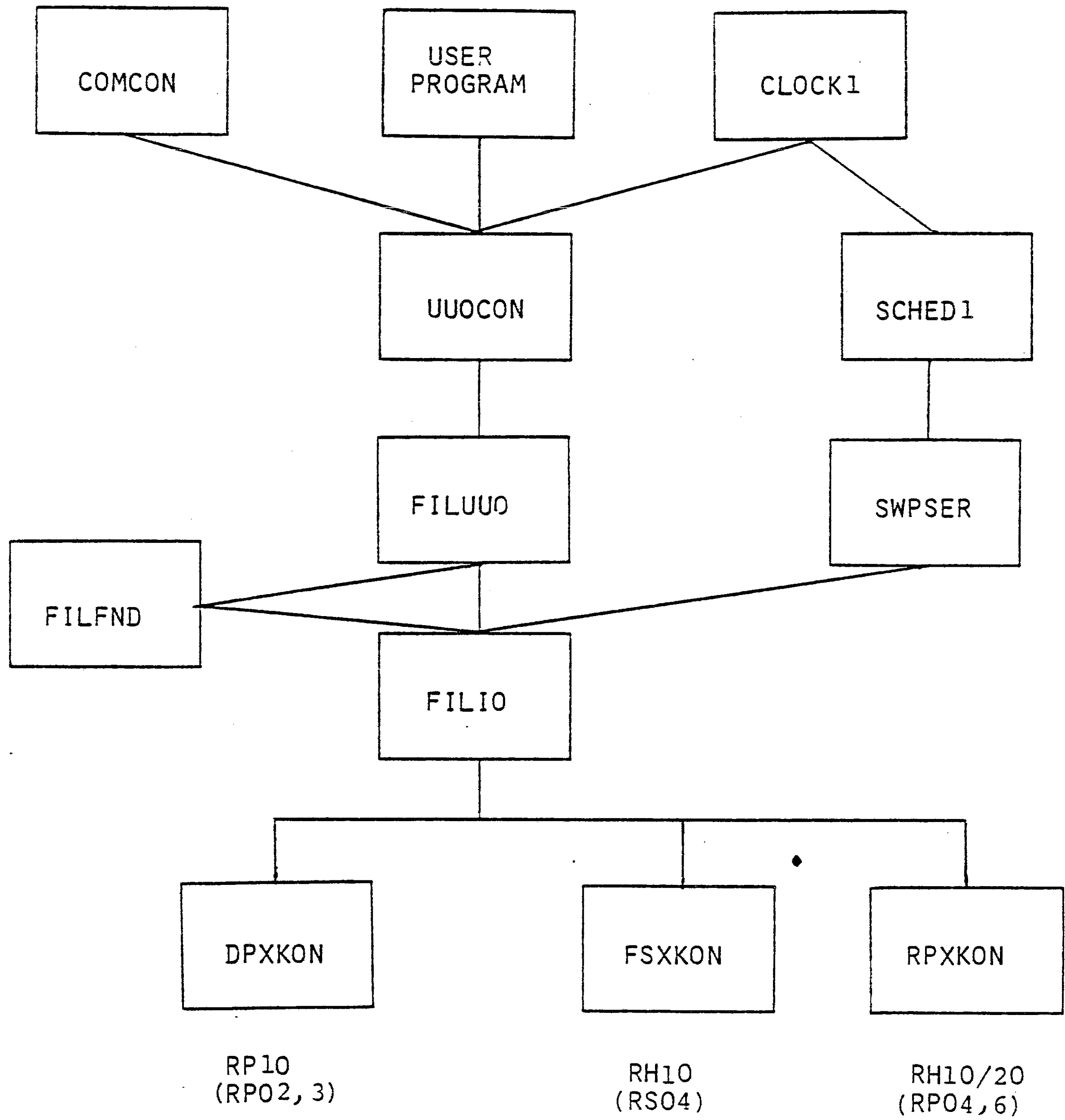


PTPSER
PTRSER
CDPSER
LPTSER
PLTSER
DTASER
CDRSER

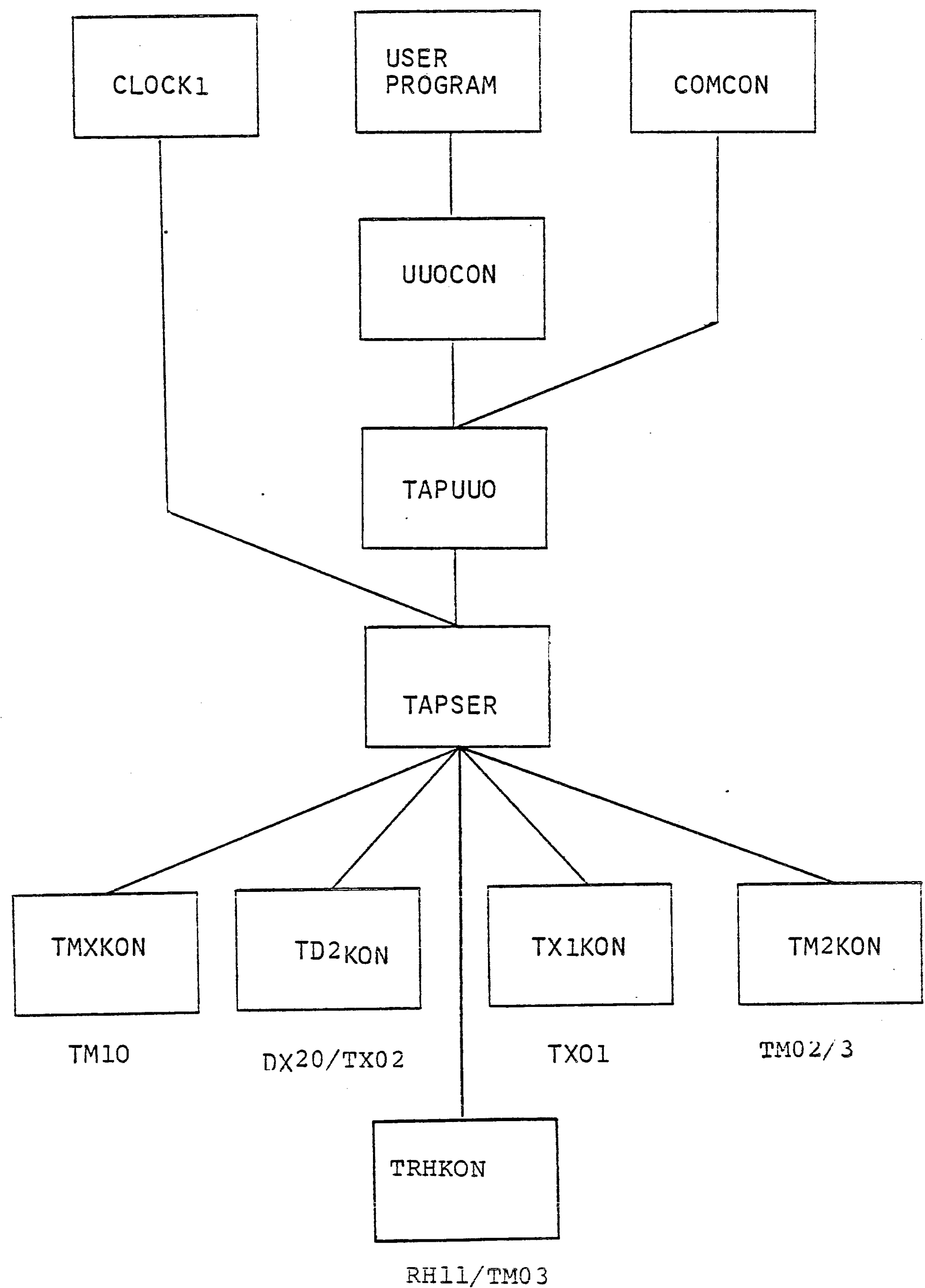
DTE BASED I/O MODULE ARCHITECTURE



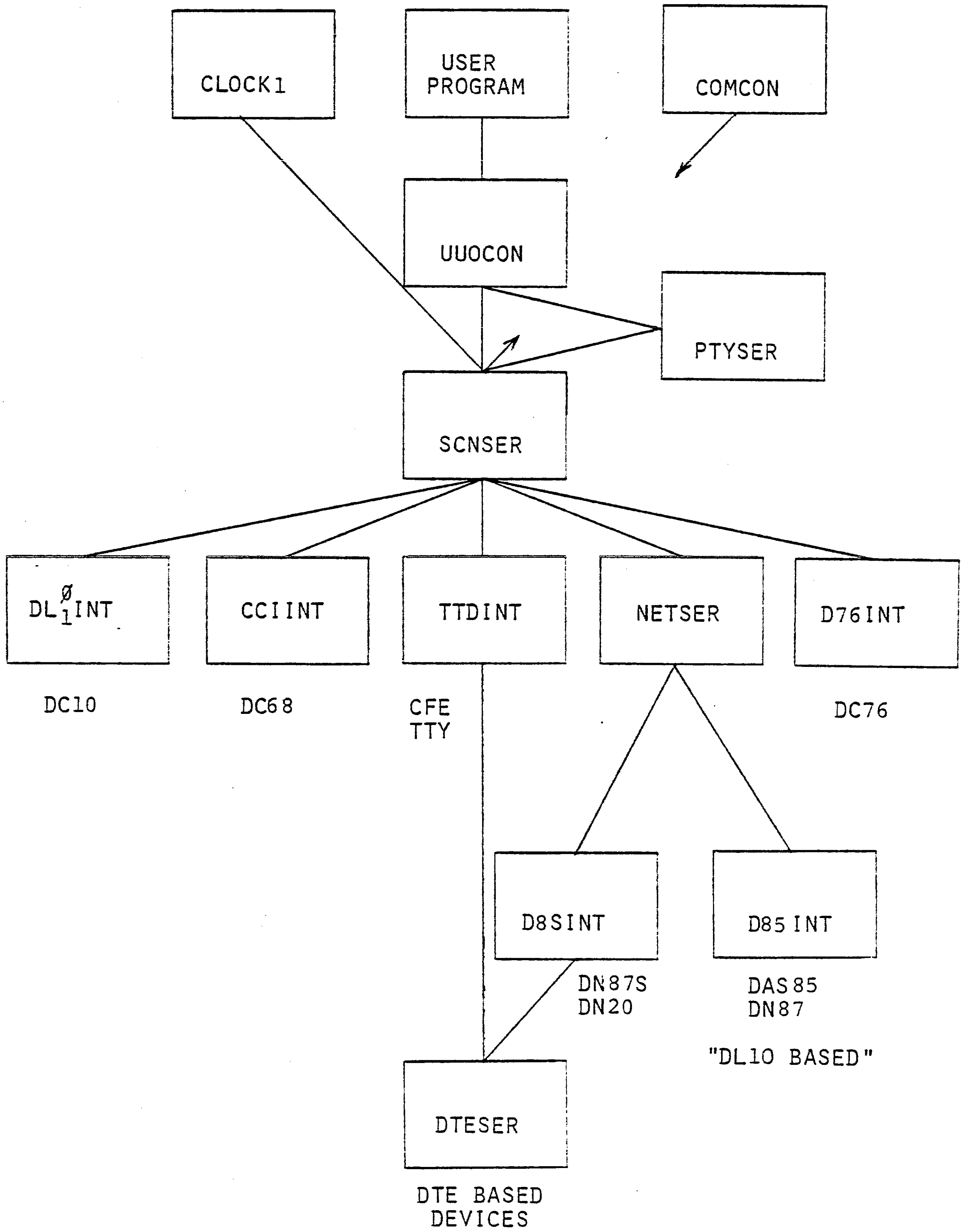
DISK I/O MODULE ARCHITECTURE



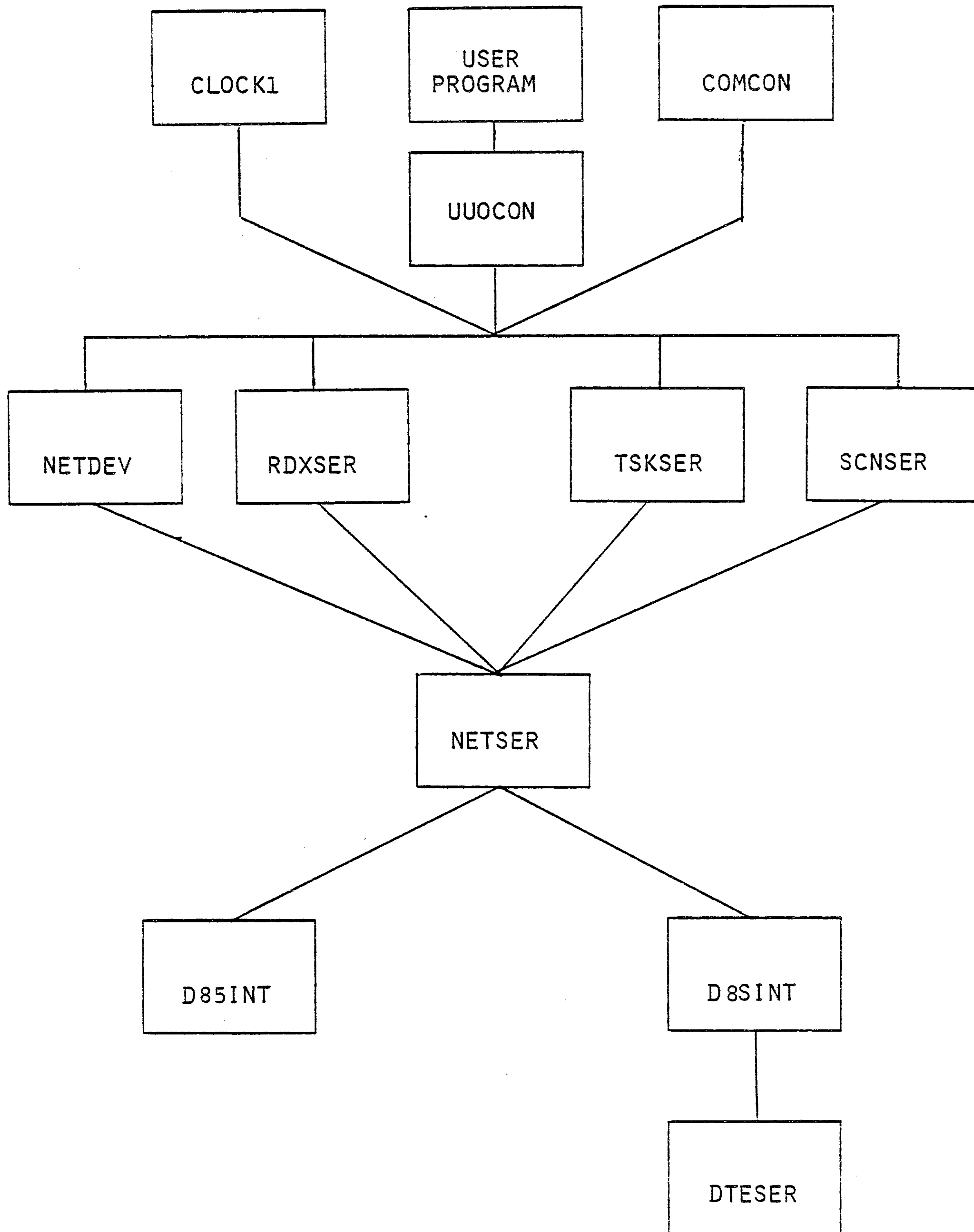
MAG TAPE I/O MODULE ARCHITECTURE



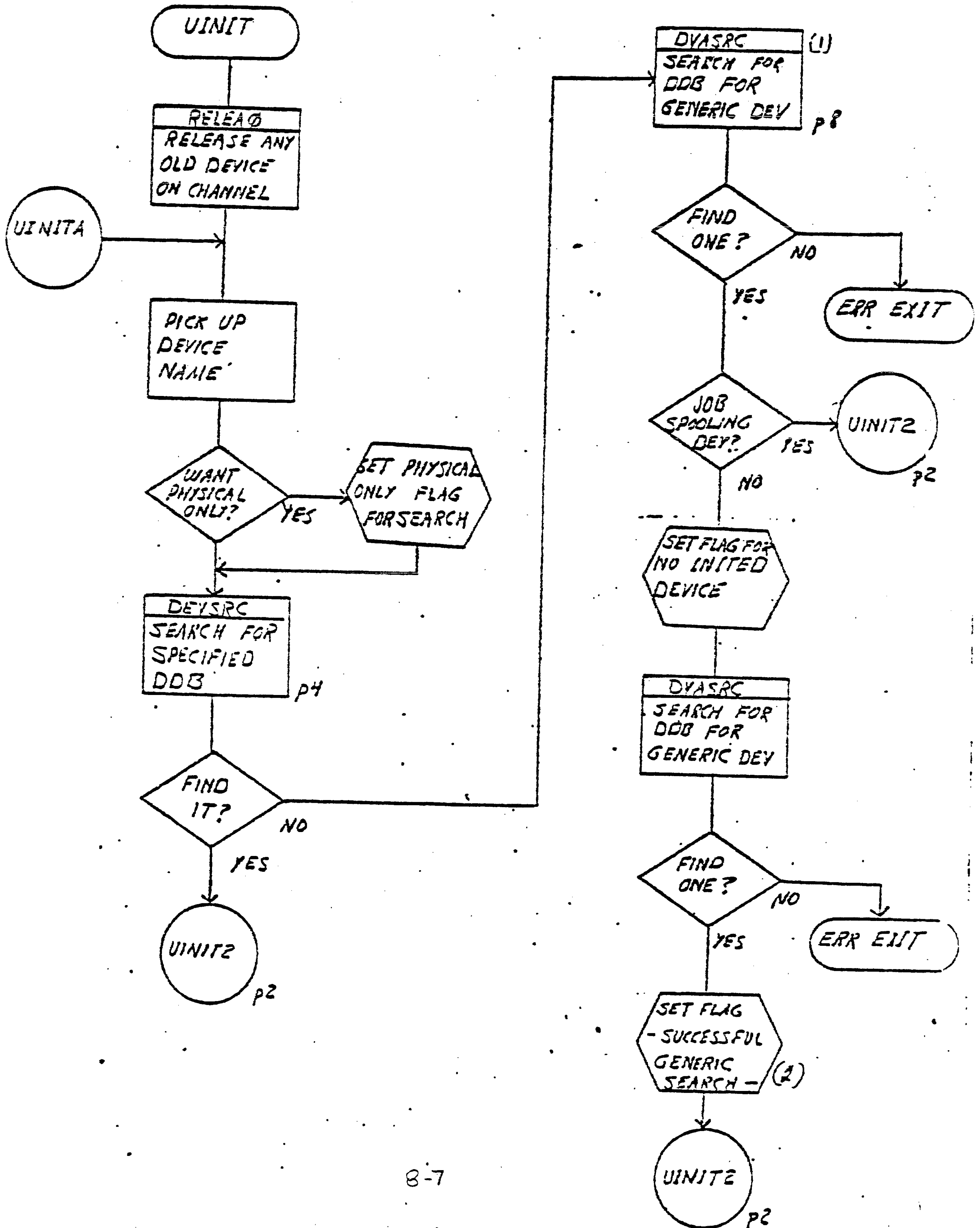
TTY I/O MODULE ARCHITECTURE



ANF-10 I/O MODULE ARCHITECTURE



INIT UUO



UINITZ

HERE WHEN WE HAVE FOUND A USEABLE DDB

CHKMOD
CHECK FOR LEGAL DATA MODE

STACHK
CHECK FOR STATION IN CONTACT

DEVICE NAME = DSK?
YES
NO

IS IT THE SYSTEM DEVICE? (3)
NO
YES

STWAIT
WAIT FOR SYS TAPE RESOURCE

UINITI
DEVICE A DTA?
NO
YES

INCREMENT INIT COUNT

UINITB
SET FLAG FOR ASSIGN-BY-PROGRAM

ASSASG (4)
TRY TO ASSIGN DEV TO JOB
p11

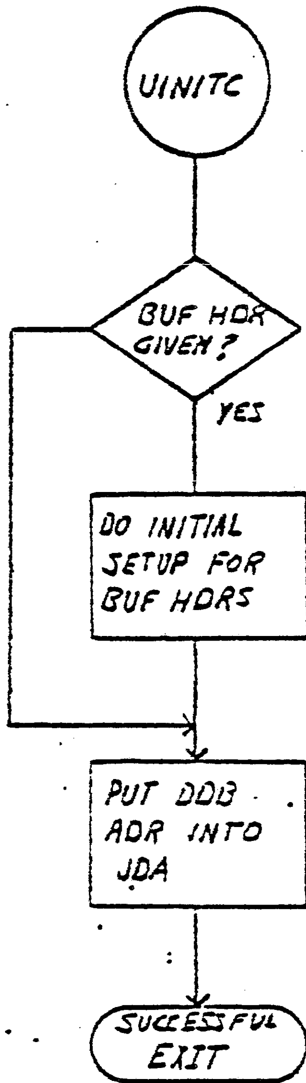
GET IT?
NO
UINITG
p3
YES

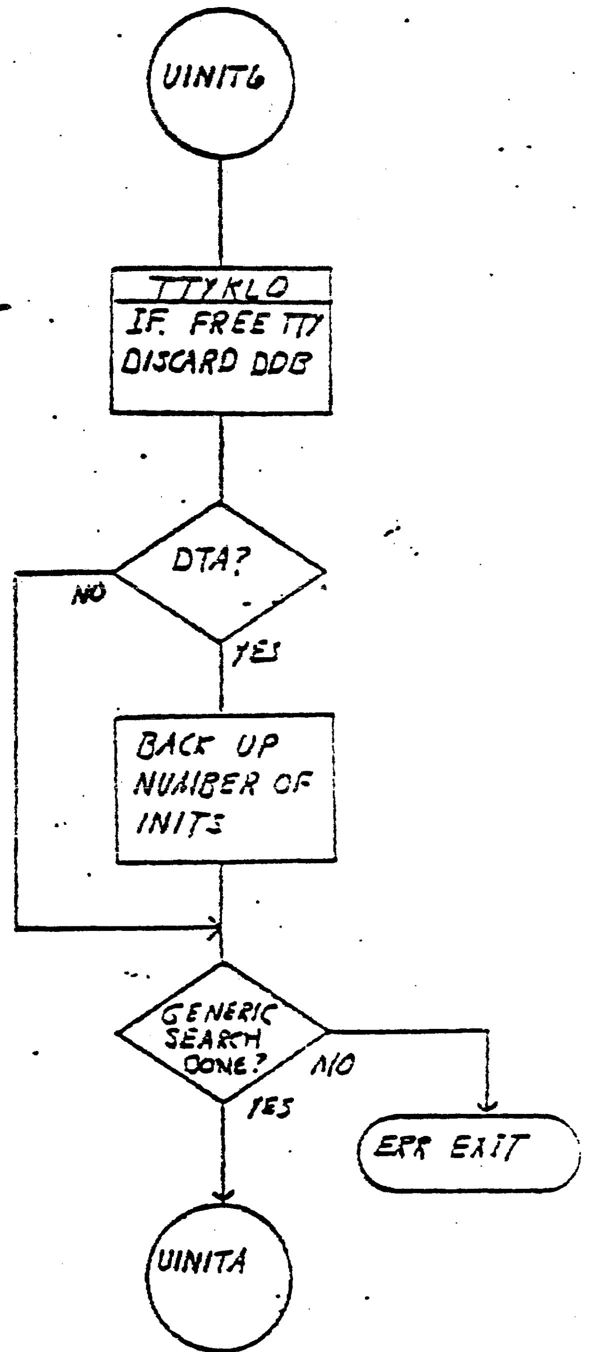
DSZ
SET UP BUFFER SIZE

SETIOS
WAIT FOR DEV TO BE INACTIVE - SET UP DEV STATUS WORD

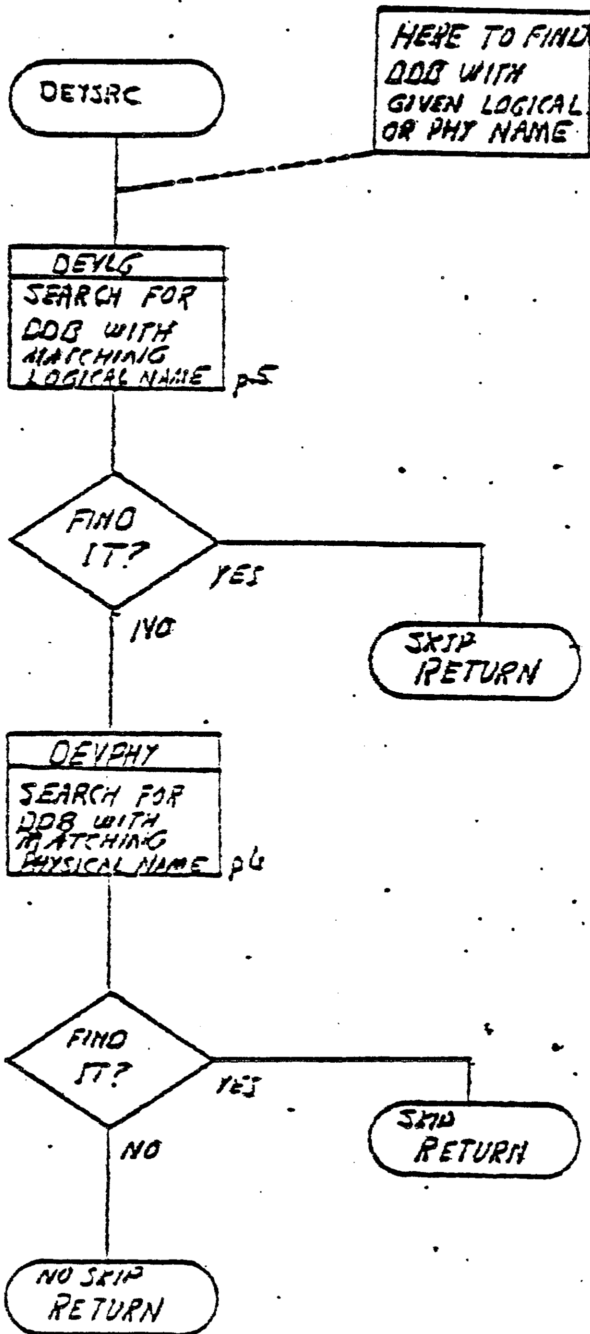
UINITL
UPDATE USER'S HIGHEST CHAN IN USE

UINITC
p3


UINITC
BUF HDR GIVEN?
YES
DO INITIAL SETUP FOR BUF HDRS
PUT DDB ADR INTO JDA
SUCCESSFUL EXIT


UINITG
TTYKLO
IF FREE TTY DISCARD DDB
DTA?
NO
YES
BACK UP NUMBER OF INITS
GENERIC SEARCH DONE?
NO
YES
ERR EXIT
UINITA

DEVSRC



NOTES ON INIT

1. DVASRC -- Generic Device Search

On the first generic device search, we are only trying to verify the existence of a device of the specified type at an appropriate station. If the user is spooling the device, this is all we need in order to let the INIT succeed. If he is not spooling, we must find a device which is available to him. This is the purpose of the second call to DVASRC.

On the generic search we look first at the user's own station. If no such device exists at his station, we look at the central station. We try to find a device ASSIGNed to this user, but not INITed. If that fails we attempt to find any free device of the correct generic type at the correct station.

There are four possible outcomes:

1. Find a device ASSIGNed to this user but not INITed
2. Find a free device
3. Device exists, but not available
4. Device does not exist

Note that if the device exists at the user's station but is unavailable we get result 3. However, if the user is at a remote station and the device does not exist at his station, we look at the central station.

2. If the device should be unavailable at the time we try to assign it, this flag says we should come back and look for another. (Normally will not happen.)

3. This is relevant only to nondisk systems.

4. This routine is used by both the INIT UUO and the ASSIGN command.

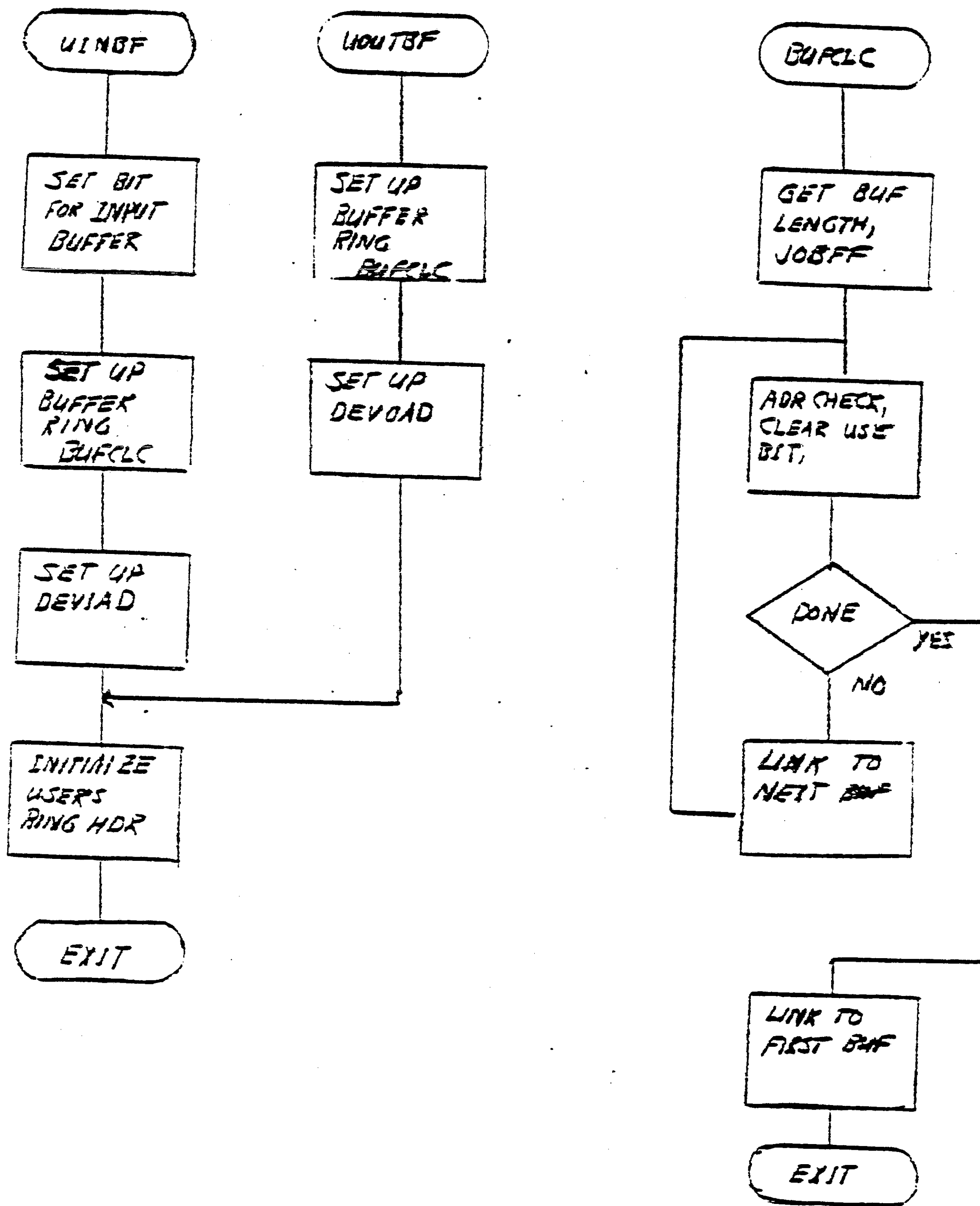
5. This flag is used when we must distinguish the real system device from a device assigned logical name SYS.

6. The device name TTY always means the job's controlling TTY.

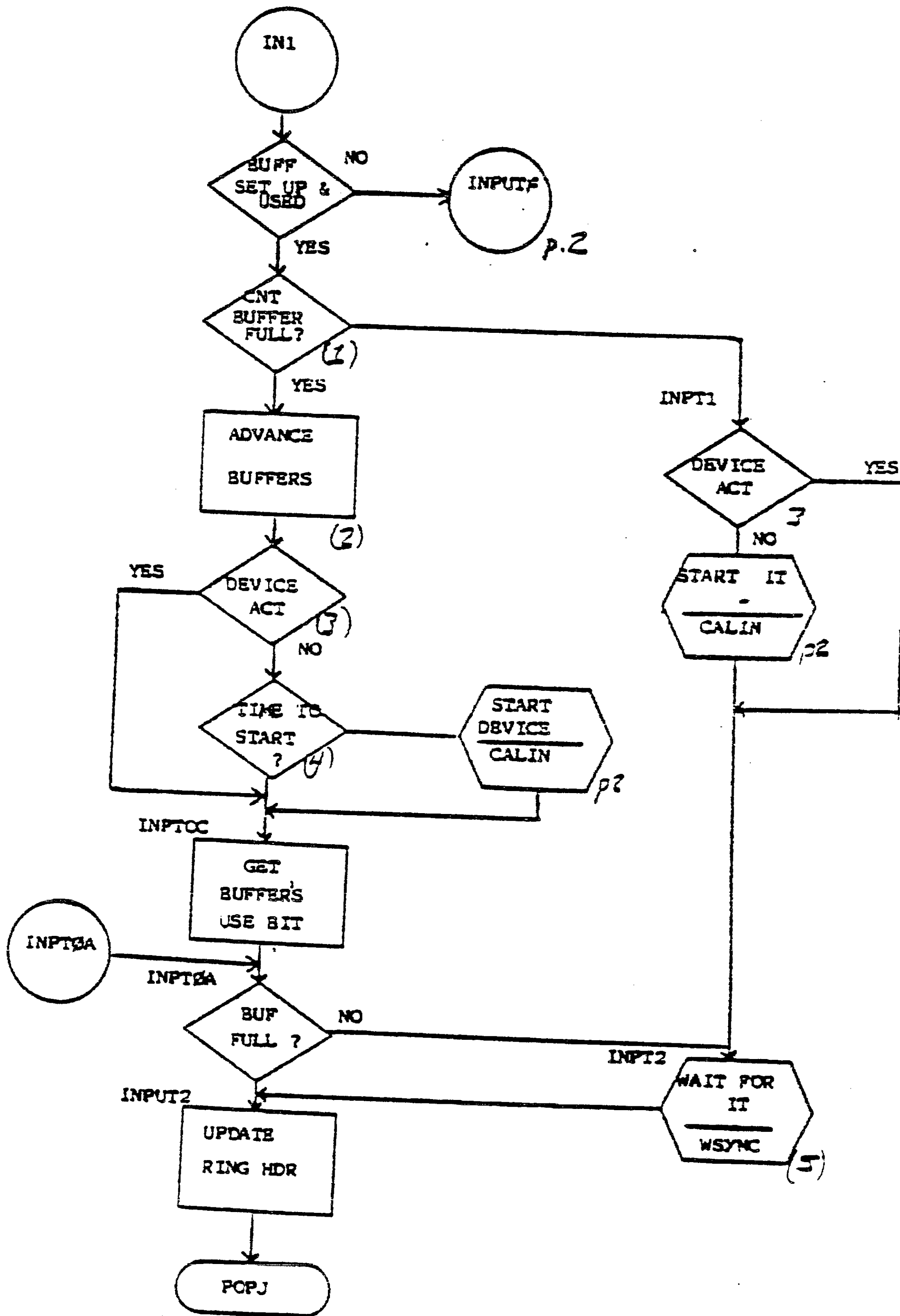
7. e.g., LPTS1

8. Unless we found a DDB that was ASSIGNed but not INITed by the user, we will set up a new DDB by copying the prototype disk DDB. We copy DEVNAM from the DDB which we found. If we found this DDB on a logical device search, DEVNAM will match the physical device name specified on the ASSIGN command which set up the logical name. Otherwise, DEVNAM will match the argument of the INIT.

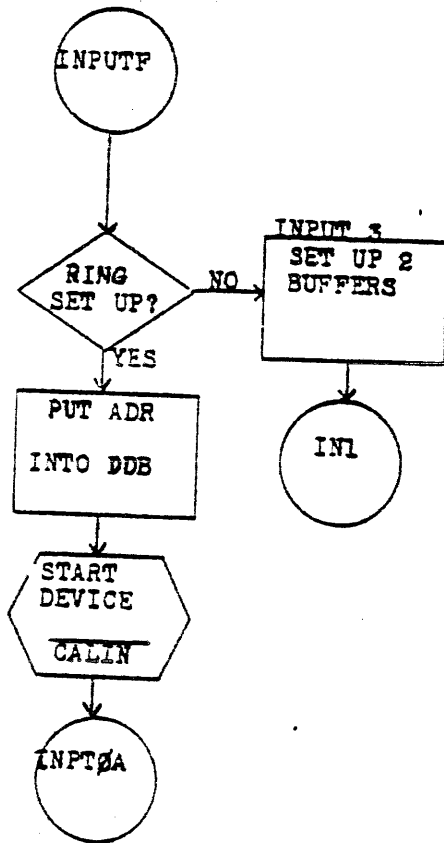
INBUF, OUTBUF

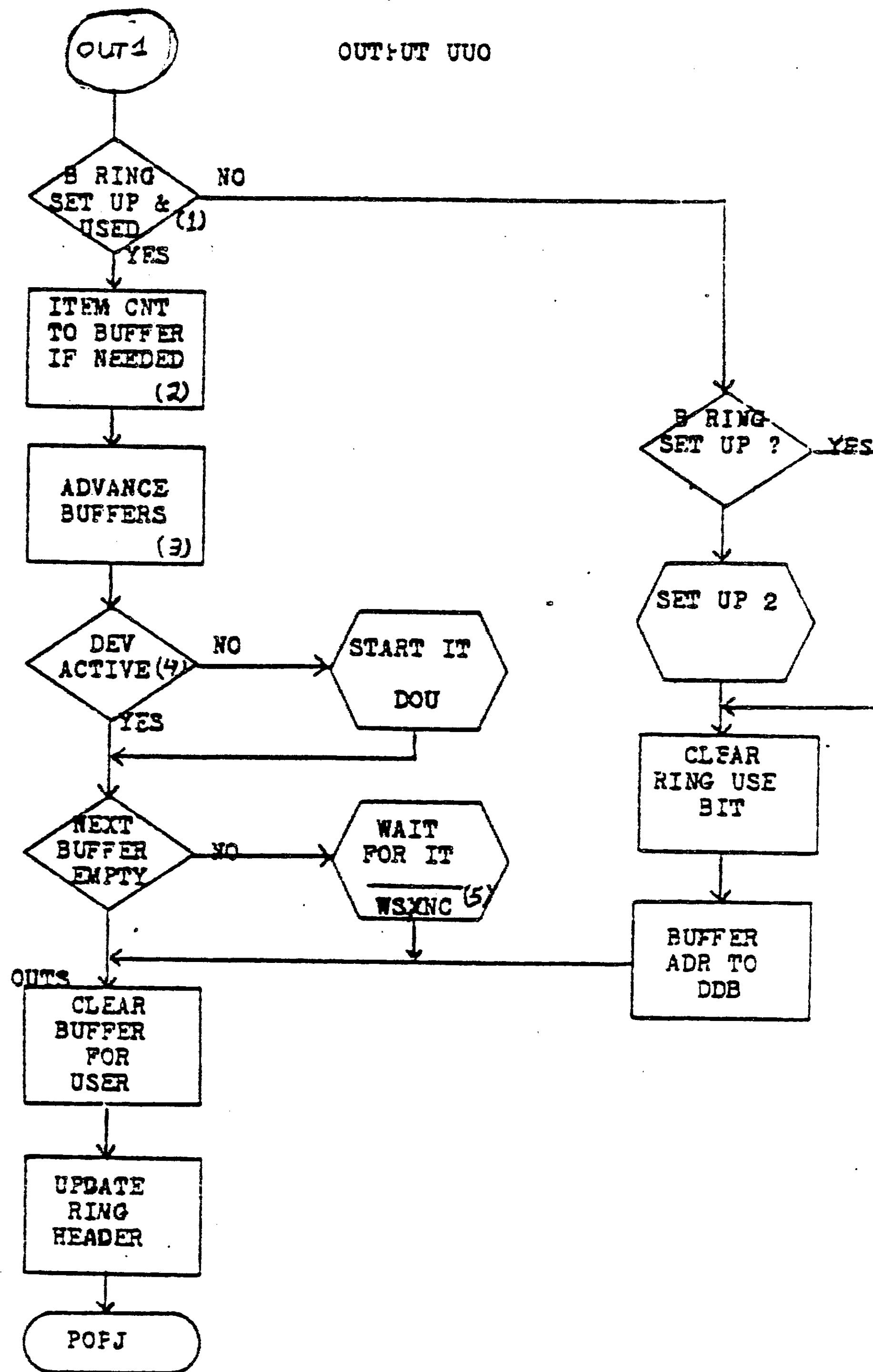


INPUT UUO

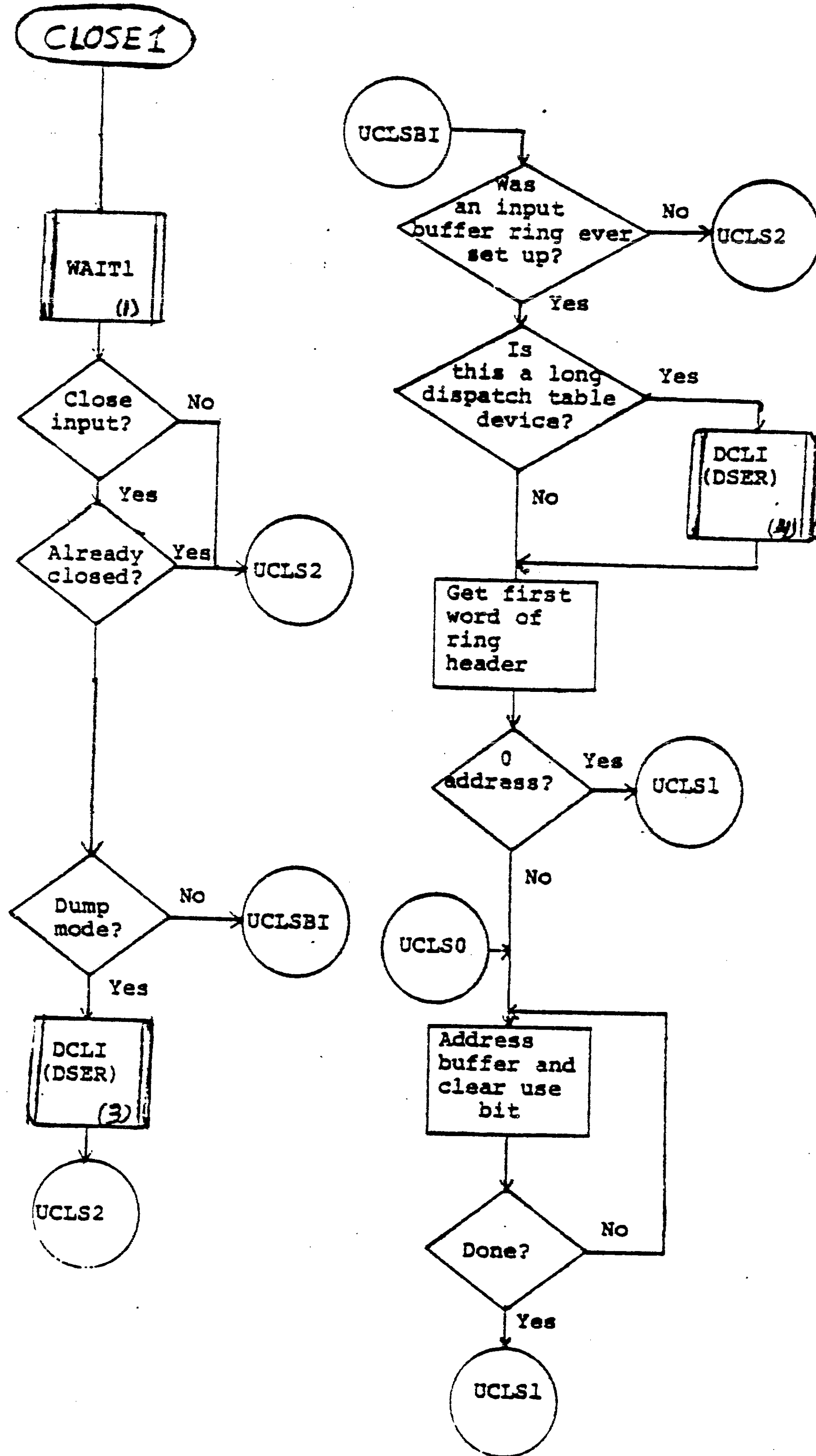


INPUT p.2

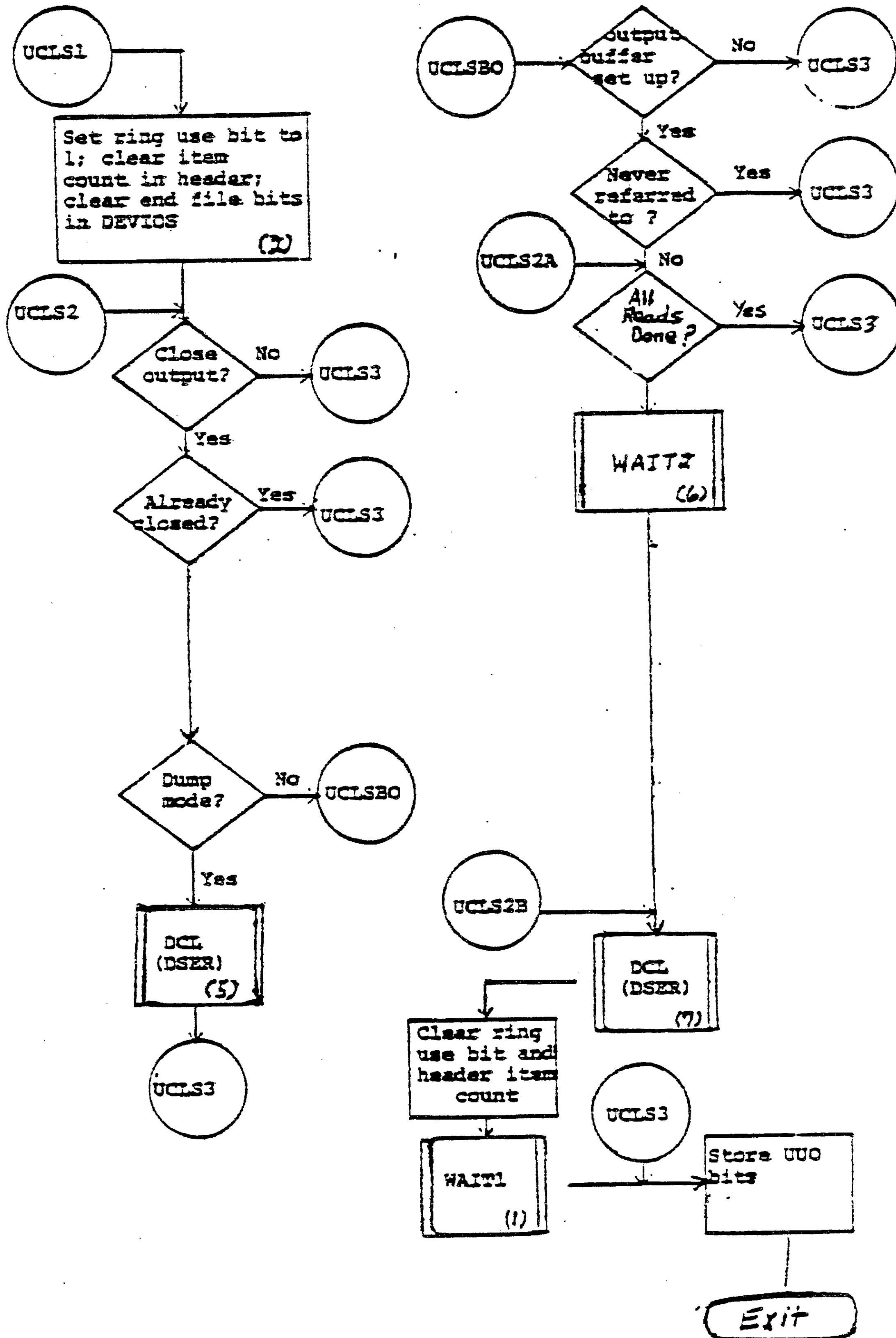



OUT1
OUTPUT UUO
B RING SET UP & USED (1)
NO
YES
ITEM CNT TO BUFFER IF NEEDED (2)
ADVANCE BUFFERS (3)
DEV ACTIVE (4)
NO
START IT
DOU
YES
NEXT BUFFER EMPTY
NO
WAIT FOR IT
WSYNC (5)
OUTS
CLEAR BUFFER FOR USER
UPDATE RING HEADER
POPJ
B RING SET UP ?
YES
SET UP 2
CLEAR RING USE BIT
BUFFER ADR TO DDB

CLOSE p.1



CLOSE p2



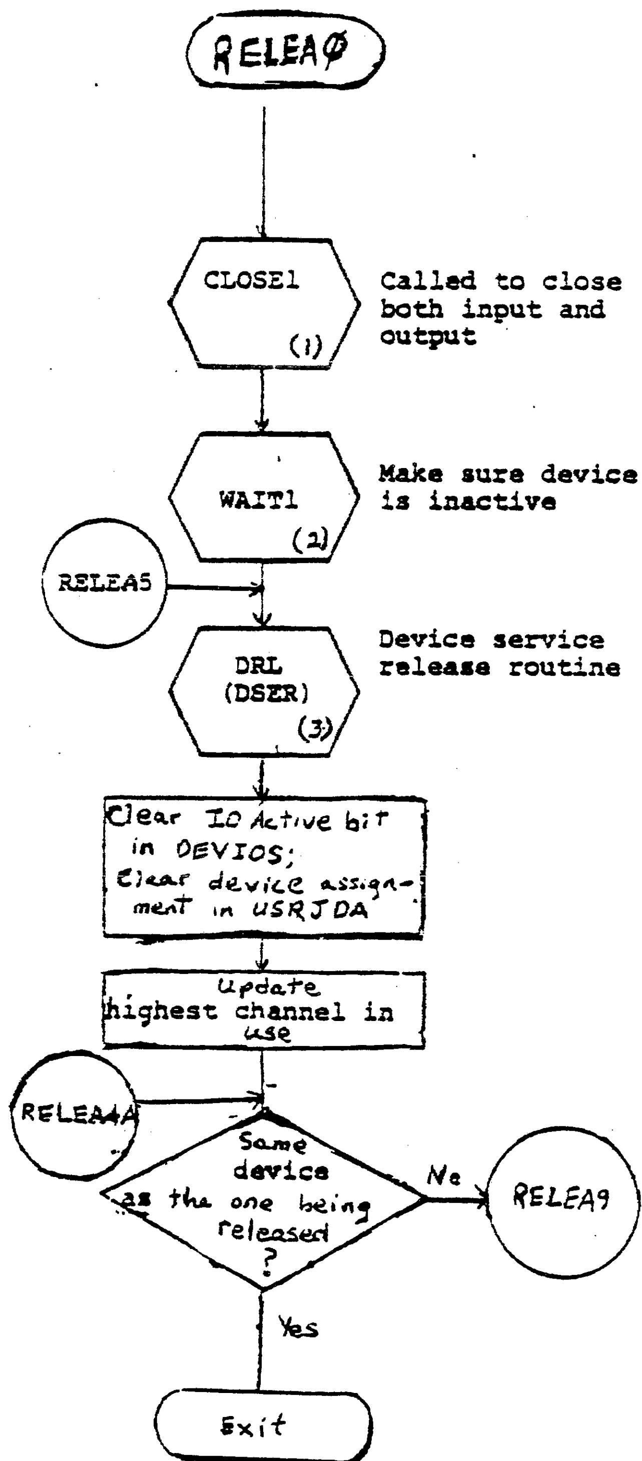
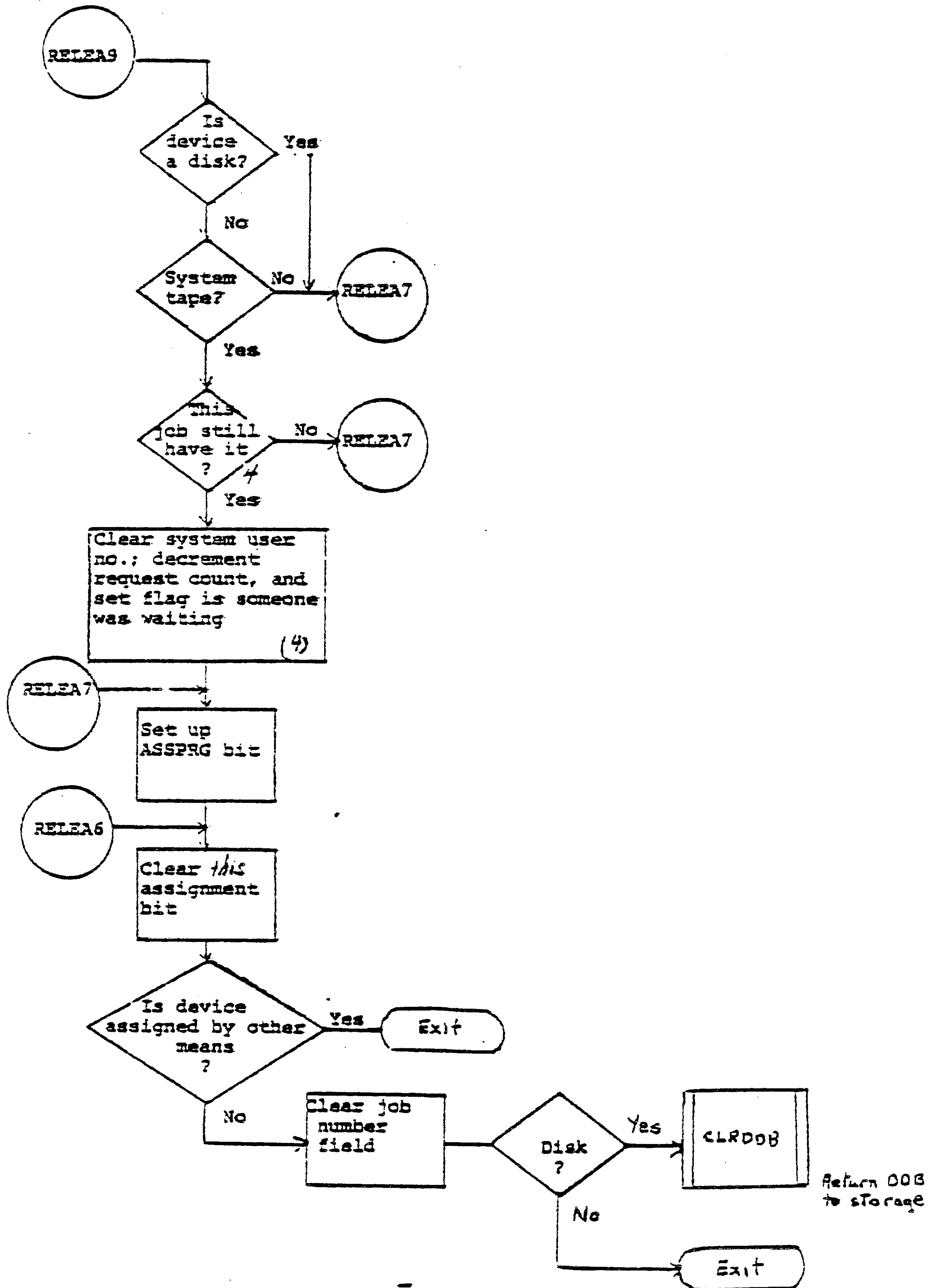


Figure 8. Flow Chart of RELEASE Operator

RELEASE p.2



NOTES ON INPUT

1. This will always be true unless the user is changing the structure of the buffer ring.

2. Mark the user's current buffer as now available to the device interrupt routine.

3. Check IOACT in DEVIOS.

4. Except for TTY, this is a check if the buffer ahead of the buffer we are about to give to the user is empty. Hence, for a N buffer ring, we start the device when N-1 buffers are empty.

   For TTY we check the same buffer which we are giving to the user. The TTY device dependent routine does not actually "start the device," but copies characters from the monitor TTY buffer to the user's buffer. See SCNSER flows for details.

5. WSYNC sets the job's wait state code to IO Wait and calls WSCHED. The job is stopped at this point and its stored PC will say to restart it after the PUSHJ to WSYNC. The interrupt routine must get the job out of IO Wait when the next buffer is full. WSYNC will give an immediate return if IOACT is not set. This allows us to give the job an "error" return on end of file.

NOTES ON OUTPUT UUO

1. Normally the user's first OUTPUT UUO will take the NO branch. Its only function then is to set up the buffer ring and initialize the buffer control block.

2. Unless the user set the IOWC bit, we compute the buffer word count by looking at the byte pointer in the ring header.

3. Mark the current buffer as available to be written out.

4. Check IOACT

5. See WSYNC note for INPUT

Notes on CLOSE

1. WAIT1 will repeatedly call WSYNC until the device is no longer active. Hence, it holds the job in IO Wait until all buffers have been released by the interrupt routine.

2. Hence, after CLOSE it will appear that the ring has been set up but not used.

3. Device dependent routine for dump mode input close.

4. Device dependent routine for buffered mode input close.

5. Device dependent routine for dump mode output close.

6. Ensures that all buffers are written.

7. Device dependent routine for buffered mode output close.

Notes on Release

1. Hence, RELEASE implies a CLOSE for the same channel.

2. This will normally give an immediate return, since CLOSE1 also called WAIT1.

3. Device dependent routine for RELEASE.

4. This applies only to non-disk systems.

INTERRUPT ROUTINE CHAIN

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


CH'N:           Ø
                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
```

8.17 NON-STANDARD DEVICE PI ASSIGNMENT

Under ordinary circumstances when COMMON is assembled, devices are assigned to PI channels according to their group priority. (Refer to Table 8-1.) If you have at your installation a device not listed as a standard device in Table 8-1 and you have written your own Monitor Device Service Routine, you must specify the device mnemonic (in 3 characters or less) and designate an appropriate priority interrupt channel. You must answer all three questions as they apply to your configuration. The first question

TYPE "DEVICE-MNEMONIC,PI-CHANNEL" FOR SPECIAL DEVICES

requests special device service routines that do not need either a Channel Save Routine or a Device Data Block. The second question

TYPE "DEVICE-MNEMONIC,PI-CHANNEL, NO.-OF-DEVICES"

requests devices with special service routines that have a Device Data Block but no Channel Save Routine. The third question

TYPE "DEVICE-MNEMONIC,PI-CHANNEL, HIGHEST-AC-TO-SAVE"

requests devices with special service routines that have a Channel Save Routine, but no Device Data Block.

Special devices that you added during the HDWGEN dialogue are chained to the requested channel. To give a device the exclusive use of a channel, you respond to the "symbol,value" question with

UNIQn,1

where n is the priority interrupt channel to be reserved. (Refer to the UNIQn,1 entry in Section 8.14.1.)

One or more priority interrupt channels may be reserved for real-time devices with the RTTRP monitor call. These devices are completely controlled by user programs and have no specific code loaded with the monitor. To reserve a priority interrupt channel for use with RTTRP, you should respond to the "symbol,value" question with

RTCHn,1

where n is the priority interrupt channel to be reserved.

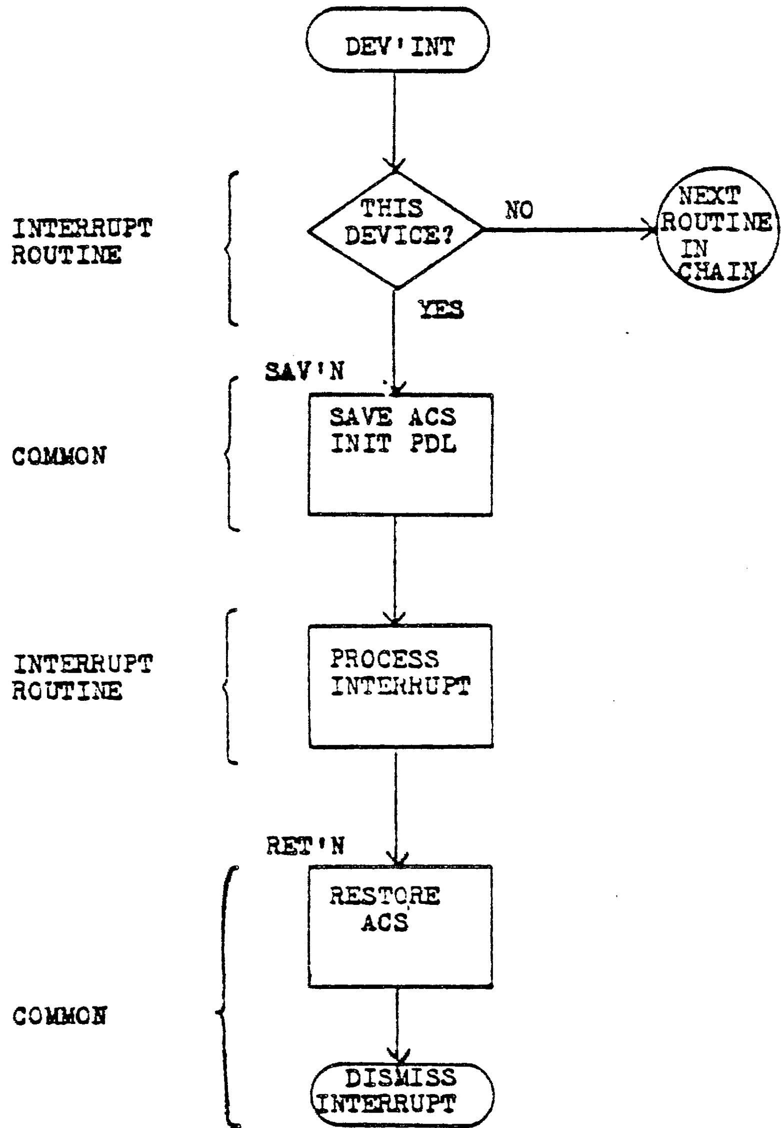
(Refer to the RTCHn,1 entry in Section 8.14.1 and to the DECsystem-10 Monitor Calls manual.)

I/O devices are grouped by their relative interrupt speeds. If any device of a particular group is present, a PI channel is assigned to that device according to its group priority. Group priorities for standard devices may be revised by rearranging the devices in INTTAB, which is in the COMMON source file.

DEVICE GROUPS FOR PI CHANNEL ASSIGNMENT

DEVICE NMEMONIC	GROUP	NAME
MTA, MTB	A	TM10A MAGTAPE DATA CHANNEL
DTA,DTB	B	TD10 DEC TAPE
RTC	C	DK10 REAL TIME CLOCK
CDP		CARD PUNCH
CDR		CARD READER DATA
APR		KI ARITHMETIC PROCESSOR
SCN	D	TERMINAL SCANNER
DLØ, DL1		DL10 PDP11 DMA INTERFACE
CCØ, CC1		680I COMMUNICATIONS
PTR		PAPER TAPE READER
CDR		CARD READER FLAGS
LPT		LINE PRINTER
DLP		RSX20 LINE PRINTER
DTA, DTB		DEC TAPE FLAG CHANNEL
MTA, MTB		MAGTAPE FLAG CHANNEL
CTY		CONSOLE TTY
DTE		DTE PRIMARY/SECONDARY PROTOCOL
DLX		IBM INTERFACE
NET		REMOTE DEVICES
DSK	E	DISK DEVICES
XTC		DA28 PDP11 DMA INTERFACE
PEN		LIGHT PEN
PTP		PAPER TAPE PUNCH
CDP		CARD PUNCH FLAG
PLT		PLOTTER
DIS	F	DISPLAY
NET	G	NETWORK SOFTWARE
CLK	H	SCHEDULER CLOCK ROUTINES (ALWAYS ASSIGNED TO PI CHANNEL 7)

DEVICE INTERRUPT ROUTINE



ADVBFF

PICK UP DEVIAD

SVEUB p2
PRESERVE BARS
& SET UP UBR SO CURRENT USER IS ADDRESSABLE

DEVICE HAVE EVM?

NO

YES

ADVBF1

DUMP S INTO BUFFER

DUMP S INTO BUFFER

SET USE BIT & PICK UP LINK

SET USE BIT & PICK UP LINK

BADRCK
ADR CHECK NEXT BUF

BADRCK
ADR CHECK NEXT BUF

ADVEVM p2
SET EPMP FOR NEXT BUF

SET NEW DEVIAD

OK?

NO

ADVBU2

YES

NEXT BUFFER FREE?

NO

ADVBU2

YES

SET NEW DEVIAD (GETS EXEC VIRT ADDR OF NEXT BUFFER)

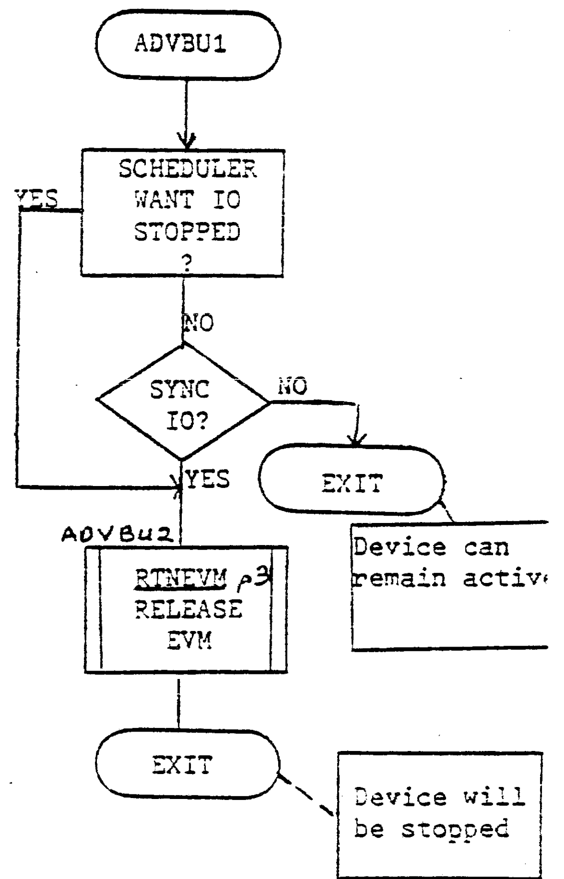
NEXT BUFFER FREE?

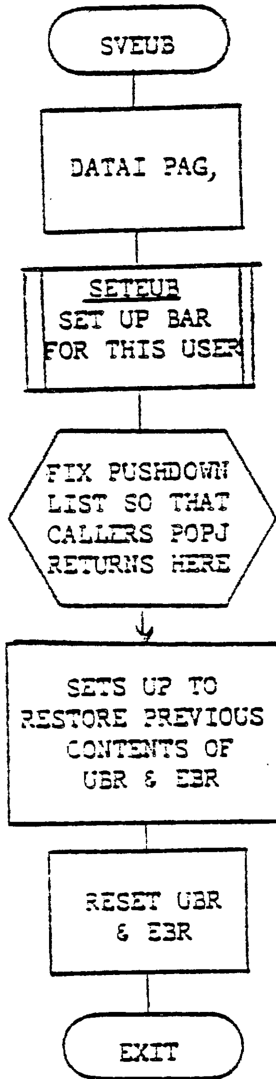
YES

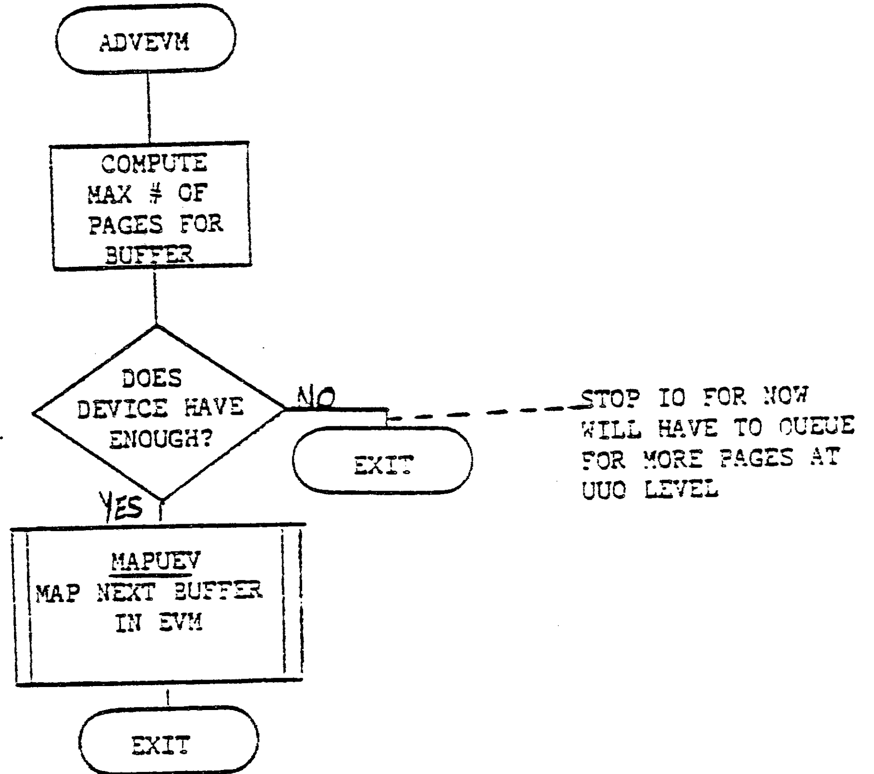
ADVBU1

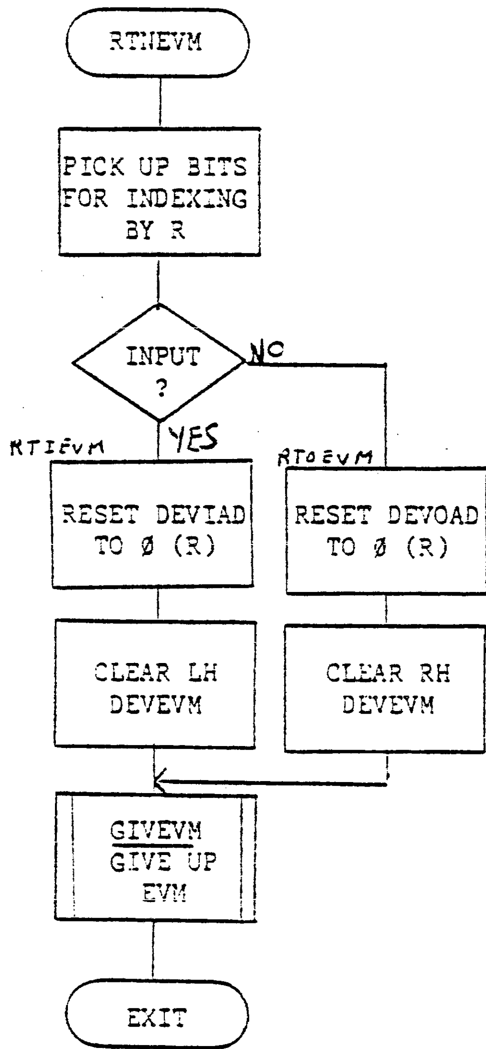
NO

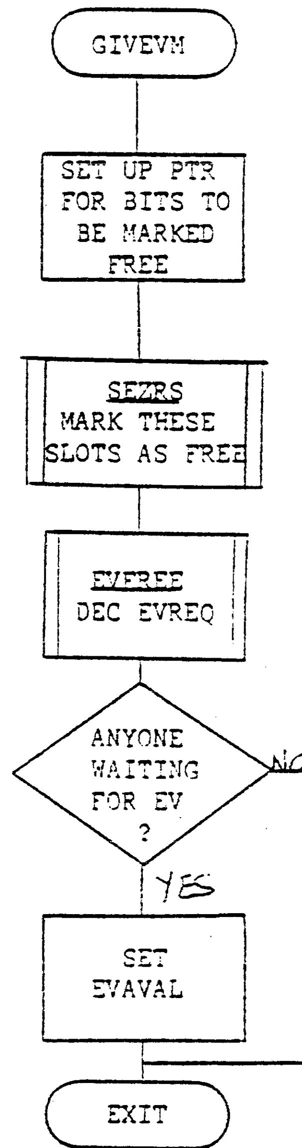
ADVBU2

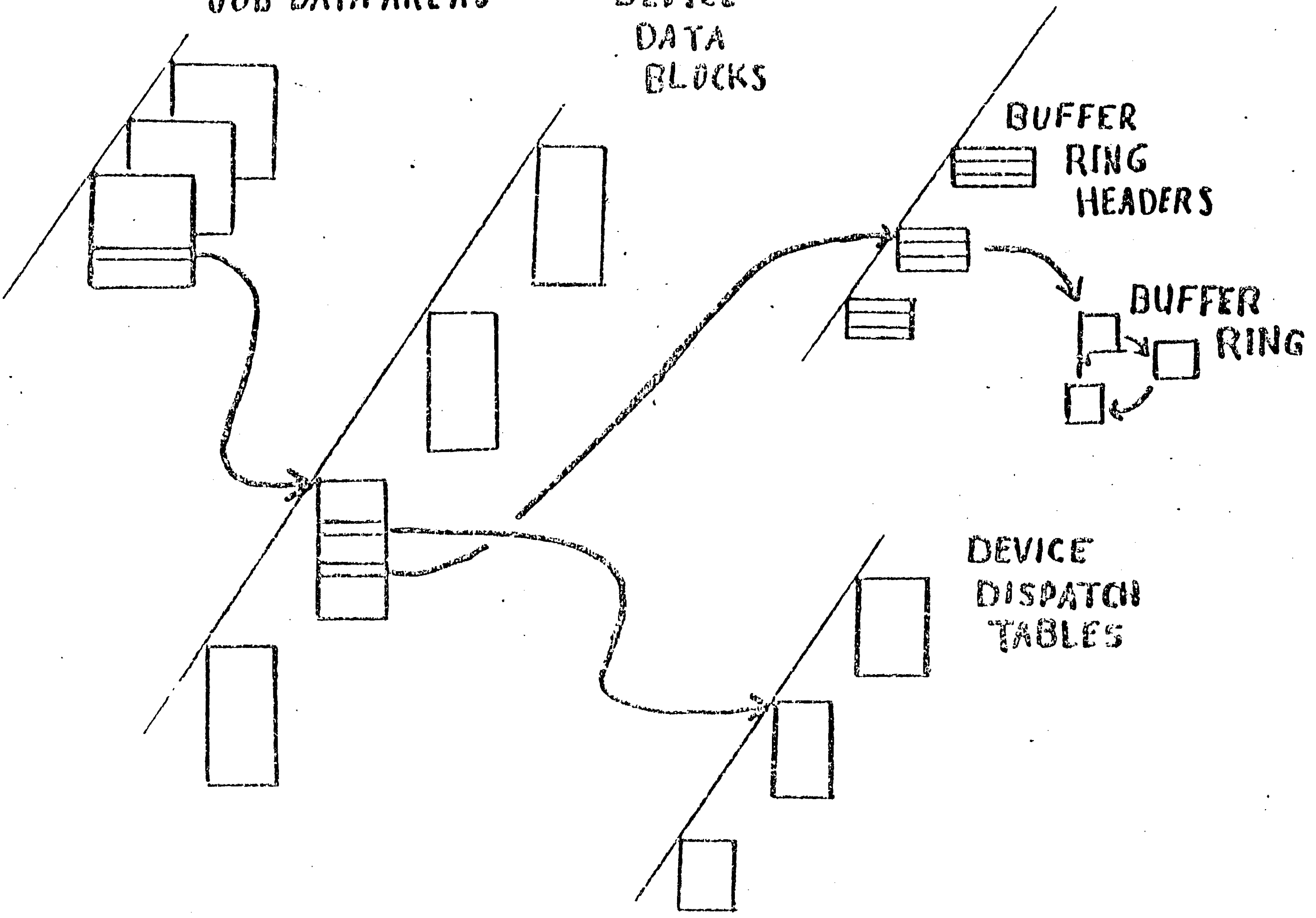



SVEUB
DATAI PAG,
SETEUB
SET UP BAR
FOR THIS USER
FIX PUSHDOWN
LIST SO THAT
CALLERS POPJ
RETURNS HERE
SETS UP TO
RESTORE PREVIOUS
CONTENTS OF
UBR & EBR
RESET UBR
& EBR
EXIT


ADVEVM
COMPUTE
MAX # OF
PAGES FOR
BUFFER
DOES
DEVICE HAVE
ENOUGH?
NO
EXIT
STOP IO FOR NOW
WILL HAVE TO QUEUE
FOR MORE PAGES AT
UUO LEVEL
YES
MAPUEV
MAP NEXT BUFFER
IN EVM
EXIT

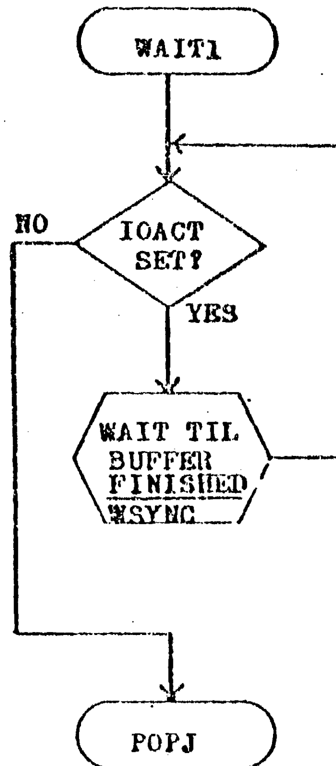

RTNEVM
PICK UP BITS FOR INDEXING BY R
INPUT ?
NO
YES
RTIEVM
RTOEVM
RESET DEVIAD TO Ø (R)
RESET DEVOAD TO Ø (R)
CLEAR LH DEVEVM
CLEAR RH DEVEVM
GIVEVM
GIVE UP EVM
EXIT

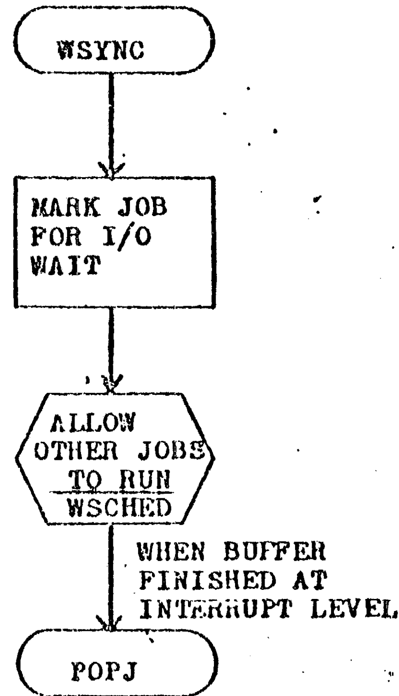

GIVEVM
SET UP PTR FOR BITS TO BE MARKED FREE
SEZRS
MARK THESE SLOTS AS FREE
EVFREE
DEC EVREQ
ANYONE WAITING FOR EV ?
NO
YES
SET EVAVAL
EXIT



JOB DATA AREAS
DEVICE DATA BLOCKS
BUFFER RING HEADERS
BUFFER RING
DEVICE DISPATCH TABLES

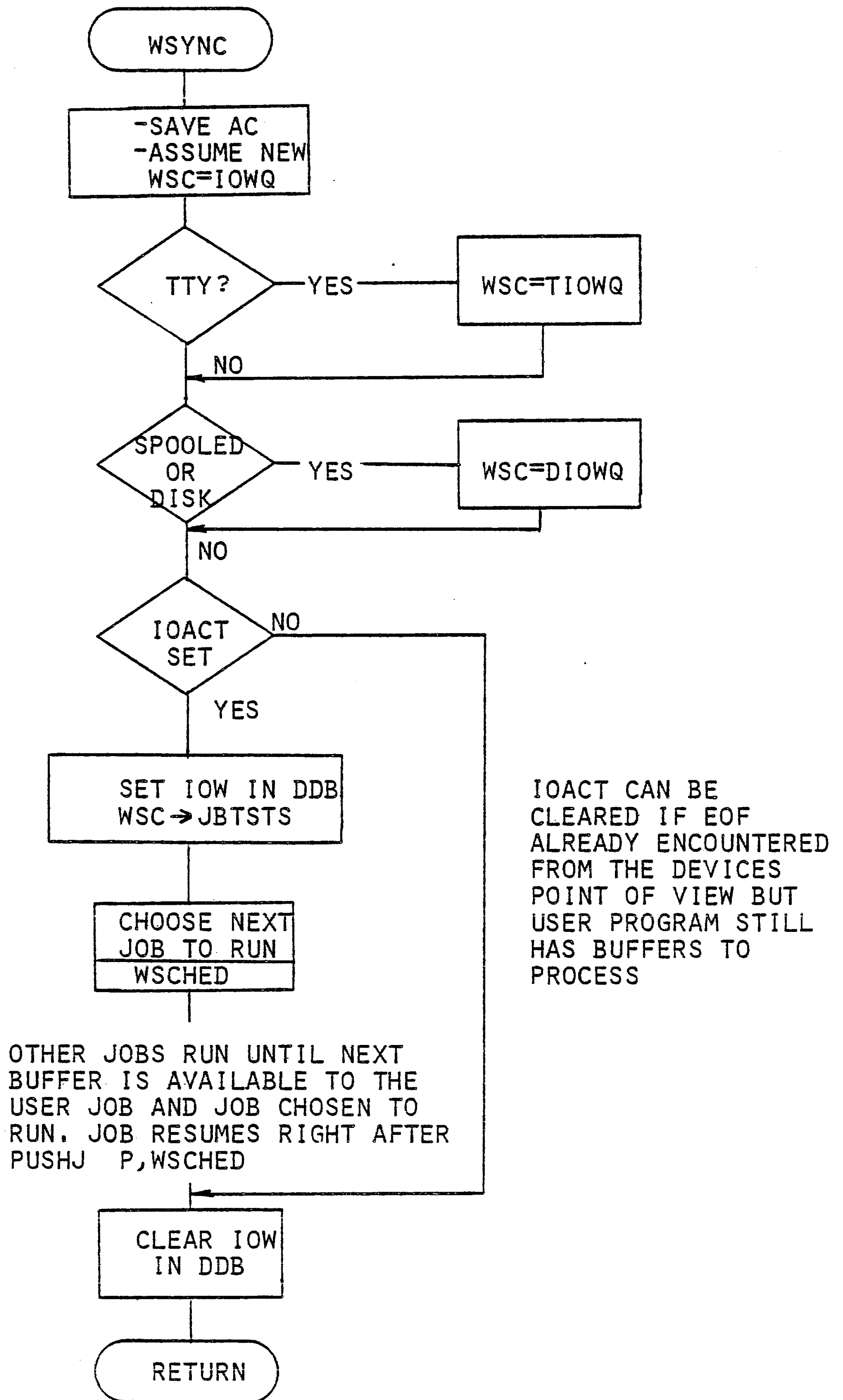
WAIT ROUTINES

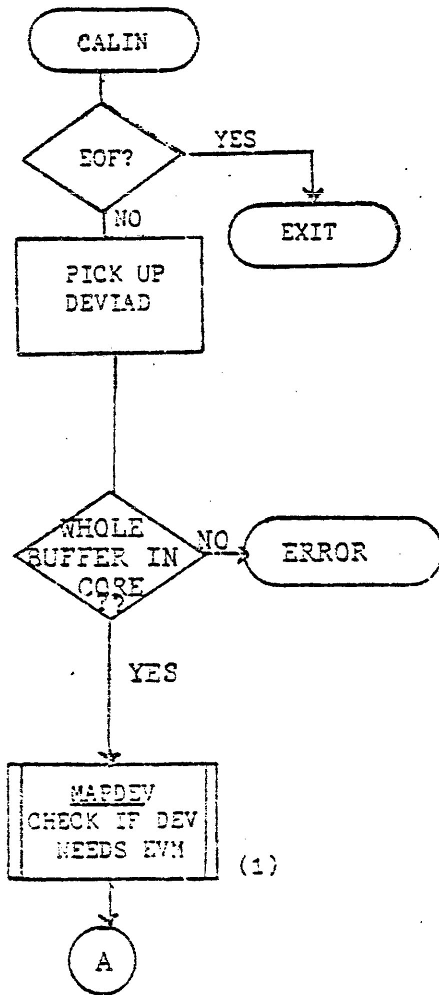


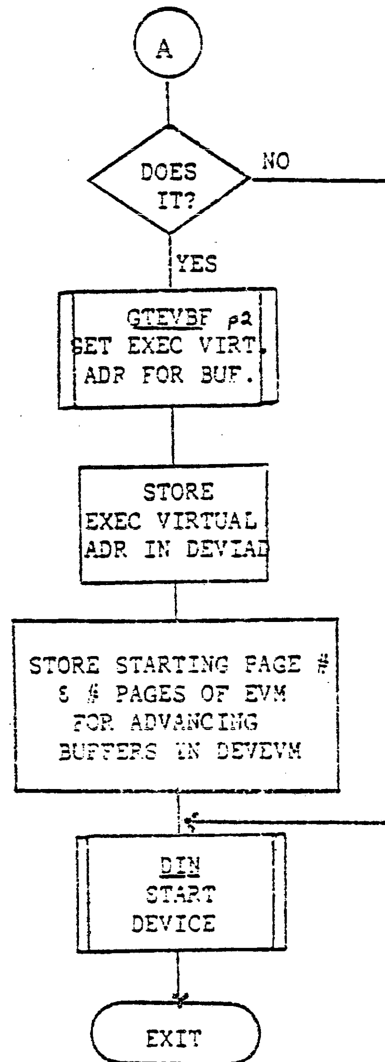


WSYNC ROUTINE IN CLOCK1

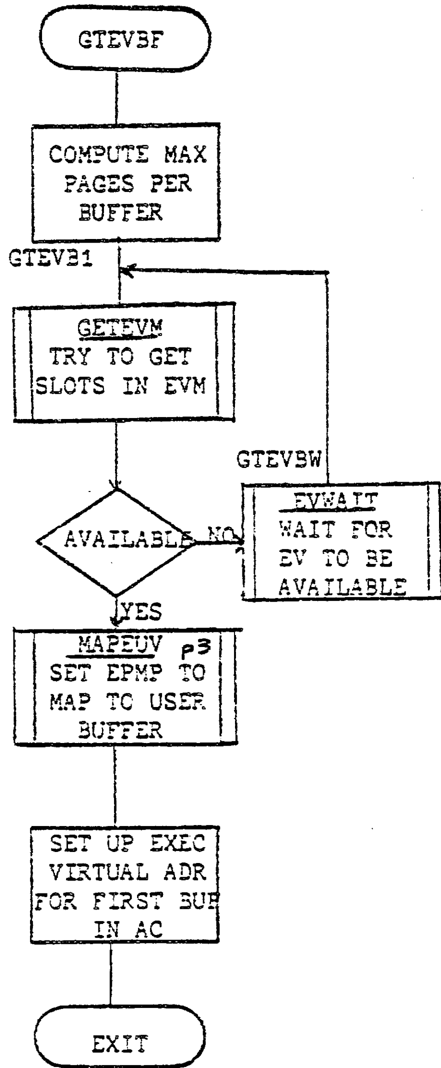
CALLED AT UUO LEVEL TO PUB JOB IN IOW IF THE NEXT BUFFER IS NOT AVAILABLE. ROUTINE SETIOD WILL BE CALLED AT INTERRUPT LEVEL TO UNBLOCK JOB.

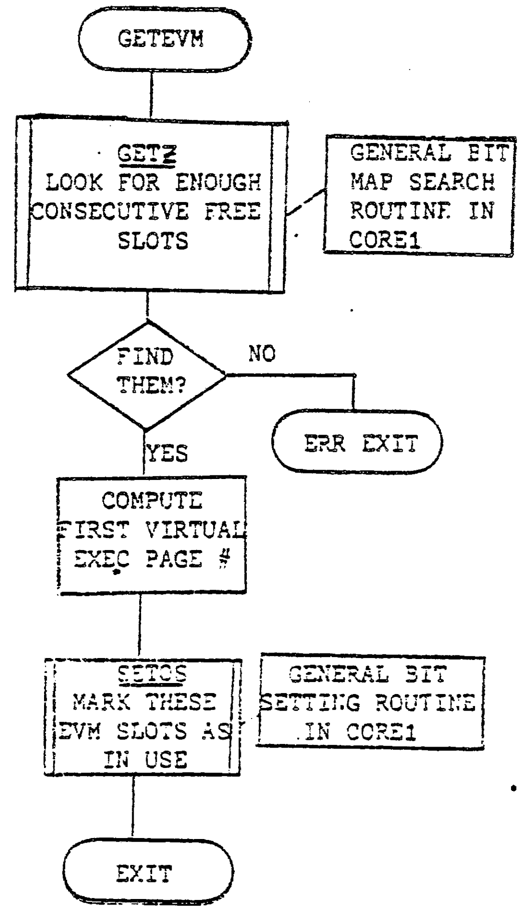




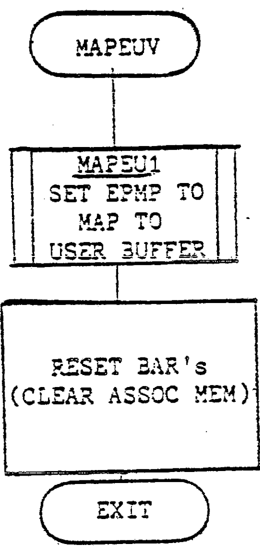


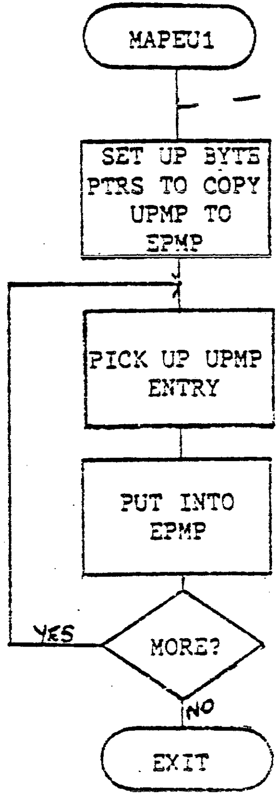
(1) DSK, MTA, TTY, & PTY do not


GTEVBF
COMPUTE MAX PAGES PER BUFFER
GTEVB1
GETEVM
TRY TO GET SLOTS IN EVM
GTEVBW
AVAILABLE
NO
EVWAIT
WAIT FOR EV TO BE AVAILABLE
YES
MAPEUV
p3
SET EPMP TO MAP TO USER BUFFER
SET UP EXEC VIRTUAL ADR FOR FIRST BUF IN AC
EXIT


GETEVM
GETZ
LOOK FOR ENOUGH CONSECUTIVE FREE SLOTS
GENERAL BIT MAP SEARCH ROUTINE IN CORE1
FIND THEM?
NO
ERR EXIT
YES
COMPUTE FIRST VIRTUAL EXEC PAGE #
SETOS
MARK THESE EVM SLOTS AS IN USE
GENERAL BIT SETTING ROUTINE IN CORE1
EXIT

3.



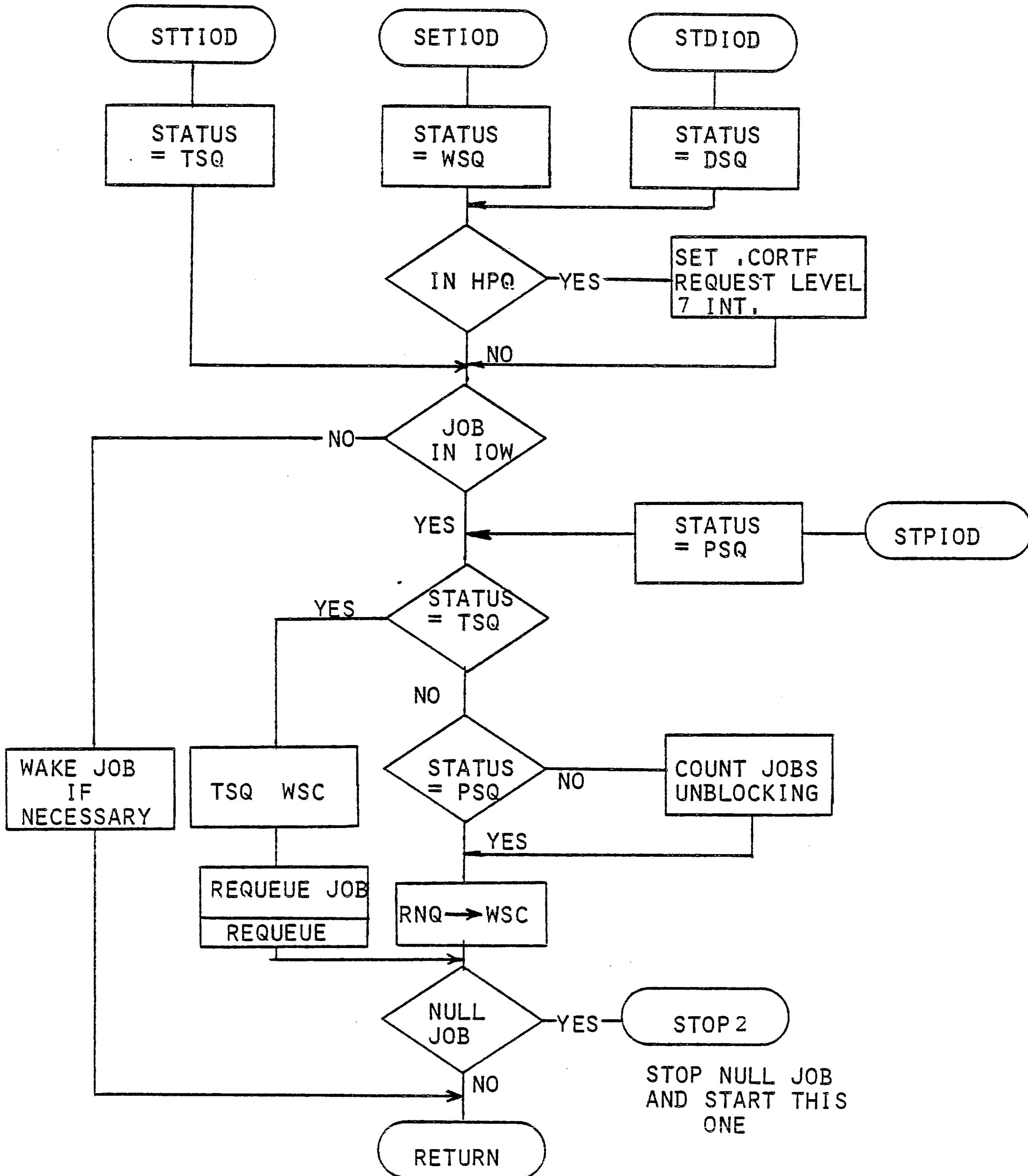


Here directly at
int level when BAR's
will be reset later
anyway

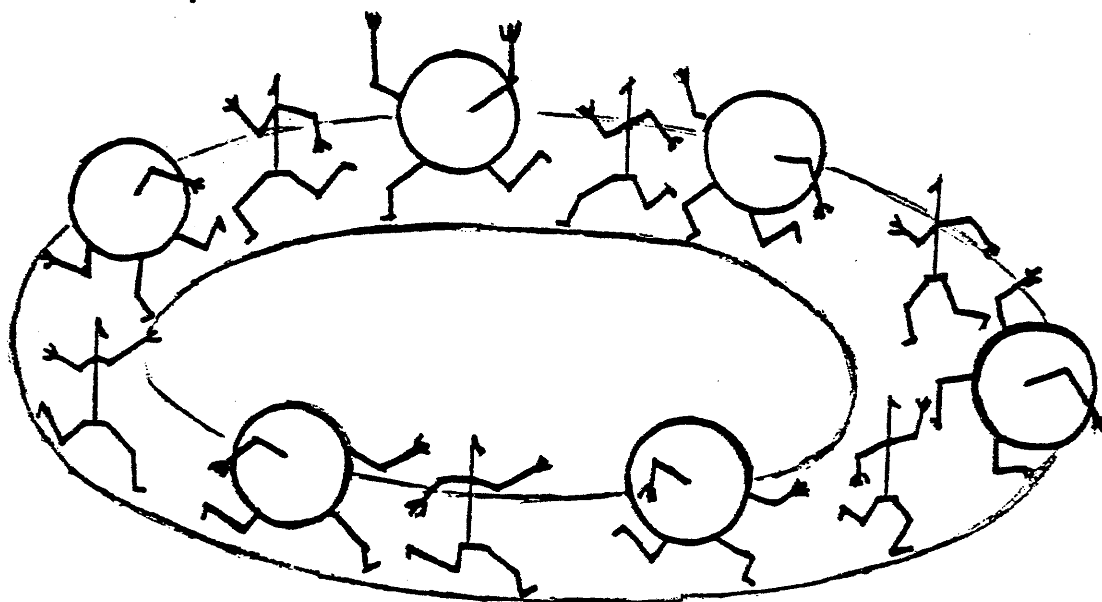
ROUTINE TO UNBLOCK A JOB FROM IO WAIT

ENTRIES:

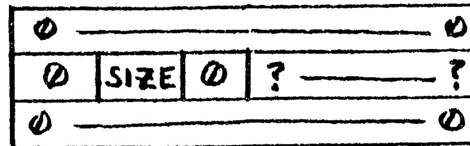
SETIOD	MAIN ENTRY, CALLED FROM DEV'SER
STDIOD	DISK I/O CALLED FROM FILIO
STPIOD	PAGING I/O CALLED FROM SWPSER
STTIOD	TTY I/O CALLED FROM SCNSER



A RACE CONDITION



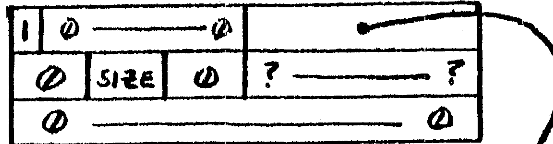
Buffer Ring after an OPEN UUO



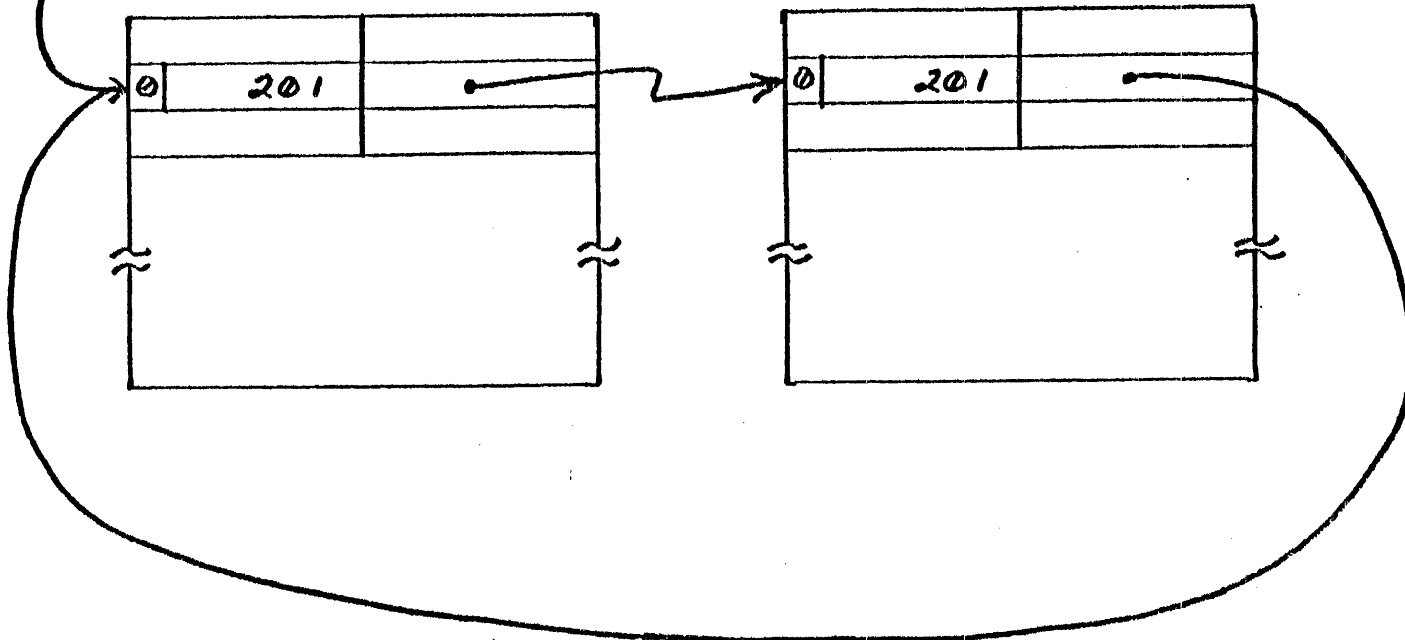
IOACT = Ø
IOW = Ø

No buffers -- only the Buffer Control Block

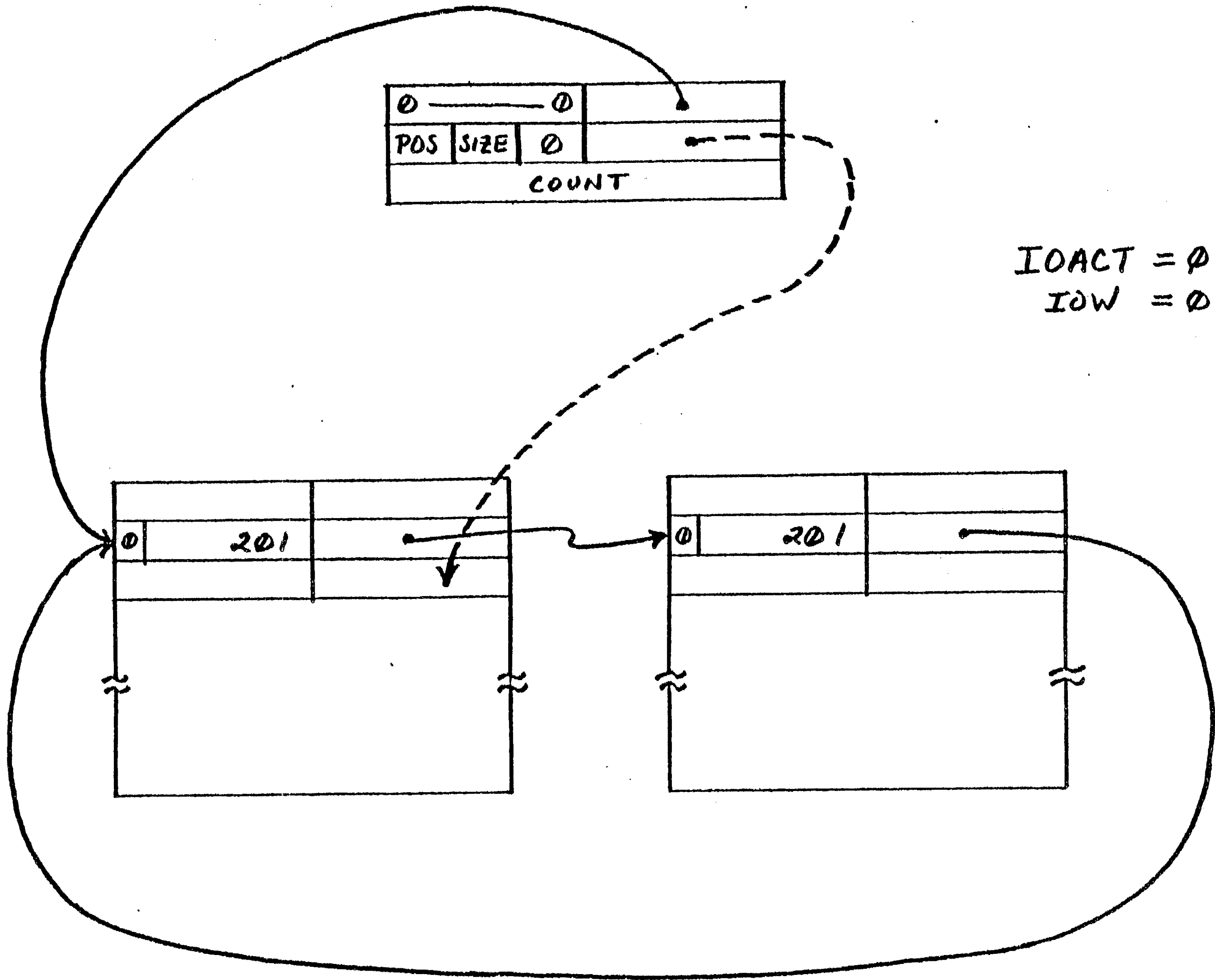
Buffer Ring after an OUTBUF UUO



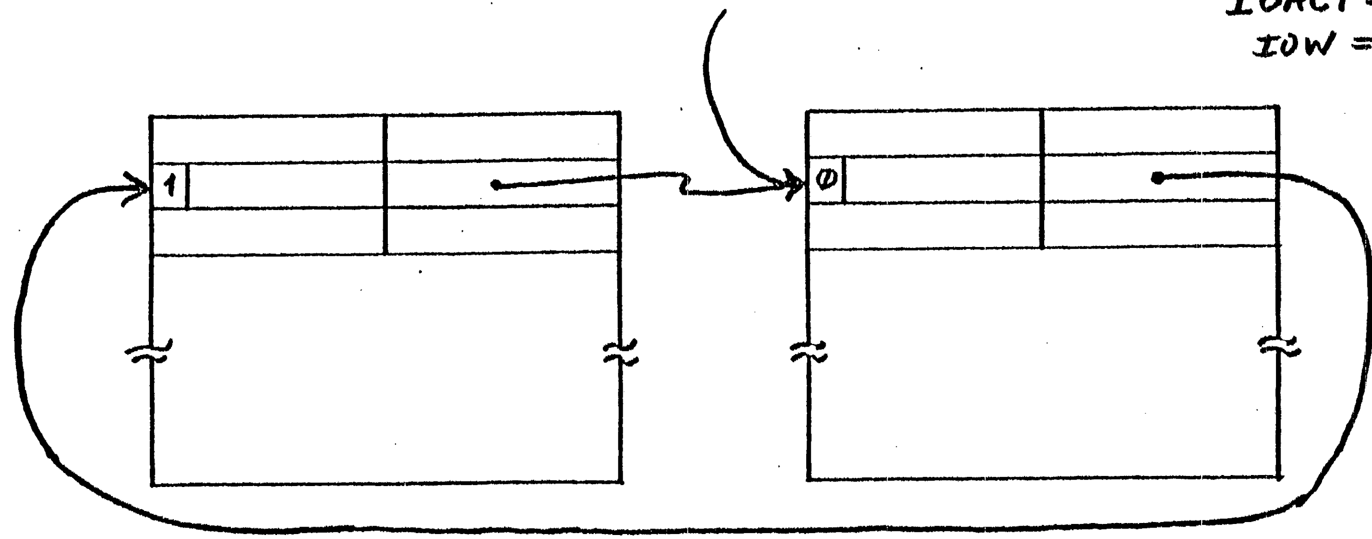
IOACT = 0
IOW = 0

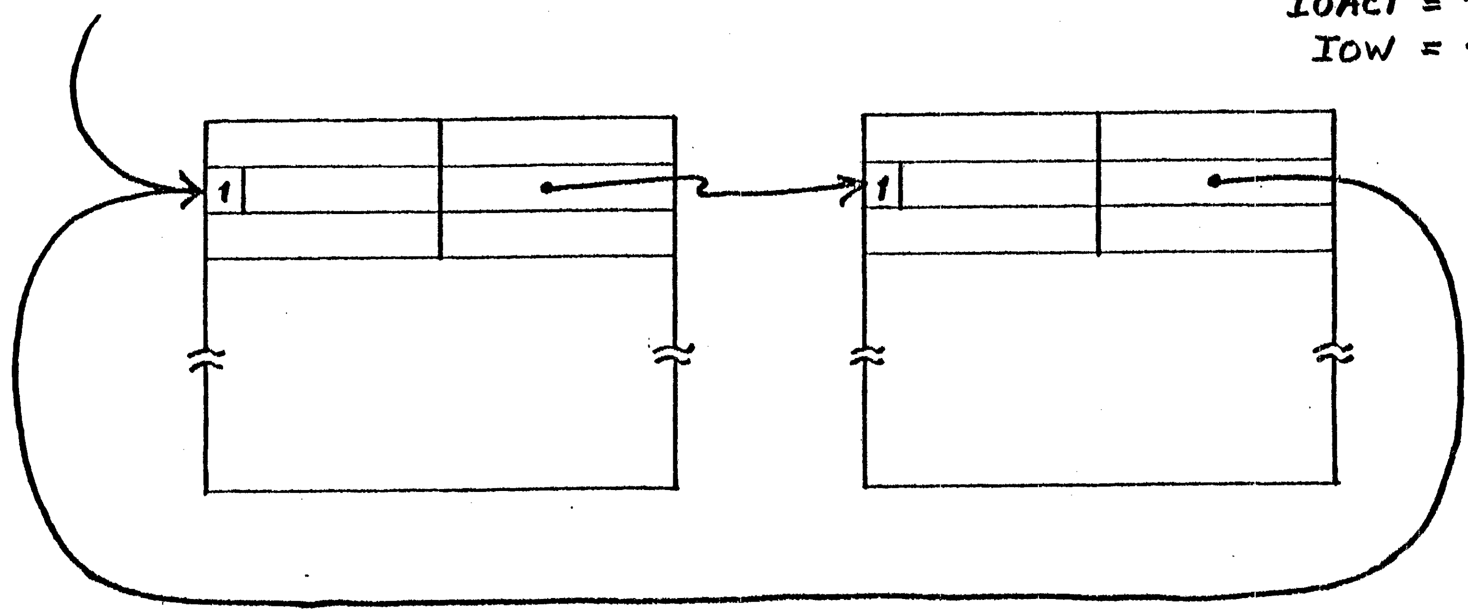


Buffer Ring after first OUT UUO

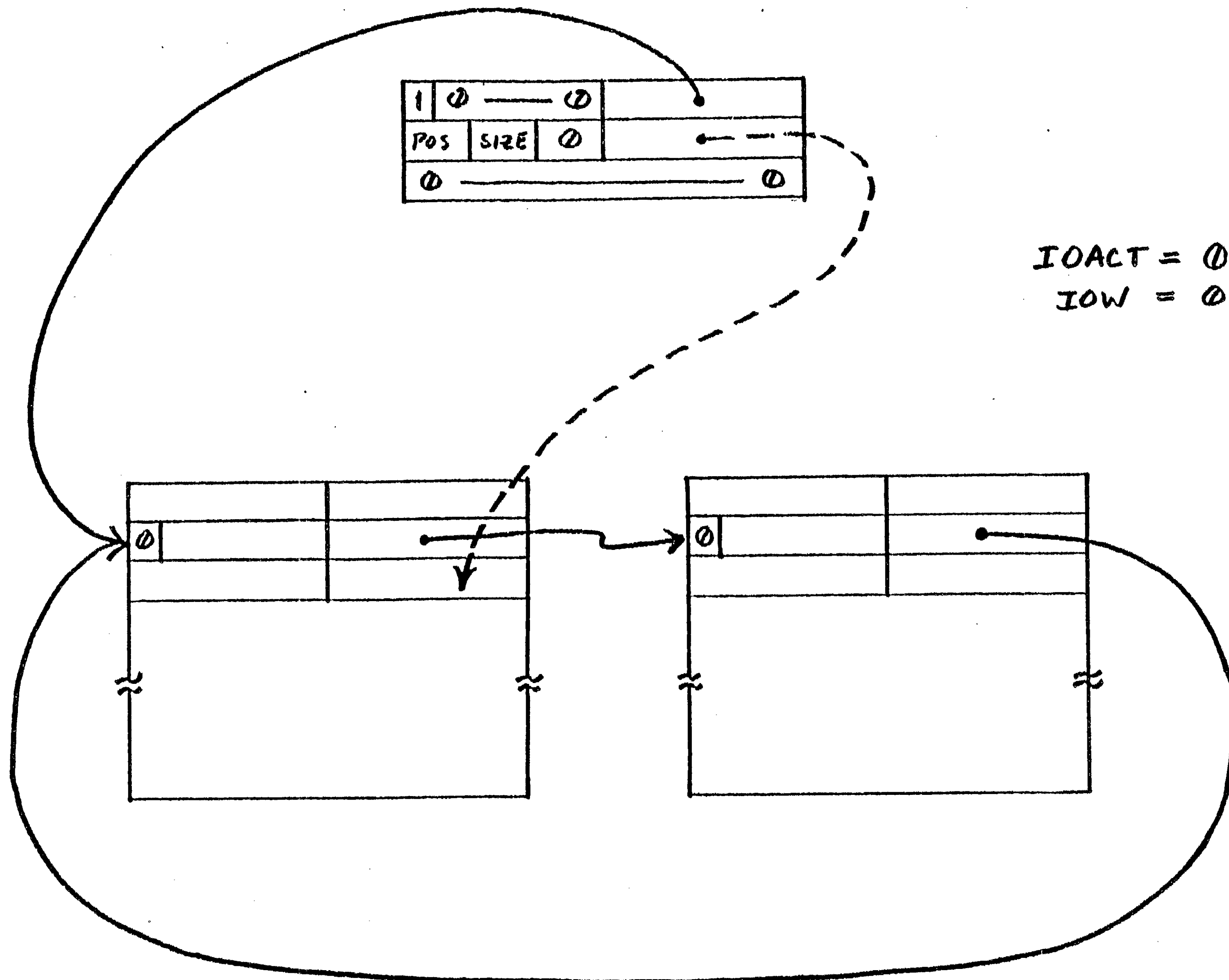


Buffer Ring after Successive OUT UUO's





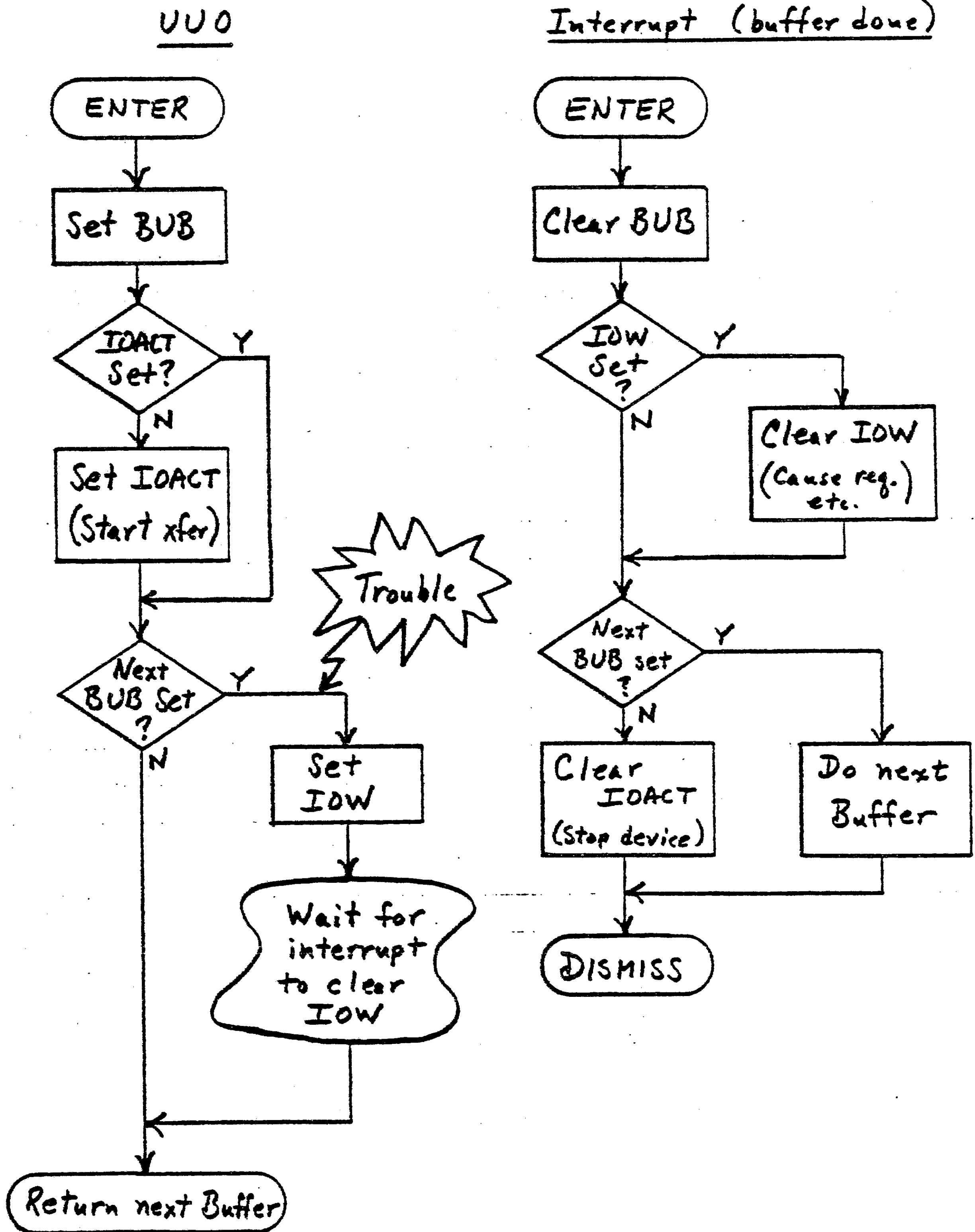
Buffer Ring after a CLOSE UUO



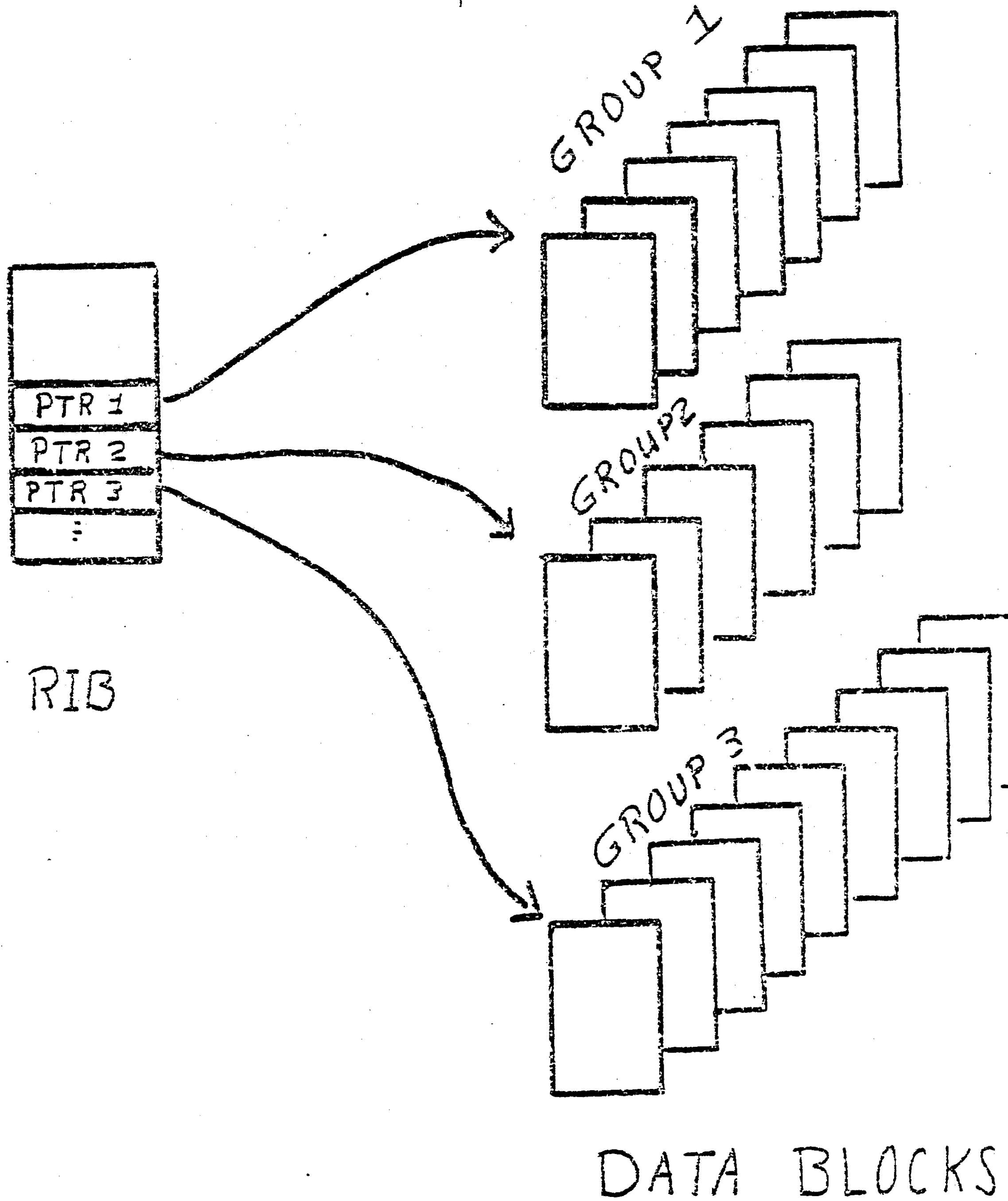
Flags used for TOPS-10's Asynchronous I/O

Flag	Target	Message	Action
IOACT	UUO-level code	Started I/O transfer	Set at UUO level Cleared at Interrupt level
IOW	Interrupt code	Scheduler should requeue	Set by UUO Clrd. by Interrupt
BUB	Both UUO and Interrupt code	Availability of Buffer	Set by UUO Clrd. by Interrupt

OUT



STRUCTURE OF DISK FILE

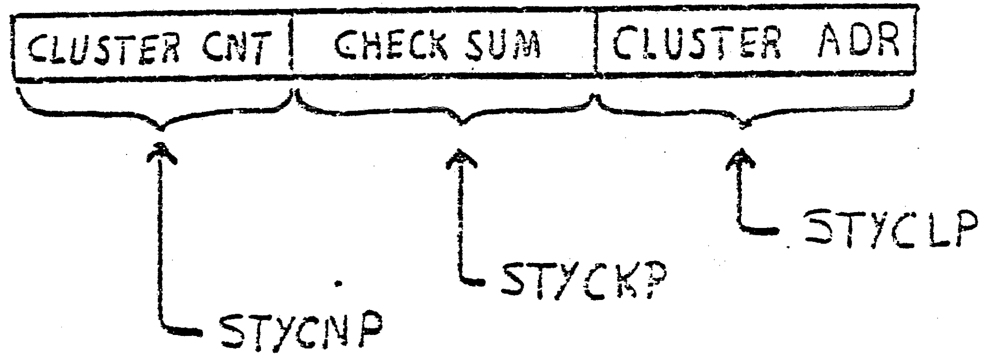


DATA BLOCKS

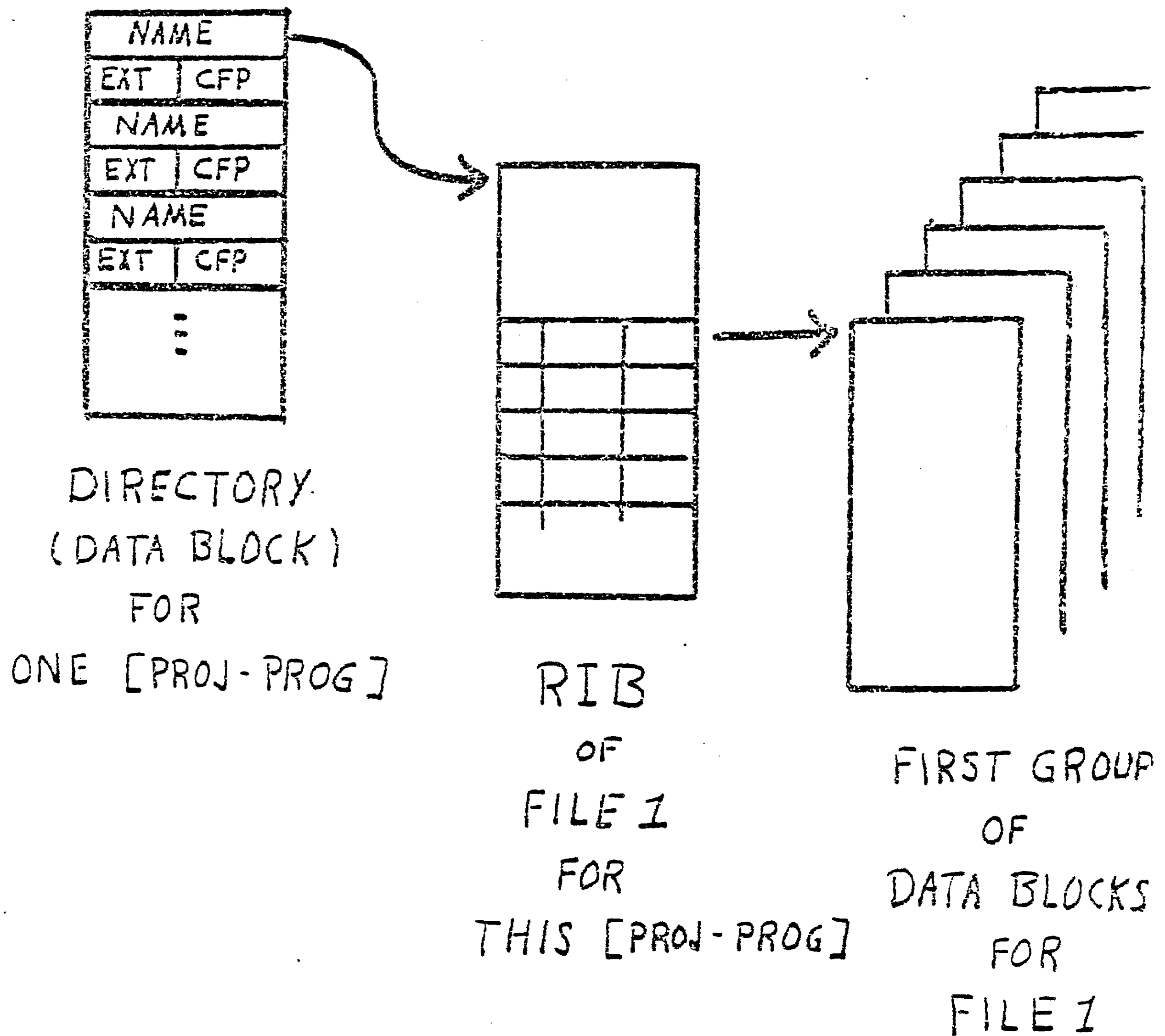
RETRIEVAL INFORMATION BLOCK

RIBFIR	-NR RETRIEVAL PTRS		FIRST PTR ADR	
RIBPPN	PROJECT		PROGRAMMER #	
RIBNAM	FILE NAME			
RIBEXT	FILE EXTENSION			ACC DATE
RIBPRV	PRV	MODE	CREAT TIME	CREAT DATE
RIBSIZ	FILE LENGTH IN WORDS			
	ADDITIONAL DESCRIPTIVE INFORMATION			
	RETRIEVAL POINTERS			

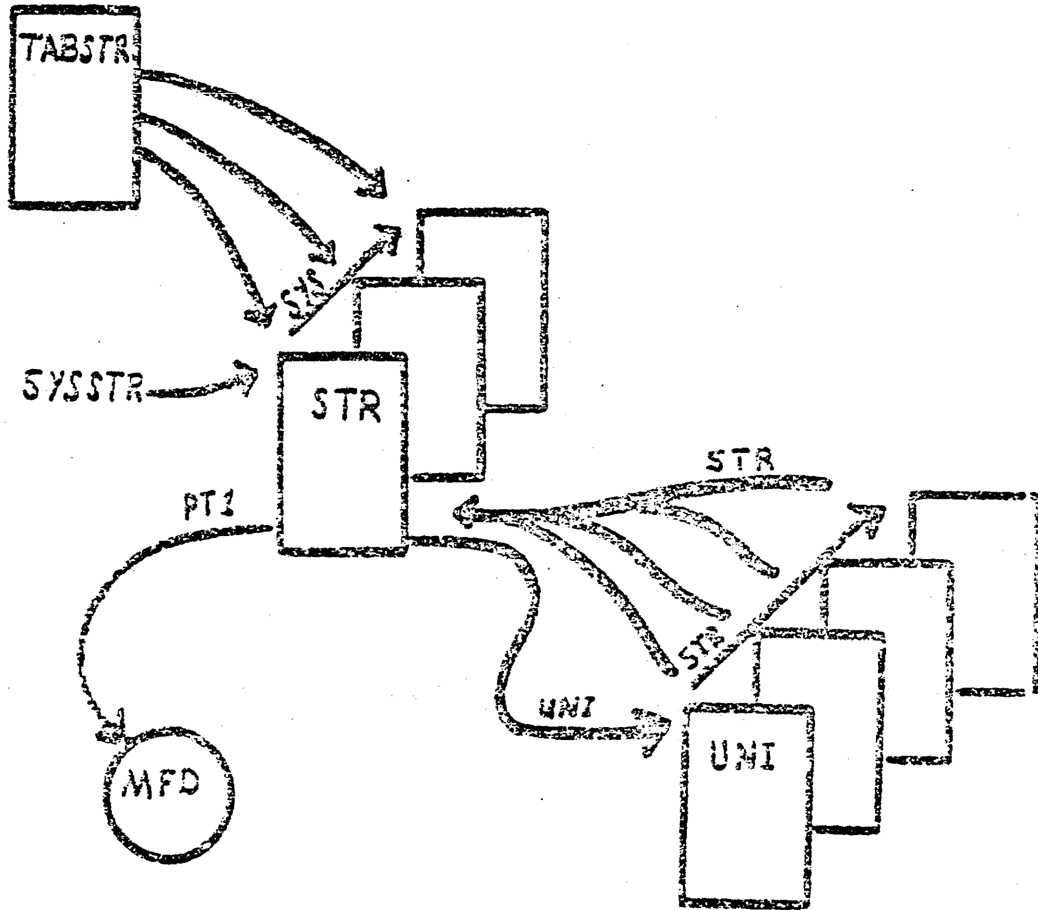
RETRIEVAL POINTERS



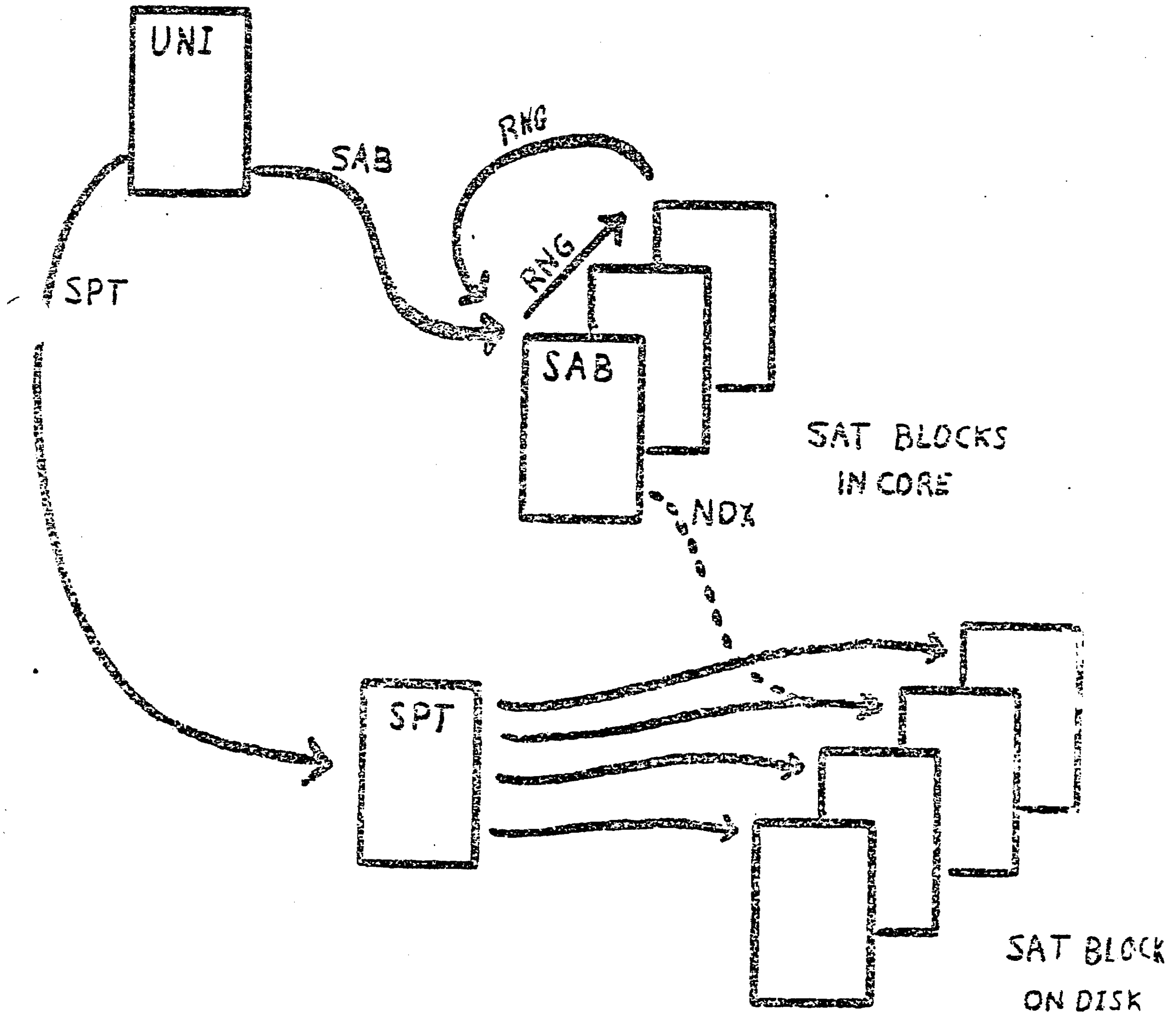
DIRECTORY STRUCTURE



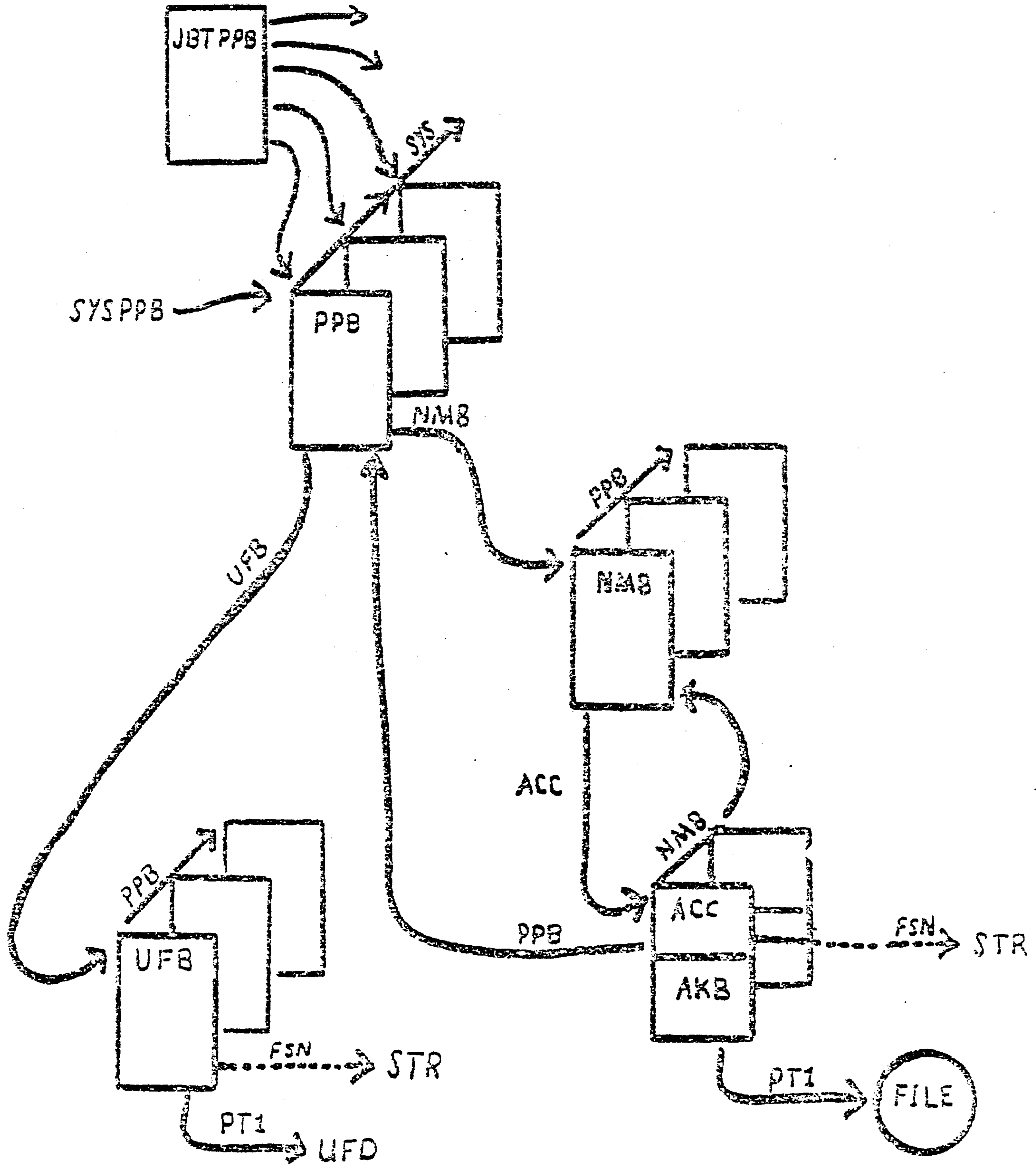
STR LINKAGES



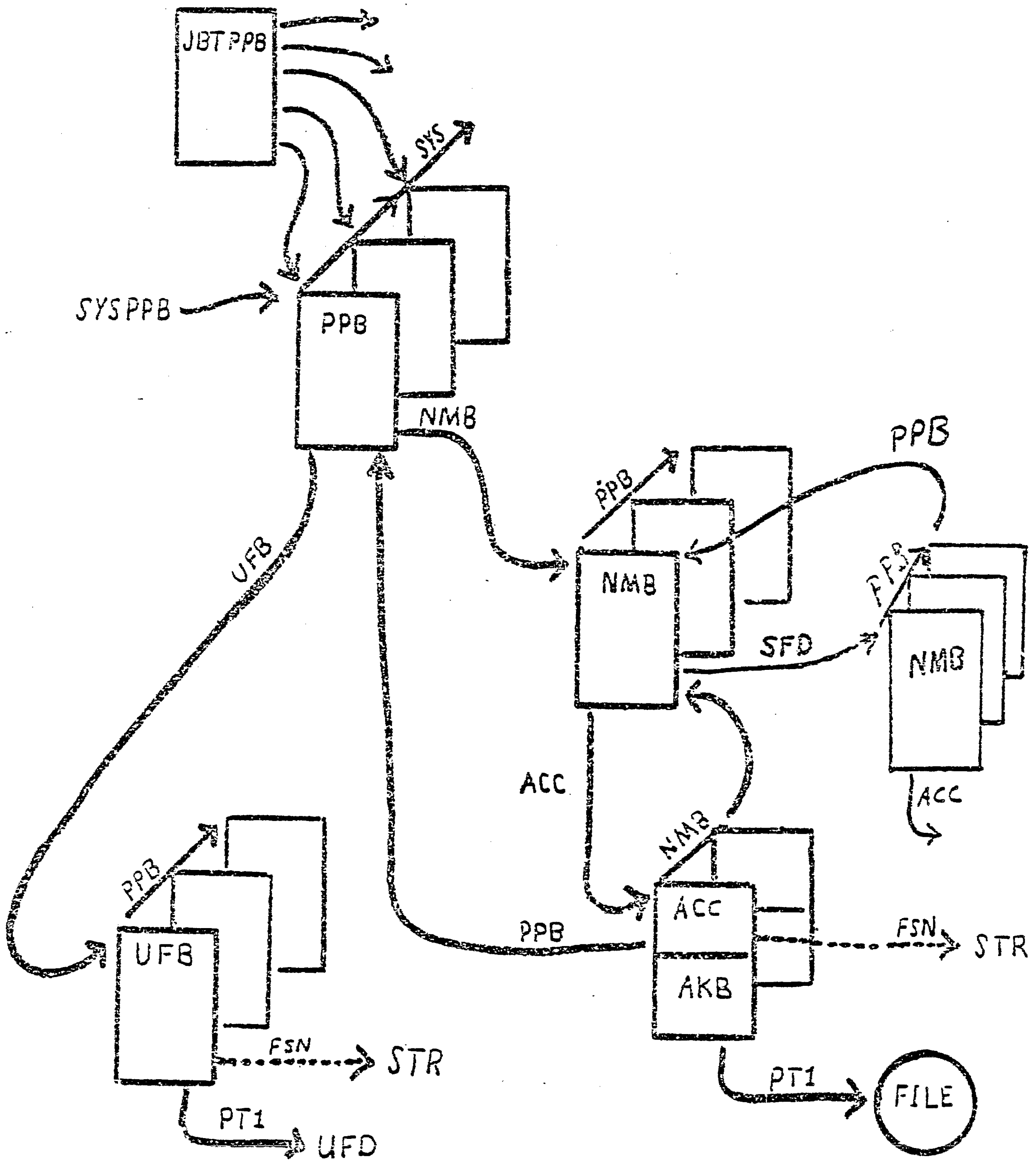
DISK STORAGE ALLOCATION



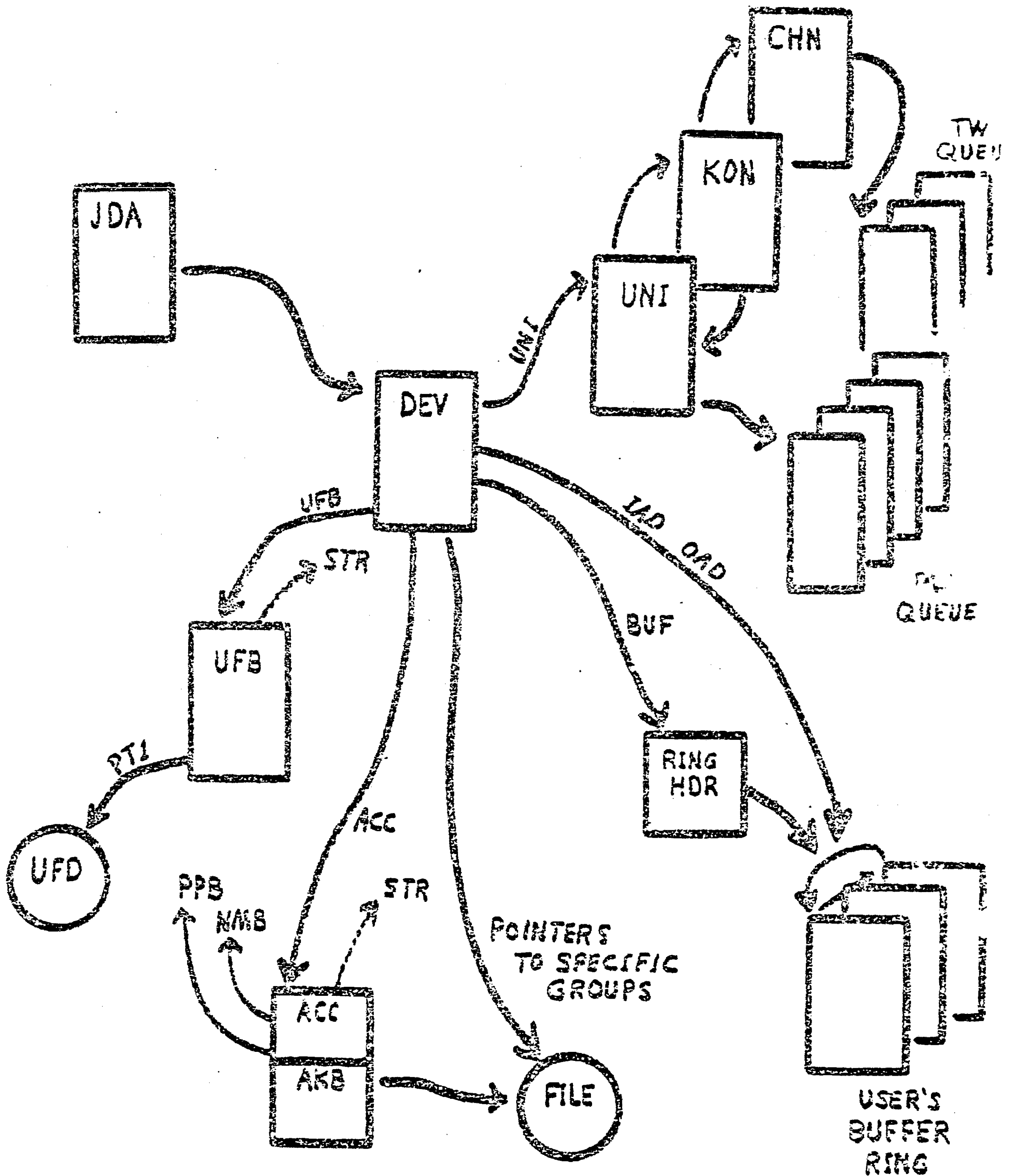
FILE ACCESS TABLES



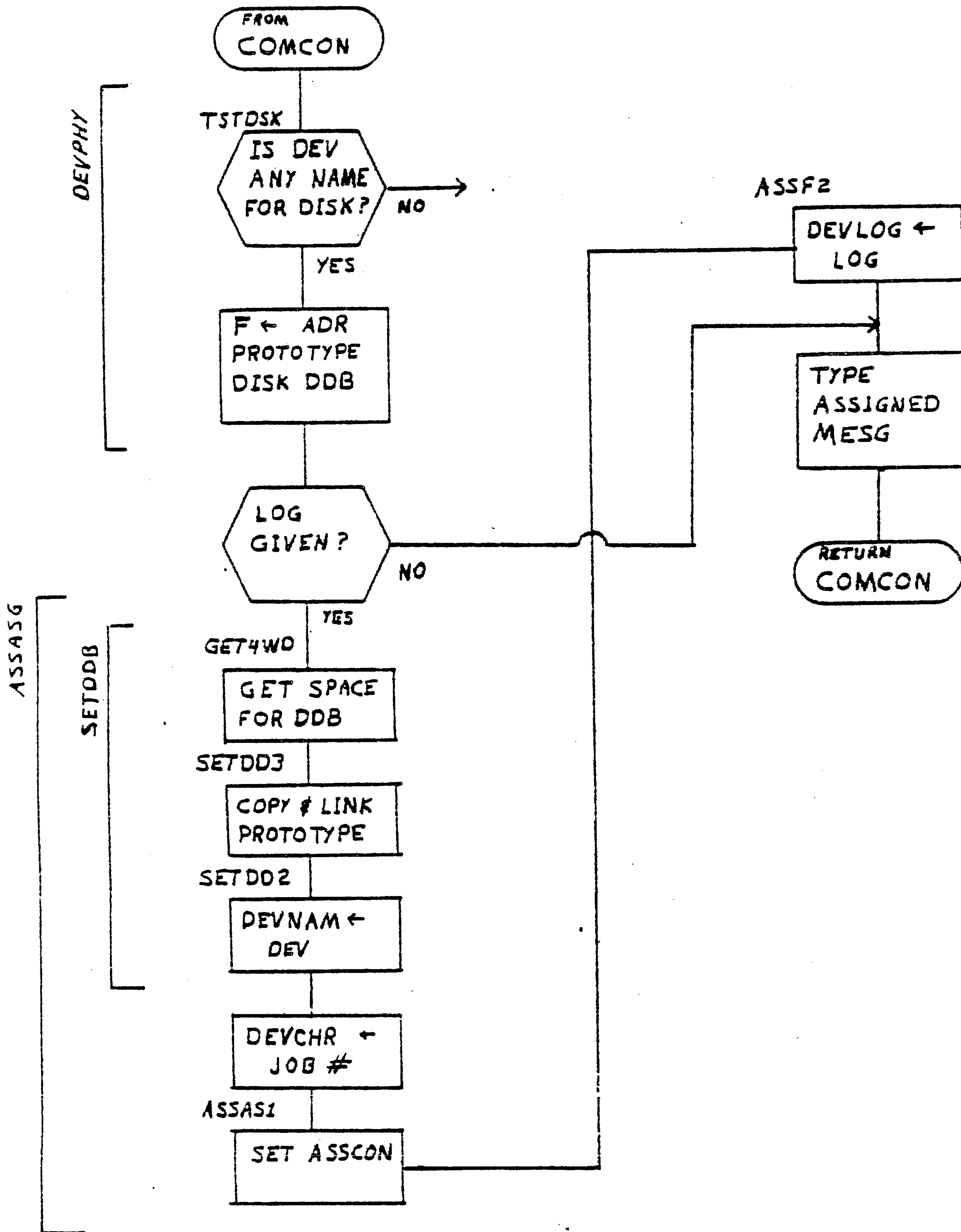
FILE ACCESS TABLES



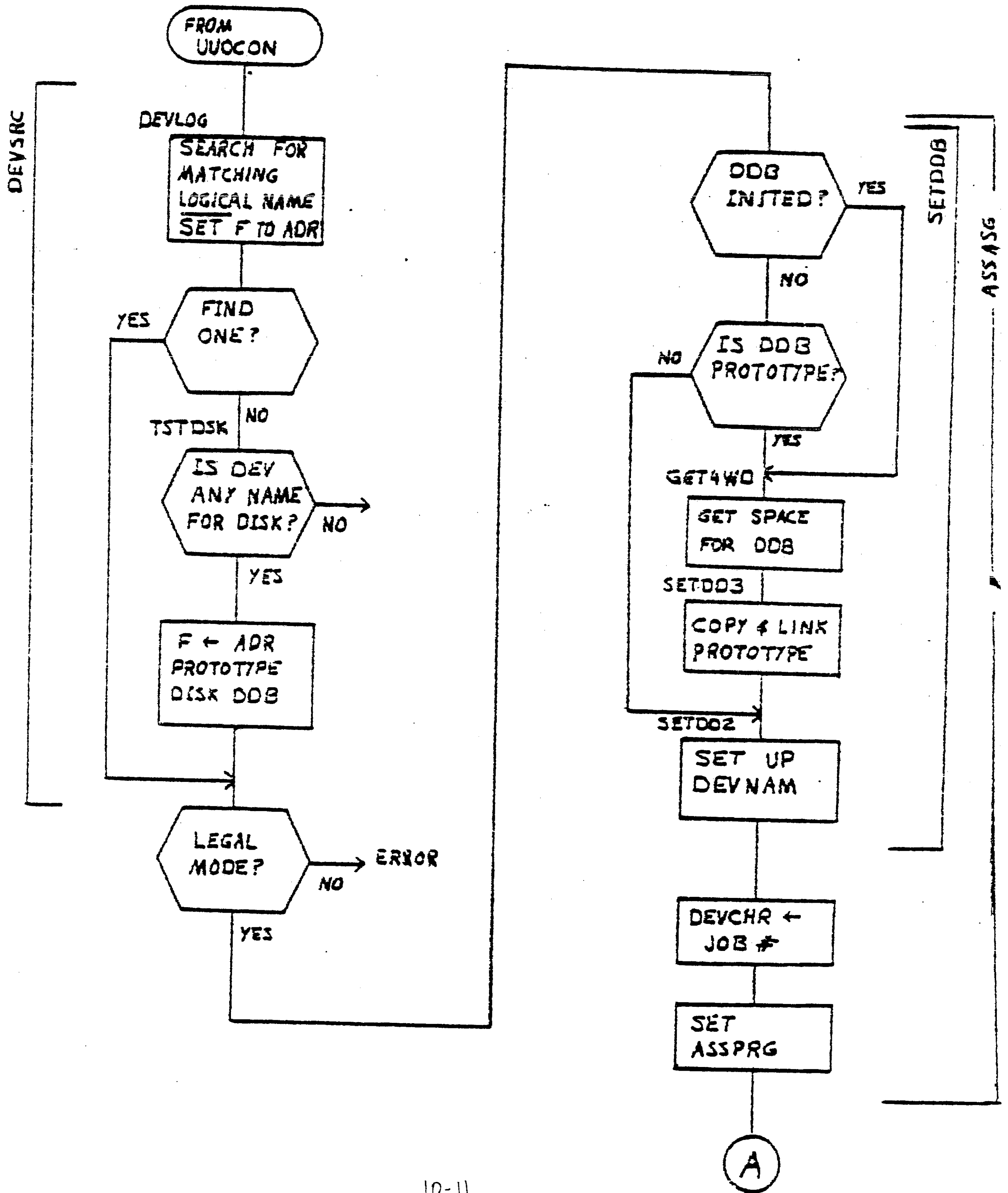
DDB LINKAGES



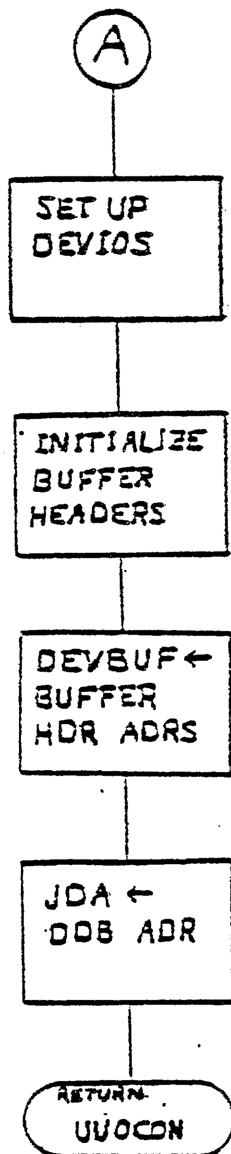
ASSIGN DEV LOG



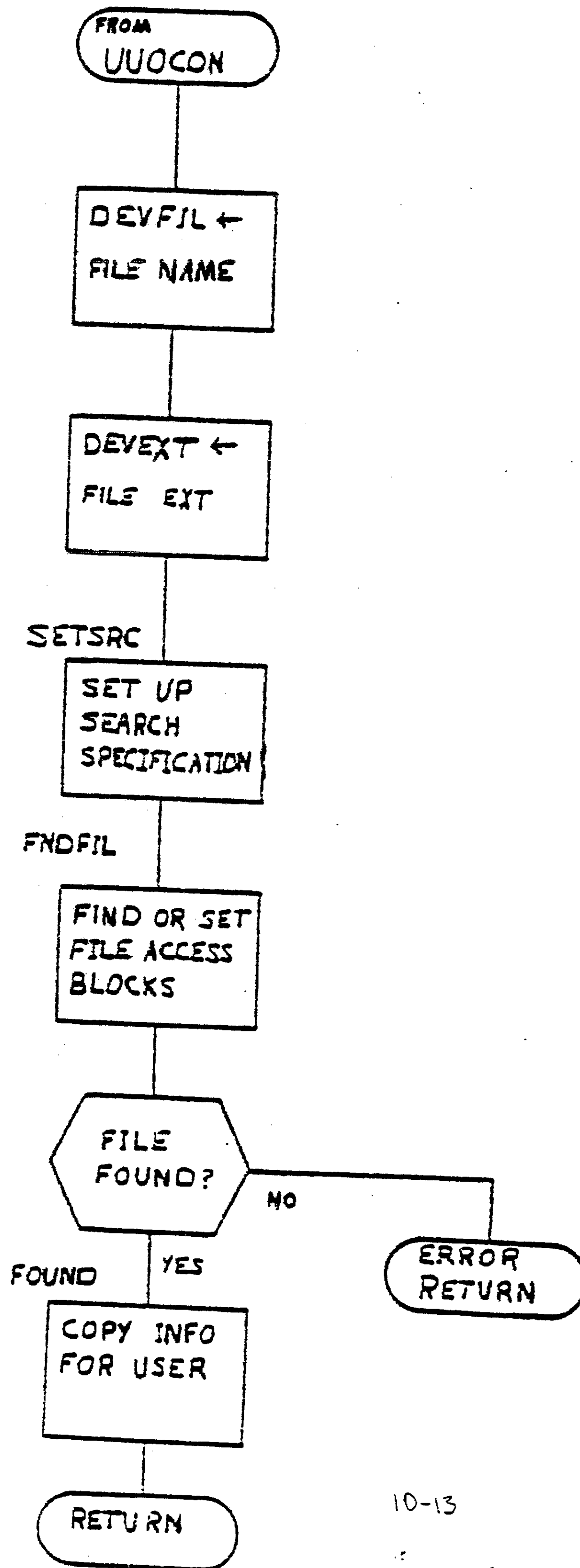
INIT DISK

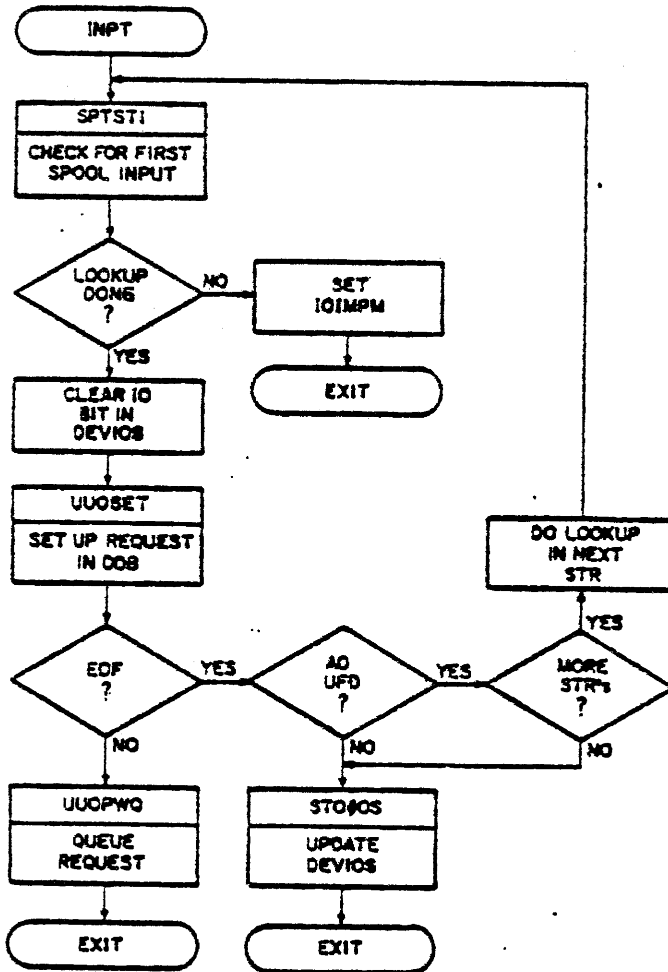


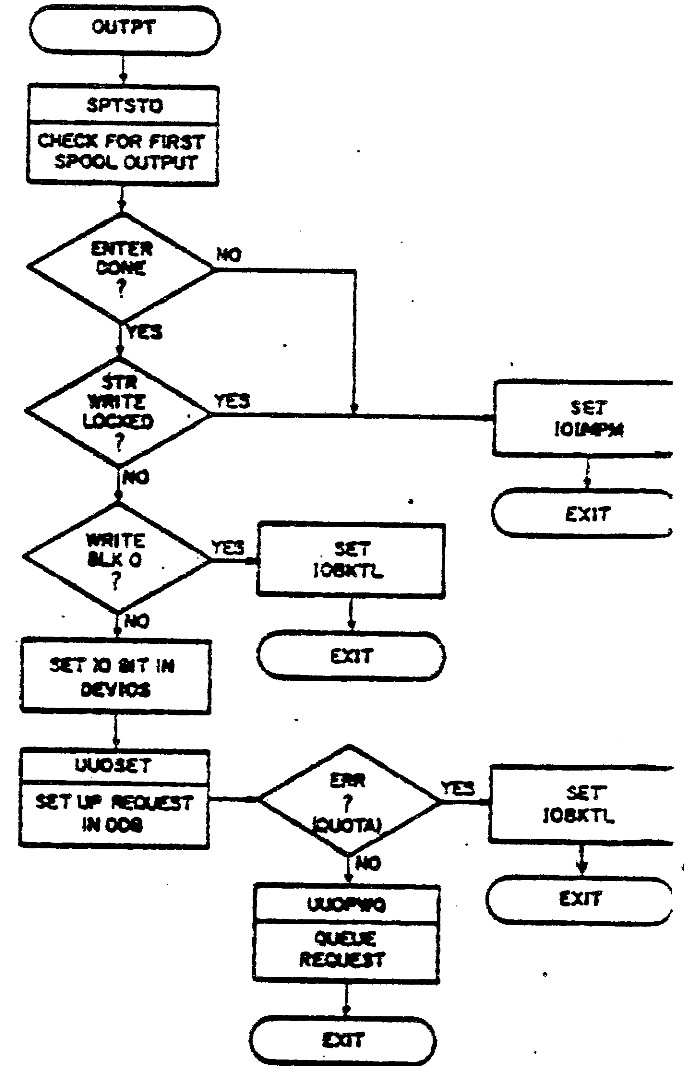
INIT DISK -2



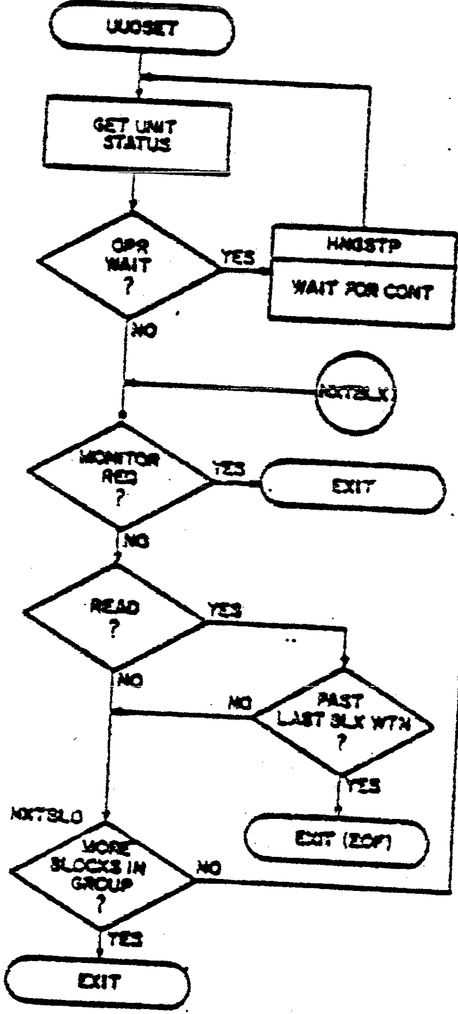
LOOKUP

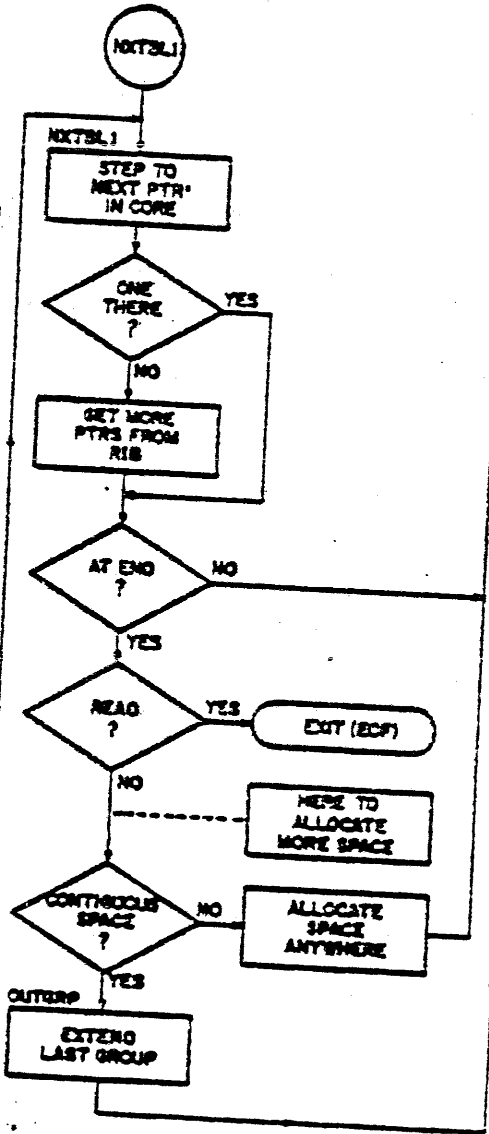


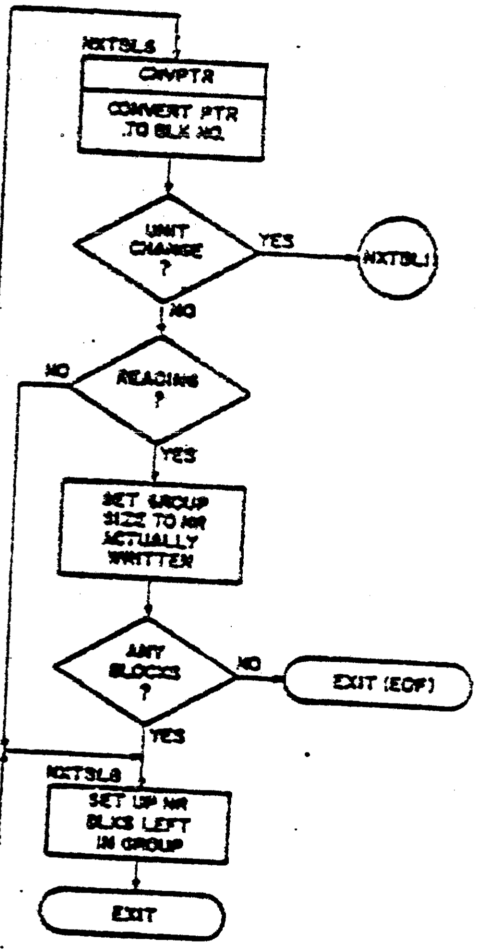

INPUT UUO
INPT
SPTSTI
CHECK FOR FIRST SPOOL INPUT
LOOKUP DONE ?
NO
SET IOIMPM
EXIT
YES
CLEAR IO BIT IN DEVIOS
UUOSET
SET UP REQUEST IN DDB
EOF ?
YES
AO UFD ?
YES
MORE STR's ?
YES
DO LOOKUP IN NEXT STR
NO
NO
NO
UUOPWQ
QUEUE REQUEST
EXIT
STO$OS
UPDATE DEVIOS
EXIT

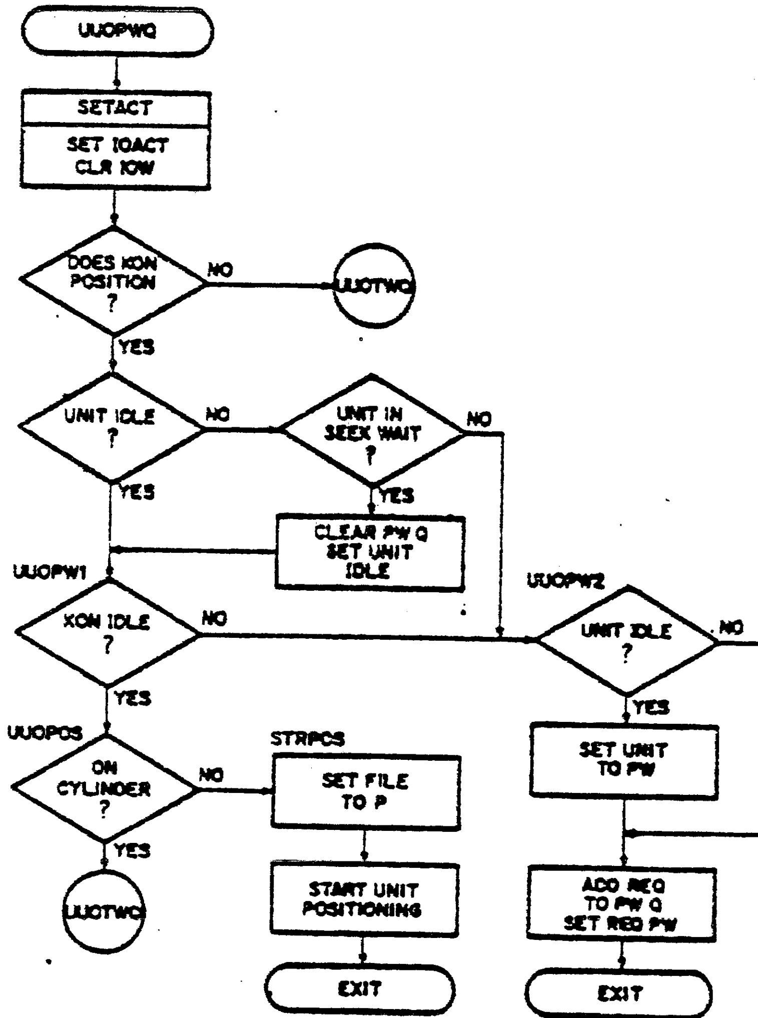

OUTPUT UUO
OUTPT
SPTSTO
CHECK FOR FIRST SPOOL OUTPUT
ENTER DONE ?
NO
YES
STR WRITE LOCKED ?
YES
SET IOIMPM
EXIT
NO
WRITE BLK 0 ?
YES
SET IOBKTL
EXIT
NO
SET IO BIT IN DEVIOS
UUOSET
SET UP REQUEST IN DDB
ERR ? (QUOTA)
YES
SET IOBKTL
EXIT
NO
UUOPWQ
QUEUE REQUEST
EXIT

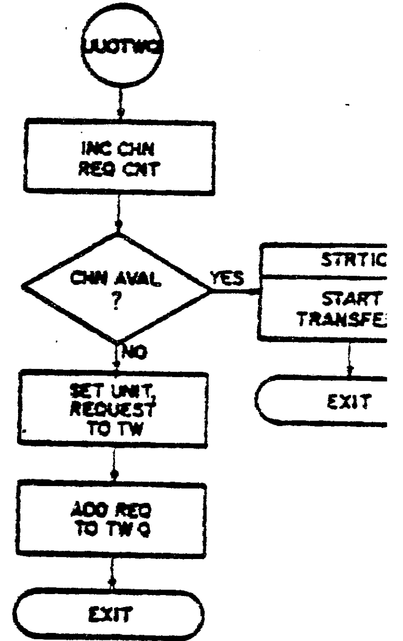
SET UP POINTERS





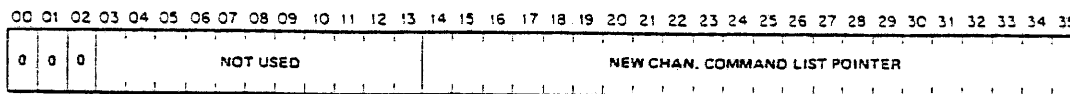



QUEUE FOR POSITIONING
UUOPWQ
SETACT
SET IOACT
CLR IOW
DOES KON POSITION ?
NO
UUOTWQ
YES
UNIT IDLE ?
NO
UNIT IN SEEK WAIT ?
NO
YES
YES
CLEAR PW Q
SET UNIT IDLE
UUOPW1
KON IDLE ?
NO
UUOPW2
UNIT IDLE ?
NO
YES
YES
UUOPOS
ON CYLINDER ?
NO
STRPOS
SET FILE TO P
SET UNIT TO PW
YES
UUOTWQ
START UNIT POSITIONING
ADD REQ TO PW Q
SET REQ PW
EXIT
EXIT

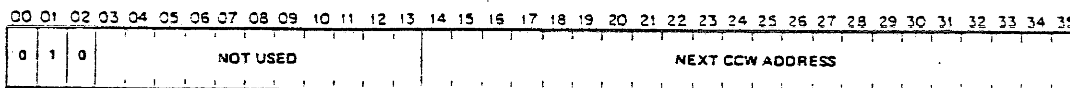

QUEUE FOR TRANSFER
UUOTWQ
INC CHN
REQ CNT
CHN AVAL ?
YES
STRTIC
START TRANSFE
NO
SET UNIT, REQUEST TO TW
EXIT
ADD REQ TO TW Q
EXIT

Channel Command Word Format

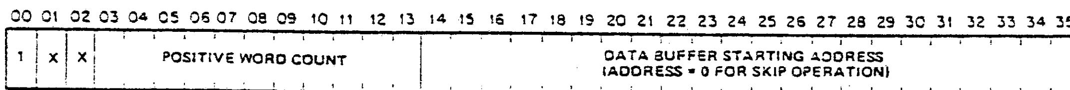
CCW – HALT



CCW – JUMP



CCW – DATA XFER



HALT
LAST
XFER

REVERSE

10-2193

OP CODE (bits 00–02)

0_8 – Causes halt in command list execution (HALT command).

2_8 – Causes branch in command list execution (JUMP command).

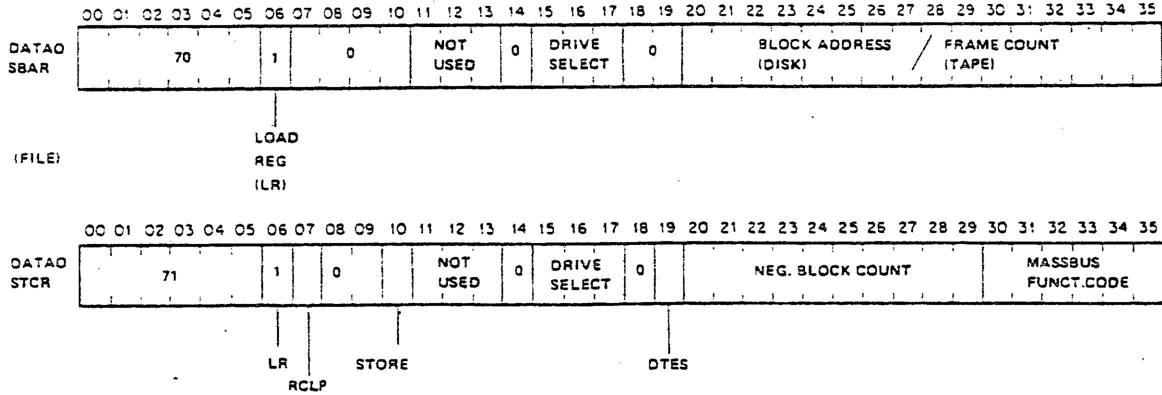
4_8 – Causes a forward data transfer (device read or write) without halting (DATA TRANSFER command).

5_8 – Causes a reverse data transfer (device read only) without halting (DATA TRANSFER command).

6_8 – Causes a forward data transfer (device read or write) and a halt ("LAST" DATA TRANSFER command).

7_8 – Causes a reverse data transfer (device read only) and a halt ("LAST" DATA TRANSFER command).

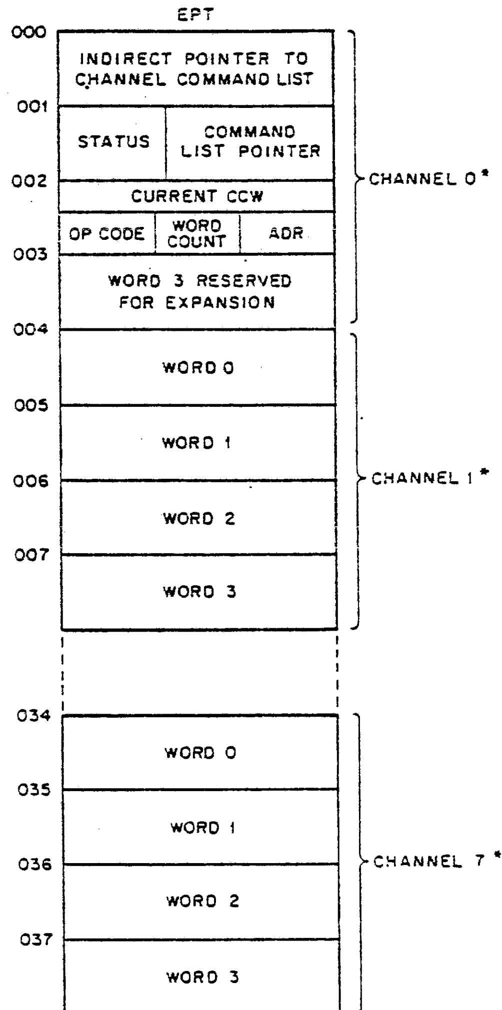
DATAO Command Format



Drive Commands

Command Code (Octal)	Drum and Fixed-Head Disk (RS04)	Moving-Head Disk (RP04)	Magnetic Tape (TM02/TU45)
01	No Operation	No Operation	No Operation
03		Unload	Rewind, Off-line
05		Seek	
07		Recalibrate	Rewind
11	Drive Clear	Drive Clear	Drive Clear
13		Release	
15		Offset	
17		Return to Centerline	
21	Readin Preset	Readin Preset	Readin Preset
23		Pack Acknowledge	
25			Erase
27			Write File Mark
31	Search	Search	Space Forward
33			Backspace
*51	Write Check Data	Write Check Data	Write Check Forward
*53		Write Check Header and Data	
*57			Write Check Reverse
61	Write Data	Write Data	Write Forward
63		Write Header and Data	
71	Read Data	Read Data	Read Forward
73		Read Header and Data	
77			Read Reverse

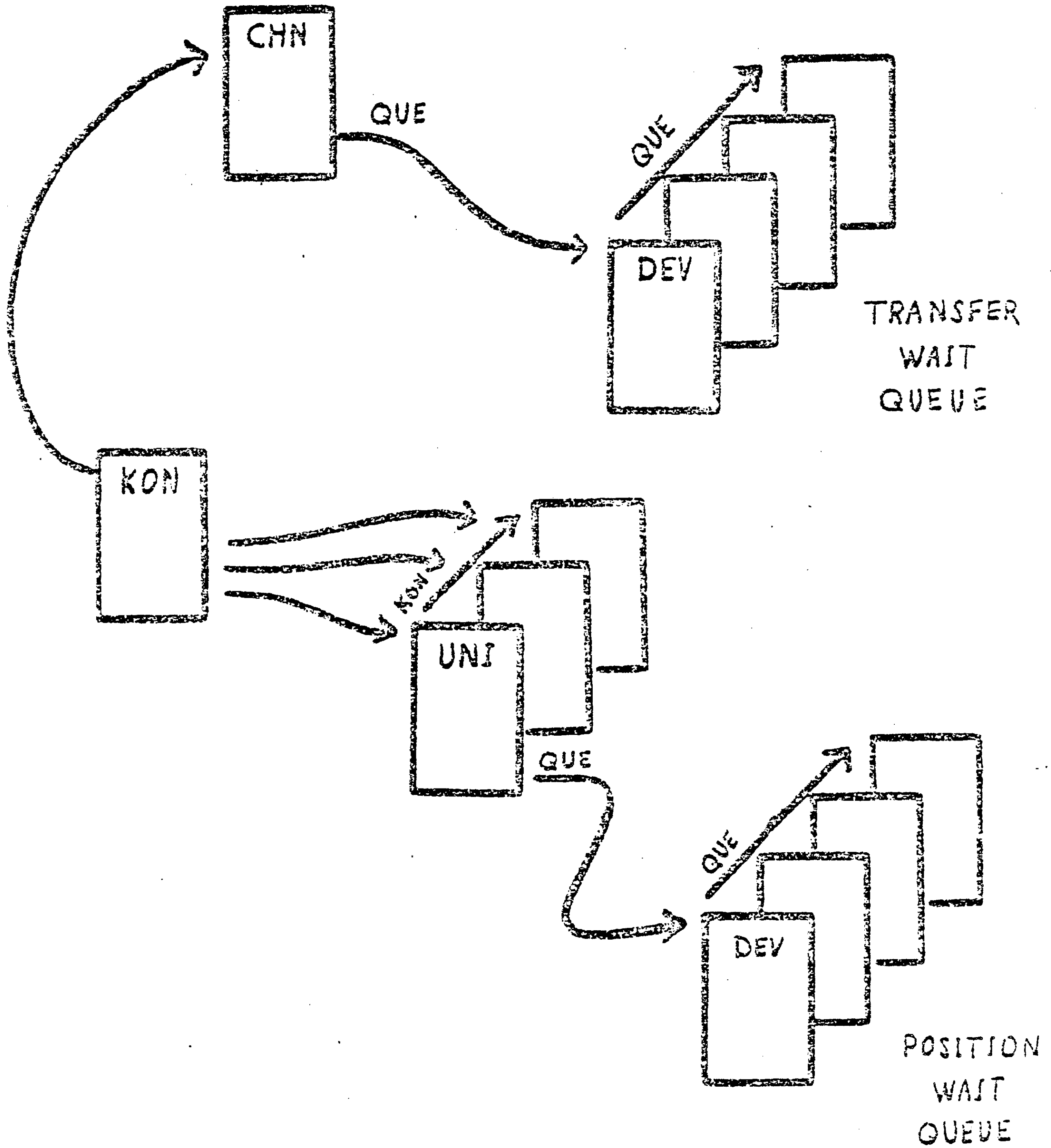
Channel Reset and Status Logout Area

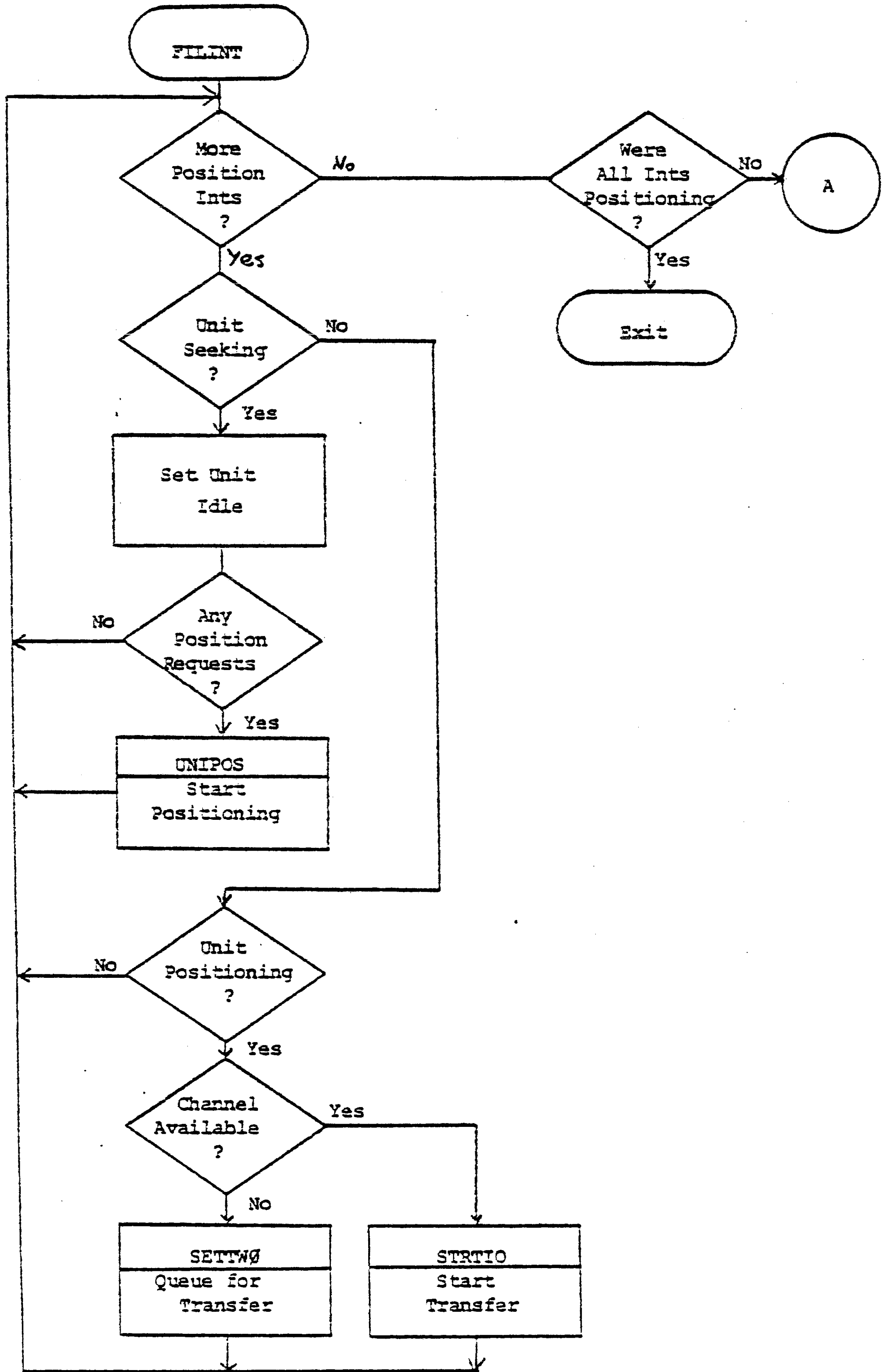


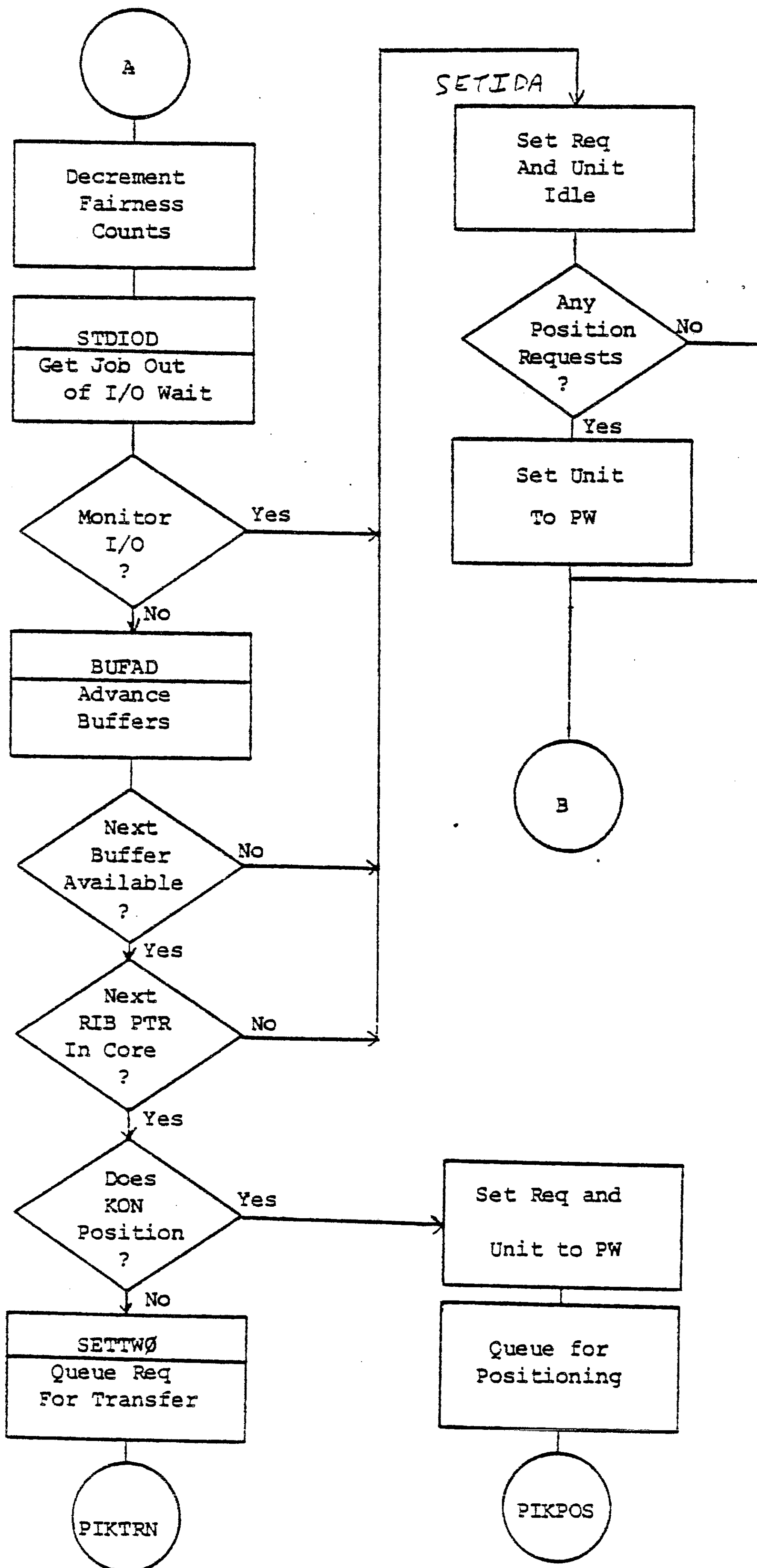
*EPT + 4 (RH20 PHYSICAL NUMBER) + 0 = WORD 0
EPT + 4 (RH20 PHYSICAL NUMBER) + 1 = WORD 1
EPT + 4 (RH20 PHYSICAL NUMBER) + 2 WORD 2

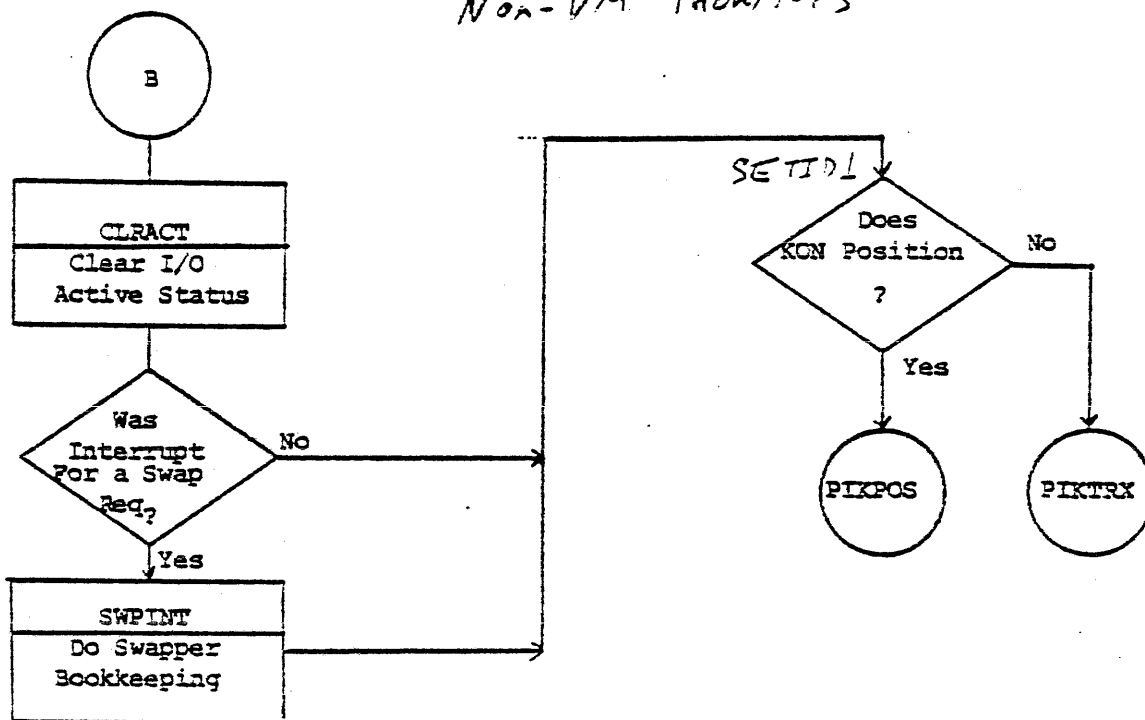
10-2086

DISK QUEUES

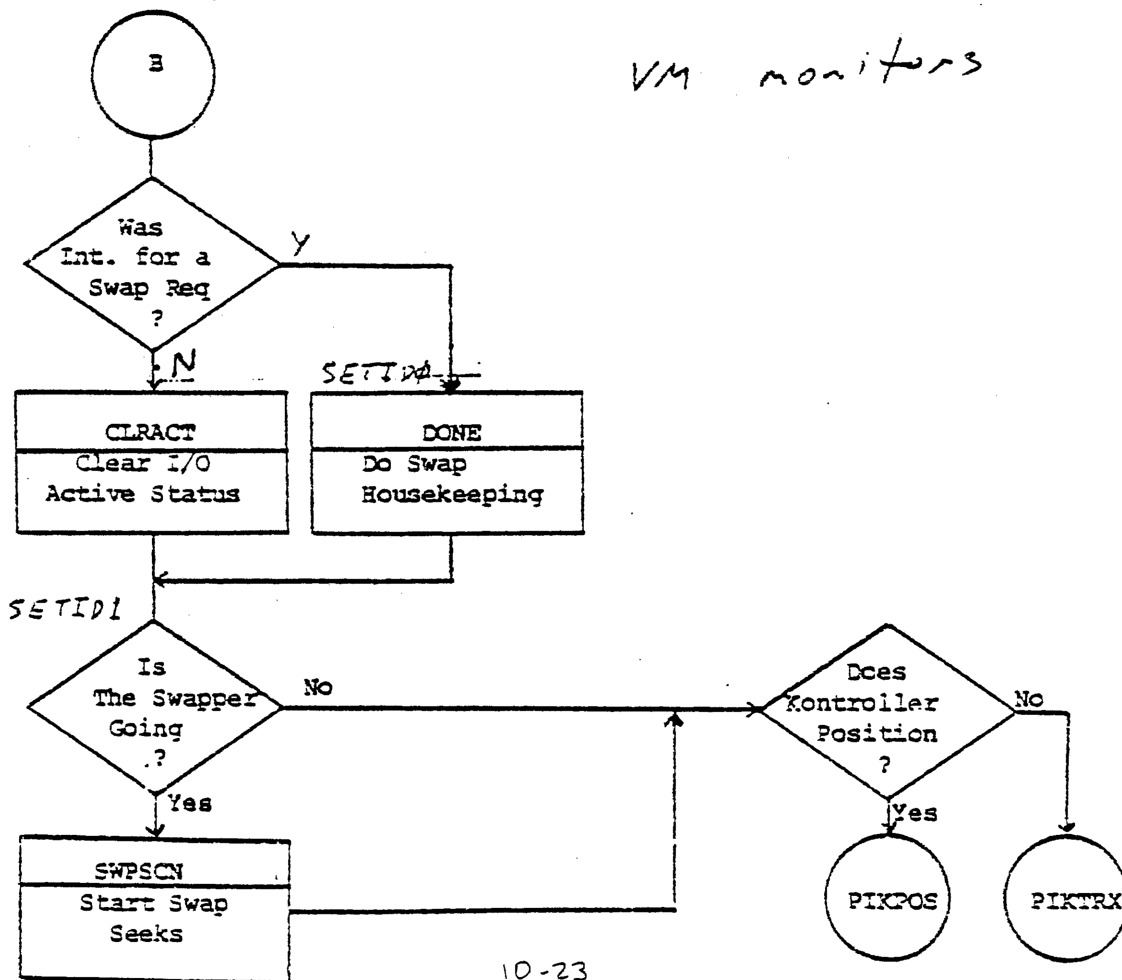



FILINT
More Position Ints ?
No
Yes
Were All Ints Positioning ?
No
A
Yes
Exit
Unit Seeking ?
No
Yes
Set Unit Idle
Any Position Requests ?
No
Yes
UNIPOS
Start Positioning
Unit Positioning ?
No
Yes
Channel Available ?
Yes
No
SETTWØ
Queue for Transfer
STRTIO
Start Transfer

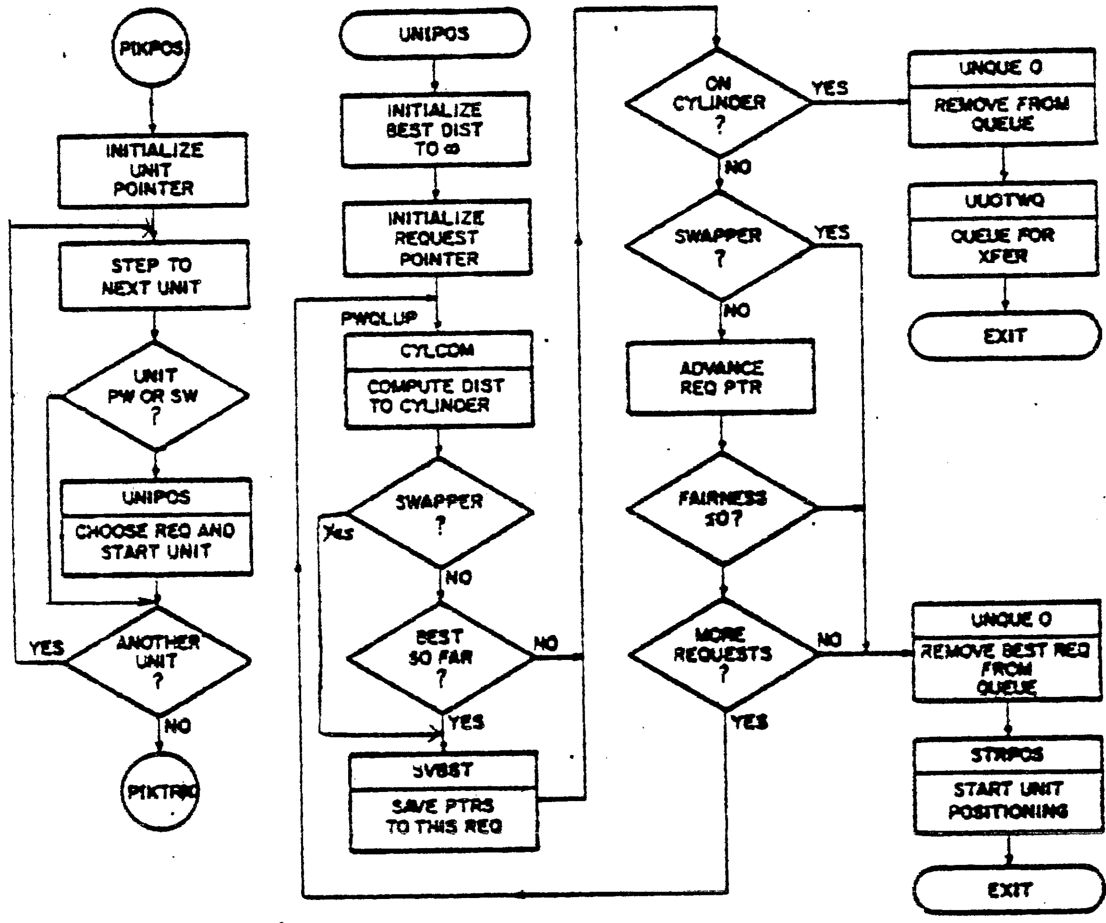

A
Decrement
Fairness
Counts
STDIOD
Get Job Out
of I/O Wait
Monitor
I/O
?
Yes
No
BUFAD
Advance
Buffers
Next
Buffer
Available
?
No
Yes
Next
RIB PTR
In Core
?
No
Yes
Does
KON
Position
?
Yes
No
SETTWØ
Queue Req
For Transfer
PIKTRN
SETIDA
Set Req
And Unit
Idle
Any
Position
Requests
?
No
Yes
Set Unit
To PW
B
Set Req and
Unit to PW
Queue for
Positioning
PIKPOS



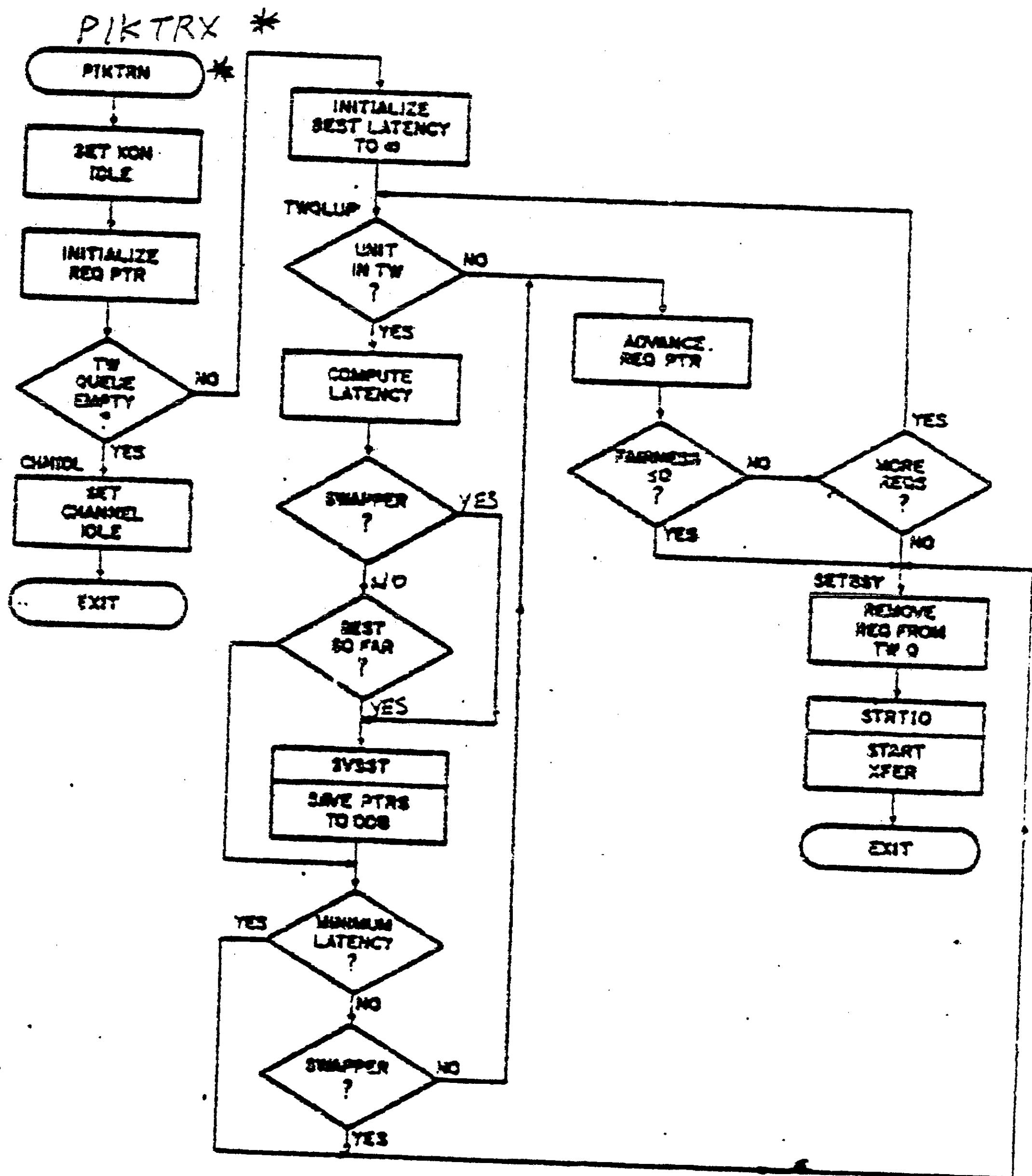
Non-VM monitors
B
CLRACT
Clear I/O
Active Status
Was
Interrupt
For a Swap
Req?
No
Yes
SWPINT
Do Swapper
Bookkeeping
SETID1
Does
KON Position
?
No
Yes
PIKPOS
PIKTRX



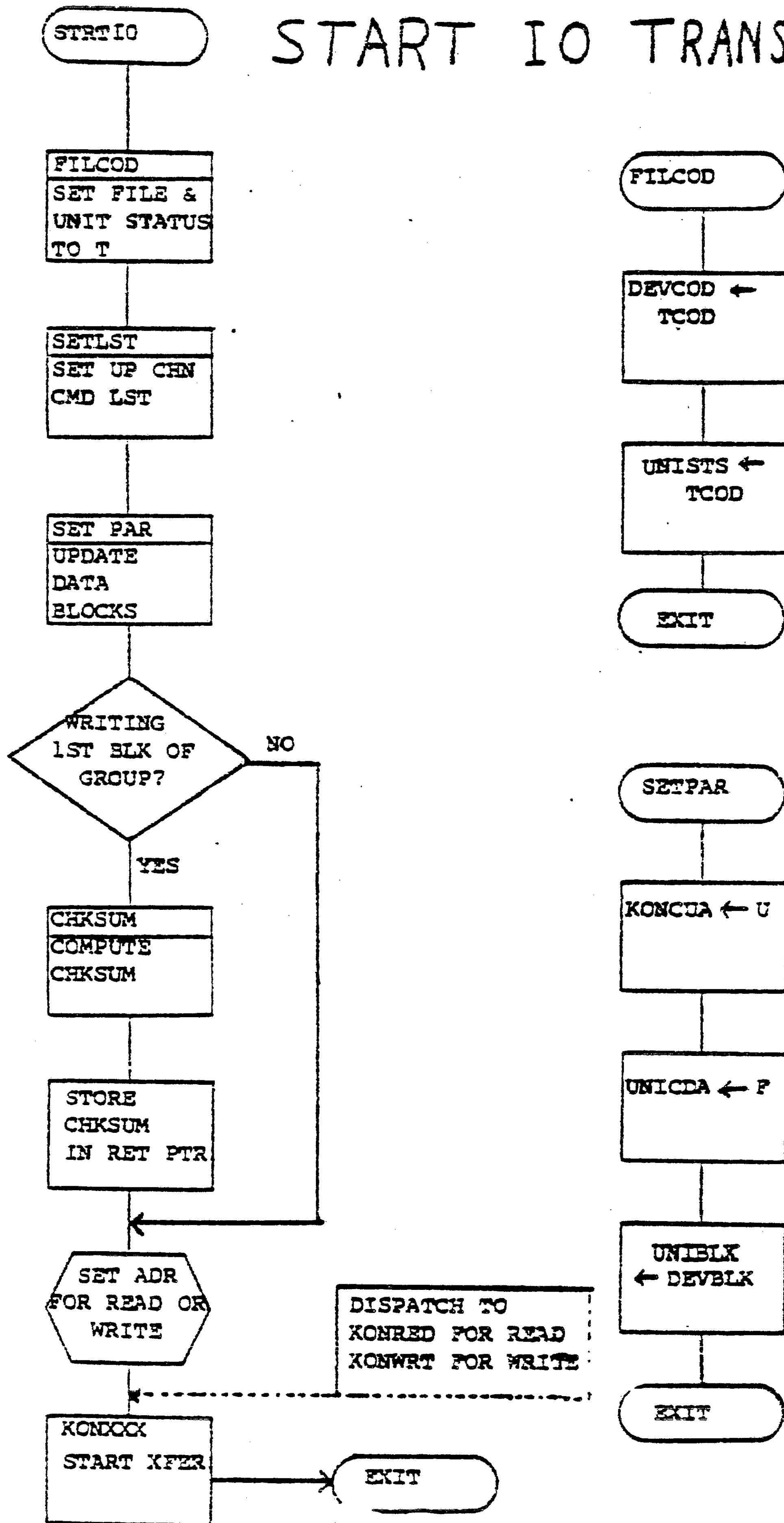
VM monitors
B
Was
Int. for a
Swap Req
?
Y
N
SETIDA
CLRACT
Clear I/O
Active Status
DONE
Do Swap
Housekeeping
SETID1
Is
The Swapper
Going
?
No
Yes
SWPSCN
Start Swap
Seeks
Does
Kontroller
Position
?
No
Yes
PIKPOS
PIKTRX


PIKPOS
INITIALIZE UNIT POINTER
STEP TO NEXT UNIT
UNIT PW OR SW ?
UNIPOS
CHOOSE REQ AND START UNIT
ANOTHER UNIT ?
YES
NO
PIKTRK
UNIPOS
INITIALIZE BEST DIST TO ∞
INITIALIZE REQUEST POINTER
PWQLUP
CYLCOM
COMPUTE DIST TO CYLINDER
SWAPPER ?
Yes
NO
BEST SO FAR ?
NO
YES
SVBST
SAVE PTRS TO THIS REQ
ON CYLINDER ?
YES
NO
SWAPPER ?
YES
NO
ADVANCE REQ PTR
FAIRNESS ≤0?
MORE REQUESTS ?
NO
YES
UNQUE 0
REMOVE FROM QUEUE
UUOTWQ
QUEUE FOR XFER
EXIT
UNQUE 0
REMOVE BEST REQ FROM QUEUE
STRPOS
START UNIT POSITIONING
EXIT

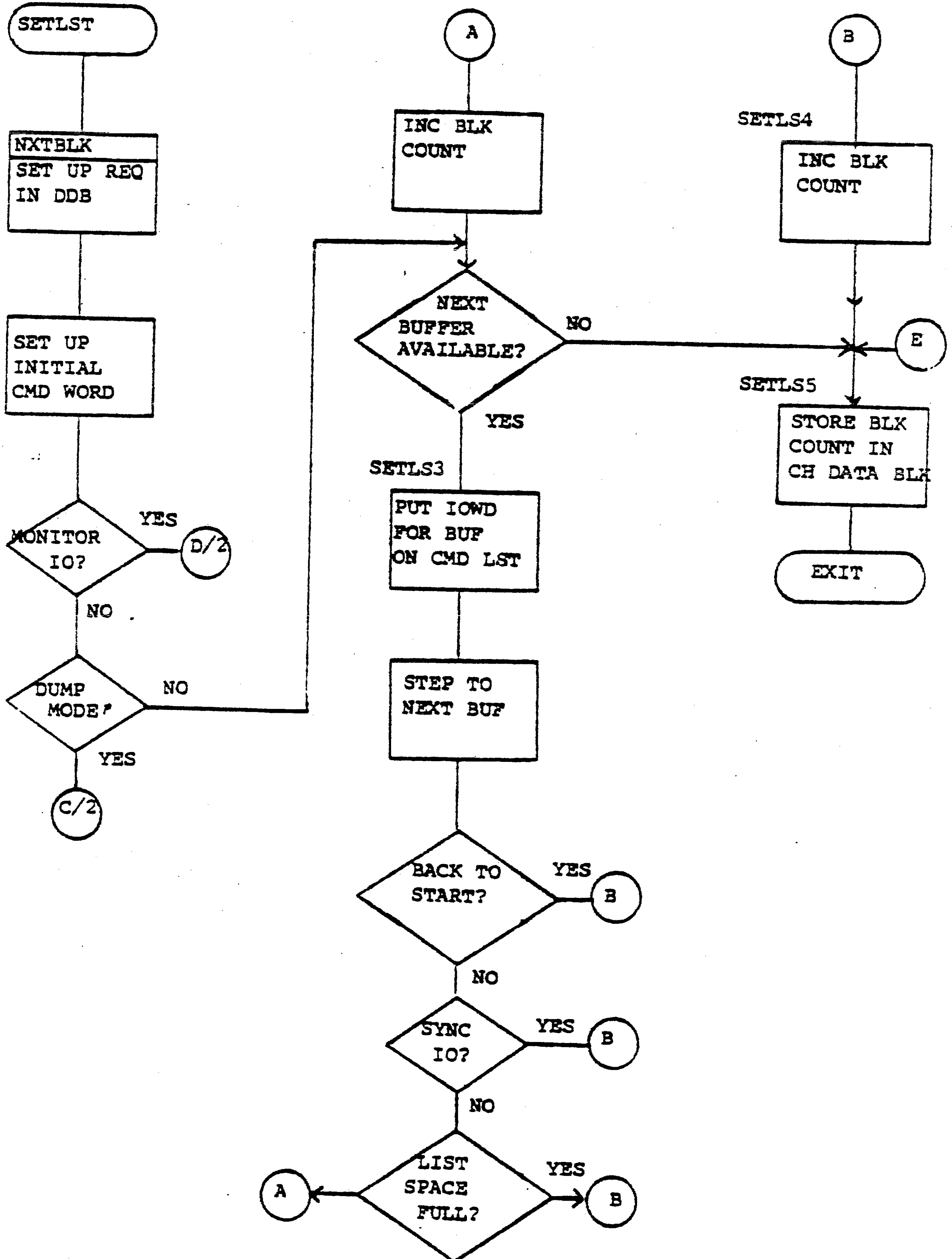
PIKTRX *



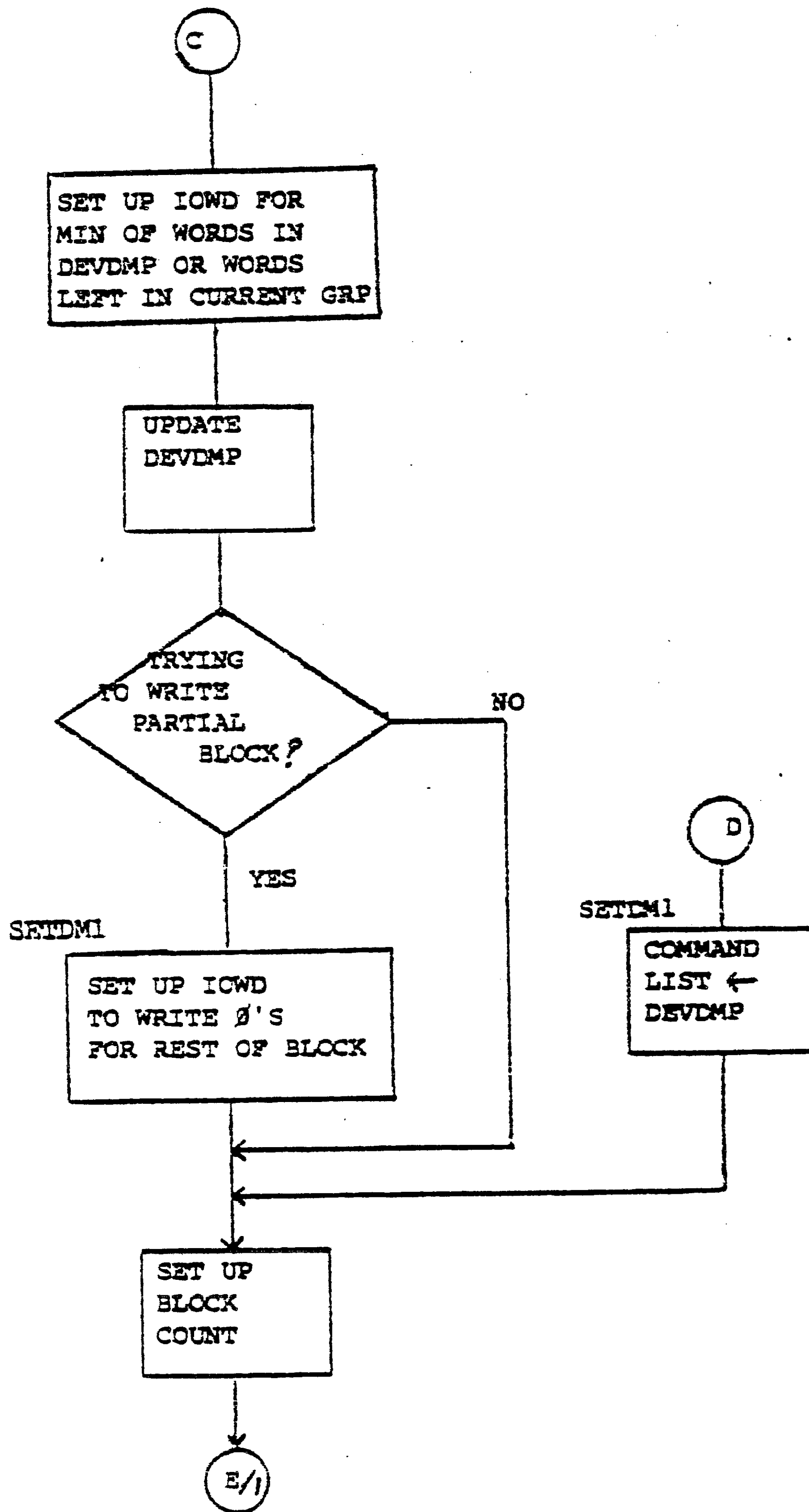
START IO TRANSFER



SET UP COMMAND LIST



SETLST - 2



PART 2

KL DOCUMENT

KL Document

DIGITAL EQUIPMENT CORPORATION
Educational Services
February 1978
REVISION I

K L Document

Copyright © 1978 by DIGITAL EQUIPMENT CORPORATION

The material in this document is for informational purposes and is subject to change without notice; it should not be construed as a commitment by Digital Equipment Corporation. Digital Equipment Corporation assumes no responsibility for any errors which may appear in this document.

The information in this document is furnished under a license and may be used or copied only in accordance with the terms of such license.

Digital Equipment Corporation assumes no responsibility for the use or reliability of its software on equipment that is not supplied by Digital.

The following are trademarks of Digital Equipment Corporation:

DIGITAL	DECsystem-10	MASSBUS
DEC	DECtape	OMNIBUS
PDP	DIBOL	OS/8
DECUS	EDUSYSTEM	PHA
UNIBUS	FLIP CHIP	RSTS
COMPUTER LABS	FOCAL	RSX
COMTEX	INDAC	TYPESET-8
DDT	LAB-8	TYPESET-10
DECCOM	DECSYSTEM-20	TYPESET-11

A

TABLE OF CONTENTS

Section	Subject
1	Introduction and KL Orientation 1
2	Paging on a KL Processor 4
	-1Ø-style 7
	-2Ø-style 14
3	KL Hardware 3Ø
	Ebox 3Ø
	Mbox 38
	Memory Subsystem 48
	Front-End Subsystem 53
	I/O Subsystem 61
4	Previous context execute................63
	Appendix A A1

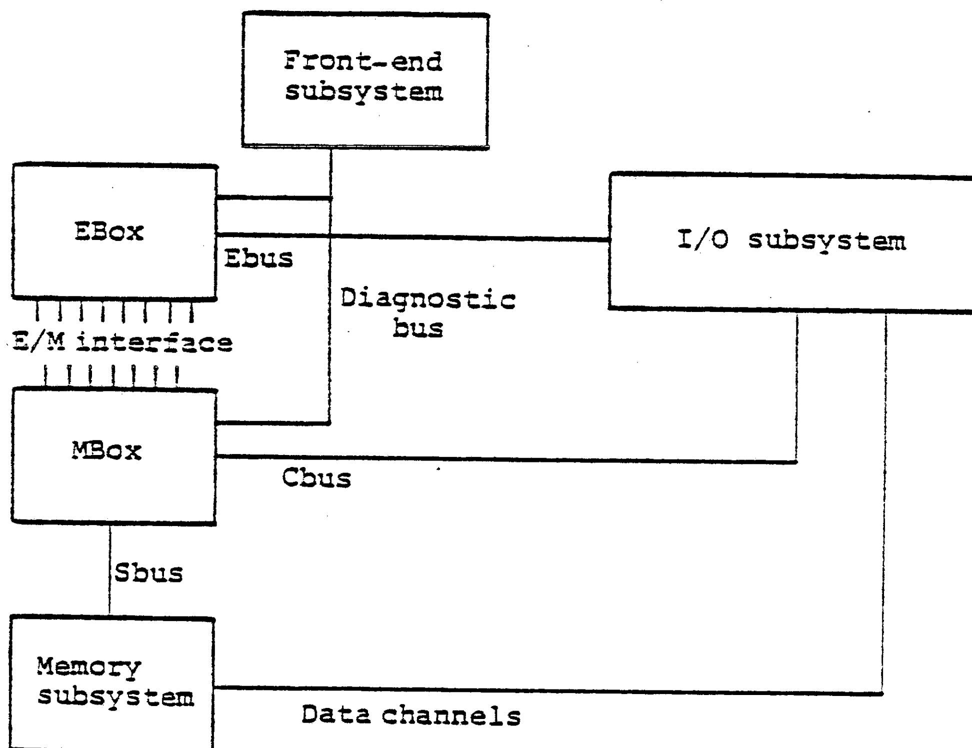
Document on the KL Processor

Introduction and KL Orientation

The KL processor is the basis of the high-end DECsystem-10 line (1080,1090) and the -20 series systems (2040, 2050). Each of these systems contains five subsystems:

- Ebox
- Mbox
- Memory
- Front End
- I/O devices

The diagram on the following page illustrates the basic configuration of the KL's subsystems.



KL Configuration

Figure 1

D7 0353

The Ebox (Execution box) is primarily concerned with the processing of program instructions. It fetches instructions from memory, computes effective addresses, and performs instruction actions. Additionally, the Ebox controls all devices by transmitting control information through the Ebus, and in turn receiving interrupts and device status along the same route. Finally, the Ebox controls data transfers between devices and memory for those devices not using a data channel.

The Mbox is responsible for coordinating physical memory requests. For instance, the Mbox must service all Ebox memory requests. Moreover, on DECSYSTEM-20 and 1090 systems the data channels are connected to the Mbox rather than being hooked to physical memory. Aside from its function in handling physical-memory requests, the Mbox has two related and significant jobs. First, the Mbox is the only system component that translates virtual addresses into physical addresses. Second, the Mbox contains and controls the cache memory.

The front-end subsystem comprises the PDP-11, associated -11 devices, and the DTE20 (which interfaces the -11 to the -10's Ebus). At the very least the -11 is responsible for overseeing operation of the KL processor. This responsibility extends to requiring the -11 to initialize the -10's memory and micromemories during bootstrap. Support of the operator's terminal is associated with these tasks. Additionally, DECSYSTEM-20s place all unit record and communications equipment under control of the front end -11, or of other -11s attached to a DTE.

The I/O subsystem includes all I/O devices that are controlled directly by the KL-10. Such devices invariably include disk controllers and tape controllers. Additionally, DECsystem-10s place unit record equipment and DECtapes in the I/O subsystem. (In other systems, such devices belong to the front end -11.) To provide this support, -10s need an additional device called the DIA.

Finally, the memory subsystem contains physical core memory. (It does not include the cache; cache is located in the Mbox.)

All KL-based systems contain these five subsystems. However, the internals of a particular subsystem might vary with the type of system. For instance, a 1080 Mbox will have cache while 2040 Mboxes do not. In the interest of clarifying the distinctions between the different systems, each section of this document will describe the appearance of the subsystems for each type of CPU.

Here is a summary, by subsystem, of optional KL-based system components.

Mbox	1080	1090	2040	2050
Cache	Y	Y	N	Y
Internal channels	N	Y	Y	Y

Memory subsystem	1080	1090	2040	2050
Internal memory	N	N	Y	Y
DMA	Y	Y	N	N
External channels	Y	S*	N	N

* Sometimes

Front end	1080	1090	2040	2050
Unit record equipment	N	N	Y	Y
Communications gear	N	S*	Y	Y

* Sometimes

I/O subsystem	1080	1090	2040	2050
DIA	Y	Y	N	N

The Ebox is substantially the same for all systems although the microcode will differ.

Paging on a KL Processor

This section describes the different types of paging available on KL processors. Section 2.1 concerns so-called KI-style paging, which is the scheme implemented on KL-10 processors (1080, 1088, 1090, 1099). Section 2.2 explains KL-style paging as implemented on KL-20 processors (2040, 2050).

Before discussing paging, it would be well to quickly review KL address management. This discussion frequently refers to the KL subsystems described in Section 1, and you might find it useful to consult Figure 1 as needed. Another available aid is Appendix A, which contains a glossary of commonly used paging terms.

Three different types of address are possible: physical, executive virtual, and user virtual. Physical addresses are 22 bits long and denote a word in the physical address space. The physical address space can contain as many as four million words. The average programmer rarely encounters physical addressing, but a study of the KL requires a knowledge of where physical addresses are used. There are four circumstances that deserve attention:

1. All requests to the memory subsystem must take the form of physical. Thus any request made by the Mbox using the Sbus must have been translated, by the Mbox, to a physical address. Also, any transfer involving an external data channel has to be initiated in terms of physical addresses.

2. Certain address inputs to the Mbox are expressed as physical addresses. Most significant are requests for (Cbus) transfers between RH20 controllers and the Mbox When a monitor program needs to initiate disk I/O, for example, the monitor must convert the address of the I/O buffer from virtual to physical before the transfer is started. The channel (i.e., the Cbus) then controls the passage of data from the physical addresses specified. Note that the treatment of internal data channels is thus logically consistent with that of external channels: both types require physical addresses.

3. A tiny subset of Ebox-to-Mbox requests is expressed in terms of physical addresses. The only physical Ebox requests are several (but not all) operations originating in the front-end subsystem.

4. Finally, most diagnostic-bus communication involves physical addresses.

Another class of address is that of exec-virtual. This address is 18-bits long and is converted (by the Mbox) into a 22 bit physical address before it is sent to the memory. An address

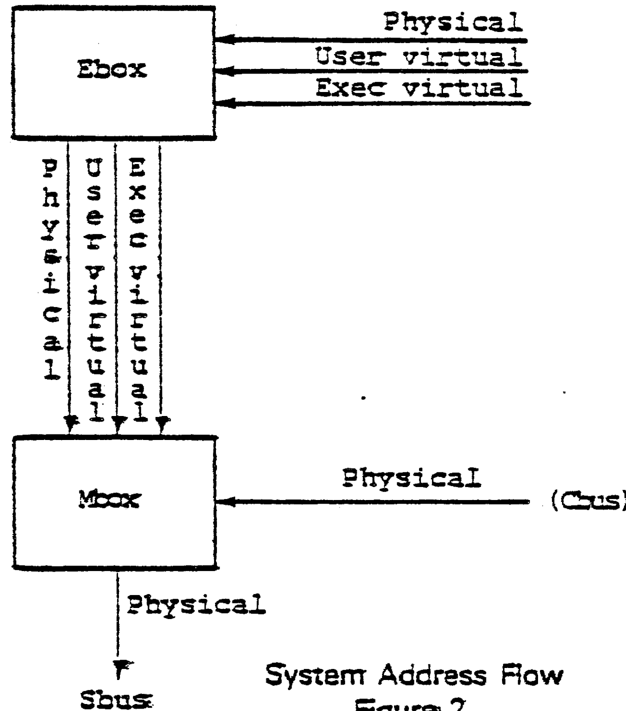
reference from an instruction is treated as exec-virtual if it originated in an instruction being performed while the processor is in an exec mode (either kernel or supervisor). The translation from exec-virtual to physical address is described in Section 2.1 for -10s and 2.2 for -20s.

Many I/O requests are expressed in terms of exec-virtual addresses. The only requests that are not exec-virtual are data-channel requests (which are physical addresses, as described above) and some real-time transfers (which could take place in I/O mode). An example of the use of exec-virtual addresses to accomplish I/O is monitor programming of DECtape, paper tape, or unit record equipment.

More importantly, a large proportion of instruction references are exec-virtual. Specifically, any instruction executed in the monitor requires at least one, and frequently more, exec-virtual memory reference. Consider the fact that instructions to be executed are fetched from the location pointed to by the processor PC-word. The PC-word contains an 18-bit counter, and this counter always points to a virtual address. The address will be treated as exec-virtual when the processor is in an exec mode and user-virtual when the processor is in user mode. Therefore, fetching an exec instruction requires an exec-virtual - to - physical translation. Of course, many instructions cause other memory references, thus adding to the total number of translations that must be made.

The third and last class of address is user virtual. User-virtual addresses are 18-bits long (like exec-virtual), and also require translation to physical addresses before memory can be read or written. Programs running in a user mode (either public or concealed) use user-virtual addressing. The translation of a user-virtual address to a physical address is described in Section 2.1 for -10s and 2.2 for -20s.

Of these three types (physical, exec-virtual, and user-virtual), exec and user-virtual addresses are the types most frequently encountered by the system programmer. Any virtual request must be translated into a physical address by the Mbox.



System Address Flow
Figure 2

It should be noted that an Ebox-based memory request might be any of the three types of address. The address will rarely be physical, but occurrences of exec-virtual and user-virtual requests are frequent. The type of address used depends on the circumstances of the request. For example, suppose that the Ebox has just finished processing an instruction. It must now read a new instruction from memory. The address of the new instruction is found in the processor's PC (Program Counter). Suppose the PC holds the number 001401. In this case, the Ebox must request the contents of address 001401 from the Mbox.

But what kind of address is 001401? It cannot be physical, if for no other reason than because physical addresses have 22 bits and the PC has only 18 address bits. Therefore the address must be either user-virtual or exec-virtual, but which?

The answer depends on the processor's mode when the instruction fetch is done. If the processor is in exec mode (as reflected by PC bit 5 being 0) then 001401 must be treated as an exec address. On the other hand, if a user program is being run then PC-bit 5 is 1, indicating that the processor is in user mode. In that event, 001401 is a user address.

At the hardware level, the Ebox makes its request by sending the address (001401) to the Mbox across the E/M interface. Additionally, the Ebox must tell the Mbox what kind of addressing is needed. (The Mbox cannot determine this directly because PC bit 5

determines processor mode, and the PC is in the Ebox.) This is accomplished by the Ebox sending an additional signal to the Mbox specifying the address mode.

Another example involves an instruction like "ADD 5,1700". The Ebox must obtain the contents of 1700 to perform the addition. As before, this requires that the Ebox set up the address (1700) on the E/M interface. And Ebox must again inform the Mbox of the desired addressing mode scheme (user or exec). The type of addressing is still dictated by PC bit 5. Thus a user program executing the instruction will cause 1700 to be treated as a user-virtual address, while the same instruction performed in the monitor would make 1700 be an exec address.

Amid this sea of confusion there is an island of fact: the only part of the system that converts one type of address to another is the Mbox. If the Ebox supplies a user-virtual address to the Mbox, that is because the Ebox found the user-virtual address elsewhere. Similarly, if the Ebox feeds the Mbox a physical address, then the Ebox was given a physical address by something else. The Ebox cannot take a virtual address and translate it, for that is the sole province of the Mbox.

Cache memory is a different matter altogether and has no direct bearing on the paging concepts just described. Cache provides a means of eliminating roughly 90% of the possible references to physical core, thus speeding up CPU operation by a substantial margin. Please note that the cache contents are indexed by physical address, therefore cache is only accessed after a virtual address has been translated to the corresponding physical address.

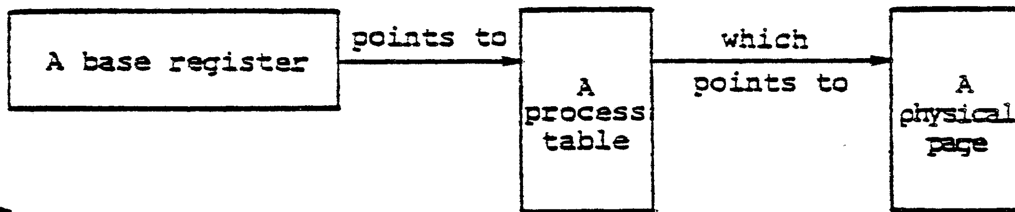
Section 2.1 describes the specifics of DECsystem-10 paging, while Section 2.2 provides information on DECSYSTEM-20 paging.

2.1 DECSYSTEM-10-STYLE PAGING ("KI-STYLE")

DECsystem-10 paging is the paging scheme supported by systems running TOPS-10 (1080, 1090).

Under KI paging, the processor has two "process tables". The User Process Table (UPT) controls the mapping of all user and some exec pages. The Exec Process Table (EPT) is used for most exec addresses.

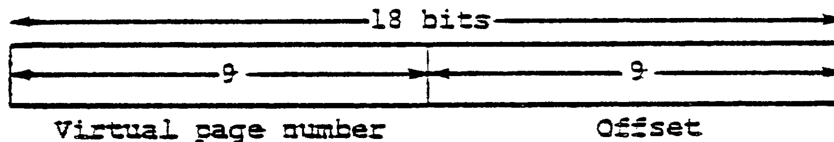
These tables are also referred to as the user/exec page maps or the user/exec page map pages.



KI-Style Paging
Figure 3

D7 0370

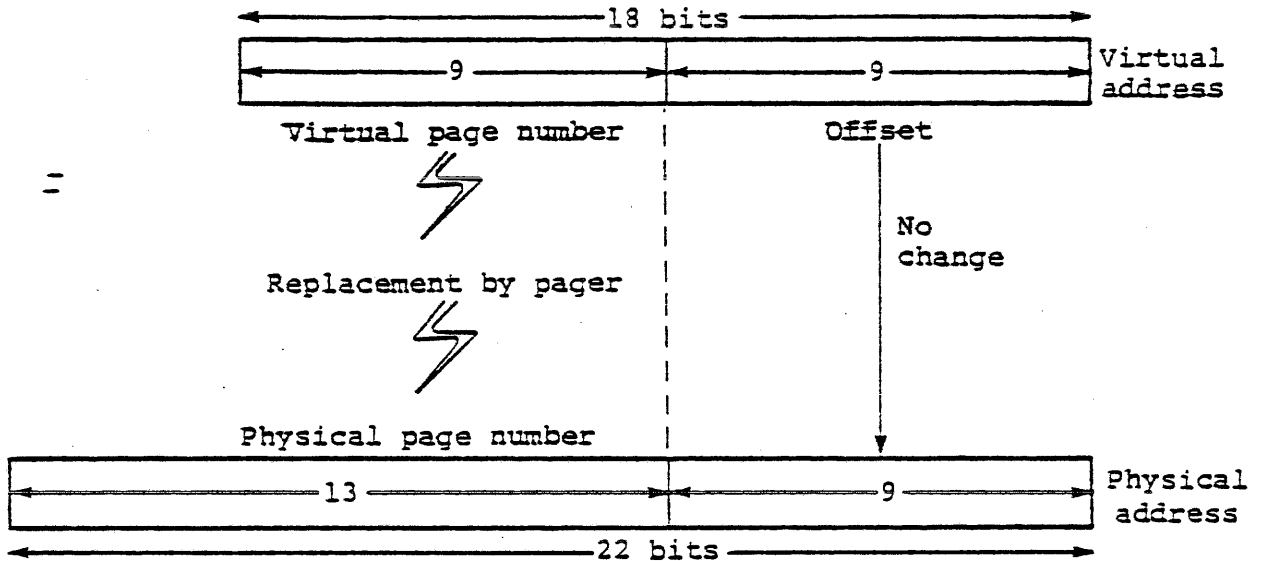
The basic mapping process involves translation of an 18-bit virtual address into a 22-bit physical address. In this process the virtual address is treated as a 9-bit virtual page number and an adjacent 9-bit "offset" into the page.



Virtual Address Structure
Figure 4

D7 0371

The mapping hardware removes the 9-bit virtual page number (the "VPN") from the address, uses these 9 bits to produce a 13-bit physical page number, and plugs the newly produced physical page number back into the address. This replacement procedure is the sole topic of section 2.1.



Address Translation

Figure 5

D7 0372

Here is a detailed presentation of the KI-style process tables.

USER PROCESS TABLE

Address	Contents	
0	USER PAGE 0	USER PAGE 1
377	USER PAGE 776	USER PAGE 777
400	EXECUTIVE PAGE 340	EXECUTIVE PAGE 341
417	EXECUTIVE PAGE 376	EXECUTIVE PAGE 377
420	RESERVED	
421	USER ARITHMETIC OVERFLOW TRAP INST	
422	USER STACK OVERFLOW TRAP INSTRUCTION	
423	USER TRAP 3 TRAP INSTRUCTION	
424	MUUO STORED HERE	
425	MUUO OLD PC WORD	
426	MUUO PREVIOUS CONTEXT WORD	
427	RESERVED	
430	KERNEL NO TRAP MUUO NEW PC WORD	
431	KERNEL TRAP MUUO NEW PC WORD	
432	SUPERVISOR NO TRAP MUUO NEW PC WORD	
433	SUPERVISOR TRAP MUUO NEW PC WORD	
434	CONCEALED NO TRAP MUUO NEW PC WORD	
435	CONCEALED TRAP MUUO NEW PC WORD	
436	PUBLIC NO TRAP MUUO NEW PC WORD	
437	PUBLIC TRAP MUUO NEW PC WORD	
440–447	AVAILABLE TO SOFTWARE	
500	PAGE FAIL WORD	
501	PAGE FAIL OLD PC WORD	
502	PAGE FAIL NEW PC WORD	
503	RESERVED	
504–505	USER PROCESS EXECUTION TIME	
506–507	USER MEMORY REFERENCE COUNT	
510–577	RESERVED	
600–677	AVAILABLE TO SOFTWARE	

EXECUTIVE PROCESS TABLE

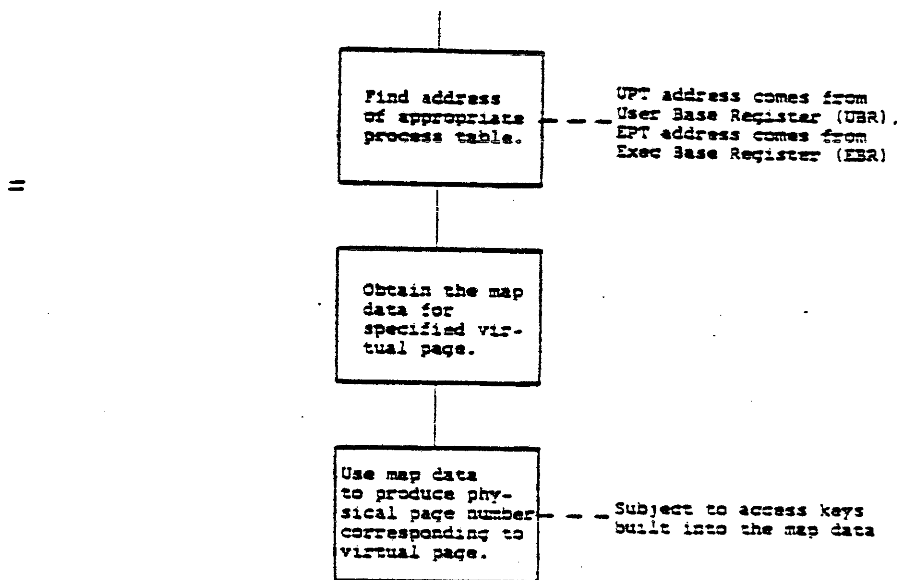
Address	Contents	
0–37	EIGHT CHANNEL LOGOUT AREAS EACH: 0 INITIAL CHANNEL COMMAND 1 GETS CHANNEL STATUS WORD 2 GETS LAST UPDATED COMMAND 3 RESERVED	
40–41	RESERVED	
42–57	STANDARD PRIORITY INTERRUPT INSTRUCTIONS	
60–63	FOUR CHANNEL BLOCK FILL WORDS	
64–137	RESERVED	
140–177	FOUR DTE20 CONTROL BLOCKS	
200	EXECUTIVE PAGE 400	EXECUTIVE PAGE 401
377	EXECUTIVE PAGE 776	EXECUTIVE PAGE 777
400–417	AVAILABLE TO SOFTWARE	
420	RESERVED	
421	EXECUTIVE ARITHMETIC OVERFLOW TRAP INST	
422	EXECUTIVE STACK OVERFLOW INSTRUCTION	
423	EXECUTIVE TRAP 3 TRAP INSTRUCTION	
424–507	RESERVED	
510–511	TIME BASE	
512–513	PERFORMANCE ANALYSIS COUNT	
514	INTERVAL COUNTER INTERRUPT INSTRUCTION	
515–577	RESERVED	
600	EXECUTIVE PAGE 0	EXECUTIVE PAGE 1
657	EXECUTIVE PAGE 336	EXECUTIVE PAGE 337
760–777	AVAILABLE TO SOFTWARE	

KI-Style Process Table Configuration

Figure 6

D7 0373

When a virtual address is received by the Mbox, the following procedure is used to translate the address. *



KI-Style Paging Algorithm

Figure 7

Here is a detailed examination of each of the three steps. Keep in mind that the ultimate goal of these steps is to produce a 13-bit physical page number.

Find Address of Appropriate Table

One of the two process tables contains the information needed to translate the address. Each process table is pointed to by a base register. In the case of the UPT the User Base Register (UBR) is used, while the EPT is pointed to by the Exec Base Register (EBR). The EBR is loaded when the system is started and never changed again. Conversely, the UBR is reloaded every time a job context switch takes place.

Both the UBR and EBR hold the (13-bit) physical page number of the page containing the corresponding process table.

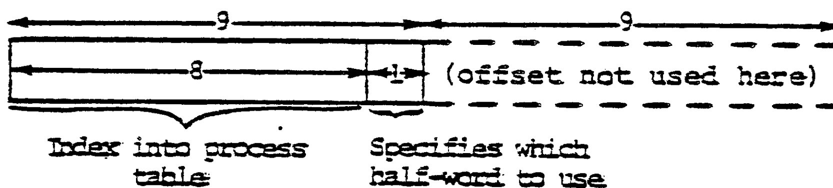
* The algorithm shows the complete logical paging process. The hardware generally takes shortcuts in the mapping process. However, these shortcuts involve the hardware page table (Section 3.2.1) and cache (Section 3.2.2).

Obtain Map Data for Specified Virtual Page

For this step the virtual page number is used as an index into the appropriate process table. The exact use of the virtual page number depends on whether the address is user or exec, and on what part of the virtual-address space the virtual address is in. The breakdown is as follows:

2.1.2.1 All User Addresses

The 9-bit VPN is treated like this:



VPN Breakdown

Figure 8

07 0375

The 8-bit field selects the process table word that holds the map data. Since the map information for a given virtual page occupies 18-bits, each process table word contains information about two pages; the desired half-word is chosen by the rightmost bit of the VPN. If the bit is 0 the left half word is used, while 1 implies the right half-word.

For example, suppose the address is user-virtual 040003. This can be interpreted as a request for word 003 of virtual page 040. The virtual page number breaks down like this:

```
040(8)=      0 0 0 1 0 0 0 0 0

    UPT word...     020(8)     0  ...left half
```

which means that the 18-bit map data are in UPT word 020, left half.

Similarly, map data for user address 277040 are in UPT word 137, right half.

EXEC Addresses Between 000000 and 337777 (Exclusive of ACs)

The virtual page number is dissected as before. However, the desired map data are in the EPT, not the UPT. Additionally, the treatment of the offset is slightly different. To select the proper EPT word, add 600 to the offset to produce an index into the table. Then select the proper half-word using the low order bit.

For example, exec address 002741 would be mapped using the left half of EPT word 601, as follows:

```
002(8)=        0 0 0 0 0 0 0 1 0
               \_______________/ ↕
                  001(8)         0
                + 600(8)
                = 601(8)         LH
```

2.1.2.3 Exec Address Between 400000 And 777777

These are handled exactly as user addresses are, except that the map data are in the EPT; the offset is not altered before use as an index into the process table.

For example, exec address 403375 is mapped through EPT word 201, right half.

```
403(8)=        1 0 0 0 0 0 0 1 1
               \_______________/ ↕
                   201(8)        1
     leads to      201           RH
```

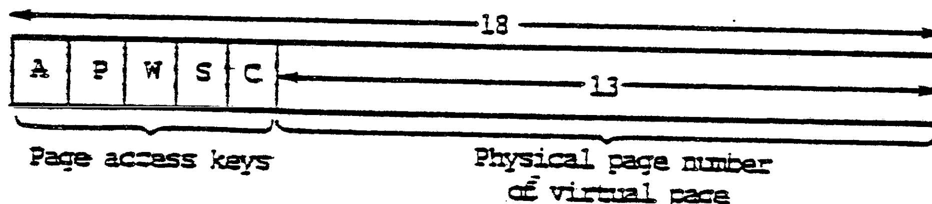
4. Exec addresses between 340000 and 377777 -- add 220 to the offset and read the desired word from the UPT. Unlike any other exec addresses, this range is mapped through the UPT, not the EPT.

An instance of this is exec address 340040. It would be mapped through UPT word 400, left half.

```
340(8)=        0 1 1 1 0 0 0 0 0
               \_______________/ ↕
                   160(8)        0
                 + 220(8)
                 = 400(8)        LH
```

Produce Physical Page Number of Desired Virtual Page

The process table data found during step 2 look like this:



KI Map Data

Figure 9

The page access keys have these meanings:

A — 0 implies that the page is inaccessible. References to such a page cause a page fault.

P — 0 implies that the page is concealed, while 1 implies the page is public.

W — 0 indicates that the page cannot be written. Attempts to write result in a page fault.

S — Available to the software. TOPS-10 uses this bit in conjunction with VM paging.

C — 0 indicates that the contents of this page may not be placed in cache.

Of course the physical page number is the 13-bit quantity we've been seeking. This field simply replaces the 9-bit virtual page number in the original address, thus providing the final physical core address.

2.2 DECSYSTEM-20-STYLE PAGING ("KL-STYLE")

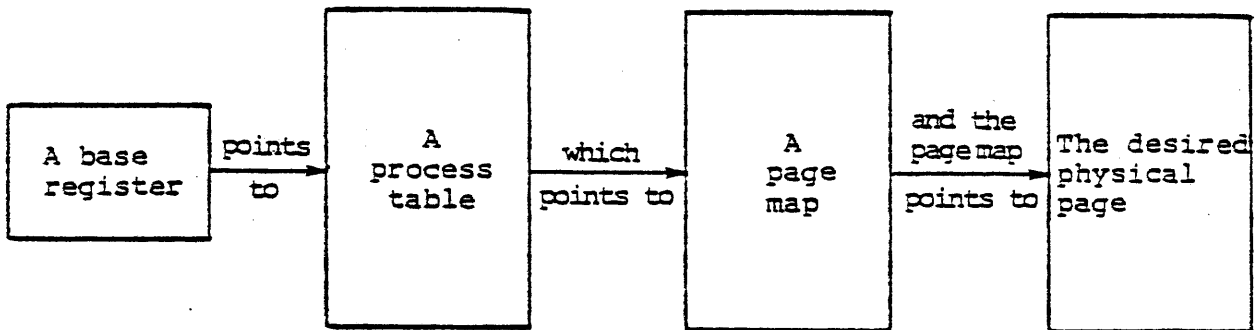
The DECSYSTEM-20 paging scheme is conceptually the same as that of the DECsystem-10 in that both map the limited user address space into a much larger memory pool. in the -10, the user space is distributed primarily within physical core, with some pages being mapped instead to a a page on disk. The -20, however, is more general: not only can user space comprise a portion of core, but it can also map user pages to parts of a file on disk. More generally still, the -20 system permits efficient sharing of both file and core pages between process; the -10 shares only core.

The description in this section focuses on the behavior of the KL20 paging microcode. However, the narrative will occasionally touch on TOPS20'S use of various pointers; otherwise, it's hard to see how the different pointer types are used.

Your understanding of the paging process will be helped by realizing the role of the paging microcode. The microcode completely handles requests for in-core pages; any other condition requires action on the part of the monitor. These other cases include reference to a disk-resident page, attempts to use non-existent pages, and troubles in the paging hardware. Any of these situations cause the microcode to issue a page fault, which is a hardware trap that terminates the current operation and gives control to the monitor.

As with so-called KI-style paging, KL paging involves the replacement of a virtual page number with a physical page number. There are two tables called the "user process table" (UPT) and "exec process table" (EPT), each one page long. These are analogous to the KI-style "user process table" and "exec process table". The user and exec process tables can be anywhere in physical core, with their addresses held in the User Base Register (UBR) and Exec Base Register (EBR) respectively.

Now for a significant difference: under KL paging, the UPT and EPT do not hold relocation information! Rather, word 440 of each process table contains a pointer which, when evaluated, will lead the hardware to a "page map"; this page map contains the mappings for specific virtual pages. This scheme is reflected by the following diagram:



Simplified KL Paging

Figure 10

You might wish to compare the KI process tables (figure 6) with the KL process tables shown here.

USER PROCESS TABLE

Address	Contents	
0–377	AVAILABLE TO SOFTWARE NOTE: : Items in parentheses and at the right in double entries apply to the unextended KL10	
400–417	RESERVED	
420	ADDRESS OF LUUO BLOCK	
421	USER ARITHMETIC OVERFLOW TRAP INST	
422	USER STACK OVERFLOW TRAP INSTRUCTION	
423	USER TRAP 3 TRAP INSTRUCTION	
424	FLAGS | MUUO OP AC	RESERVED
425	MUUO OLD PC	MUUO STORED HERE
426	E OF MUUO	MUUO OLD PC WORD
427	MUUO PREVIOUS CONTEXT WORD	
430	KERNEL NO TRAP MUUO NEW PC WORD	
431	KERNEL TRAP MUUO NEW PC WORD	
432	SUPERVISOR NO TRAP MUUO NEW PC WORD	
433	SUPERVISOR TRAP MUUO NEW PC WORD	
434	CONCEALED NO TRAP MUUO NEW PC WORD	
435	PUBLIC TRAP MUUO NEW PC WORD	
436	PUBLIC NO TRAP MUUO NEW PC WORD	
437	PUBLIC TRAP MUUO NEW PC WORD	
440	USER SECTION POINTER	
441–477	RESERVED	
500	PAGE FAIL WORD	
501	PAGE FAIL FLAGS	PAGE FAIL WORD
502	PAGE FAIL OLD PC	PAGE FAIL OLD PC
503	PAGE FAIL NEW PC	PAGE FAIL NEW PC
504–505	USER PROCESS EXECUTION TIME	
506–507	USER MEMORY REFERENCE COUNT	
510–577	RESERVED	
–777	AVAILABLE TO SOFTWARE	

EXECUTIVE PROCESS TABLE

Address	Contents
0–37	EIGHT CHANNEL LOGOUT AREAS EACH: 0 INITIAL CHANNEL COMMAND 1 GETS CHANNEL STATUS WORD 2 GETS LAST UPDATED COMMAND 3 RESERVED
40–41	RESERVED
42–57	STANDARD PRIORITY INTERRUPT INSTRUCTIONS
60–63	FOUR CHANNEL BLOCK FILL WORDS
64–137	RESERVED
140–177	FOUR DTE20 CONTROL BLOCKS
200–417	AVAILABLE TO SOFTWARE
420	
421	EXECUTIVE ARITHMETIC OVERFLOW TRAP INST
422	EXECUTIVE STACK OVERFLOW TRAP INSTRUCTION
423	EXECUTIVE TRAP 3 TRAP INSTRUCTION
424–437	RESERVED
440	EXECUTIVE SECTION POINTER
441–477	RESERVED
500–507	RESERVED
510–511	TIME BASE
512–513	PERFORMANCE ANALYSIS COUNT
514	INTERVAL COUNTER INTERRUPT INSTRUCTION
515–577	RESERVED
600–777	AVAILABLE TO SOFTWARE

KL-Style Process Table Configuration

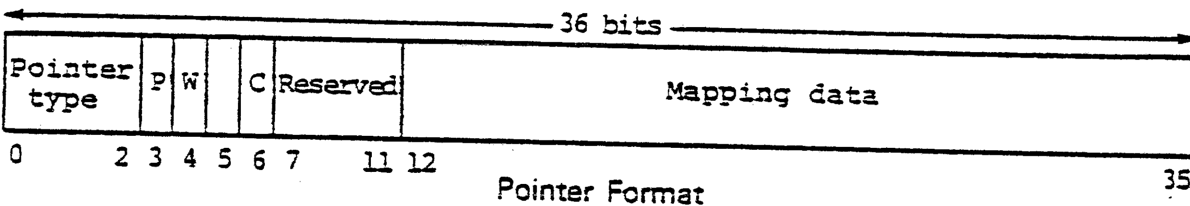
Figure 11

07 0380

The page map entries differ markedly from the pointers used in KI-style page maps. Under KI paging the process table contains halfword entries, each of which holds a physical page address and five access keys (see figure 9). On KL20 processors the page map entries are each one word long and hold a pointer which must be evaluated to determine the final memory address. This address is frequently the page number of a page in core, in which case the microcode performs the substitution. In other cases the memory address is a disk address, which results in the microcode turning over the translation process to the monitor.

To reiterate, there are two different sets of pointers involved in this process. The first type of pointer (called the "section pointer") effects the link between the process table and the page map. The second type (the "map pointer") resides in the page map and points, directly or otherwise, to a specific physical page. This page may either be on disk or, more commonly, in core.

Both pointer types have the same format:



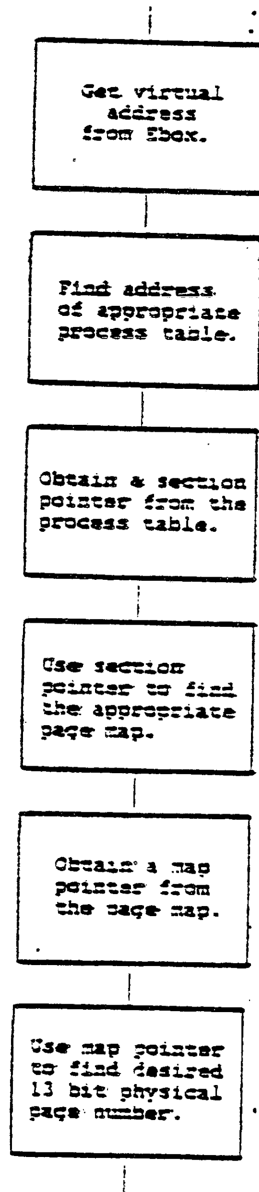
Pointer Format

Figure 12

D7 0378

Despite this similarity, the two pointer types function somewhat differently and serve different purposes. This distinction will be dealt with in succeeding sections, as section pointers are treated in section 2.2.1 and map pointers are explained in section 2.2.2.

The entire mapping process can be treated as a series of discrete steps, as follows:



KL-Style Paging Algorithm

Figure 13

37 0379

The following sections examine each of these steps in somewhat more detail.

1. Get virtual address from the Ebox

The virtual address arrives from the Ebox as 18 address bits plus a signal indicating whether the address is exec or user. Actually, nothing happens in this step other than hardware handshaking (wireshaking?); however, this is a good place for us to logically split the address into two halves. The high-order 9 bits, bits 18-26, are treated as the virtual page number, while the low order 9 bits, bits 27-35, are used as the offset into the page. The virtual page number supplies the pager with the information necessary to determine the address of the page corresponding to the specified virtual page. Once determined, the physical page number replaces the virtual page number in the original address. The 9 bit word index will never be changed by the mapping process. This is illustrated in figure 5.

2. Find address of appropriate process table

This is handled the same way it was on the KI. Please see section 2.1.1.

3. Obtain section pointer from process table

The section pointer is held in location 440 of the process table.

4. Use section pointer to find the appropriate page map

This operation begins at a process table. For a user address this is the UPT, which is pointed to by the User Base Register (UBR). For an exec address, the EPT is used. The EPT is pointed to by the Exec Base Register (EBR).

Once the process table is found, the pager reads word 440, the USECT (or ESECT) word. The word contains a "section pointer" which eventually produces the address of the page map.

There are four different kinds of section pointers. They are treated in section 2.2.1.

5. Obtain a map pointer from the page map

The preceding step provided the address of a page map. Page maps contain 512 one-word entries that specify the physical address of a single memory page belonging to a process's virtual address space. Usually the page is in core, though it's sometimes on disk. Rarely the reference is illegal and corresponds to nothing, in which case the issuing process is in error.

6. Use map pointer to find desired memory

Page map pointers (hereafter referred to as map pointers) have the same format as section pointers, but are used somewhat differently. There are four types: no-access, immediate, shared, and indirect. They are discussed in detail in section 2.2.2.

Please keep in mind that the ultimate goal of this step is either to determine the 13 bit physical page number corresponding to the virtual page specified in the original address, or to produce a disk address that the monitor will use to bring in the needed page.

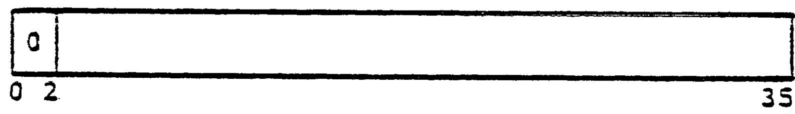
Section Pointers

Keep in mind that the section pointer's purpose is to point to a page map. Evaluation of the pointer will eventually produce a 13 bit physical page number. In that page is the page map.Note that only 13 bits are needed to find the page map: all page maps start on a physical page boundary, and there are at most (2)13 physical pages of memory.

The treatment of this pointer varies depending on the first 3 bits:

if the first 3 bits are...	then the pointer is...
000	no-access
001	immediate
010	shared
011	indirect

2.2.1.1 No-access section pointers



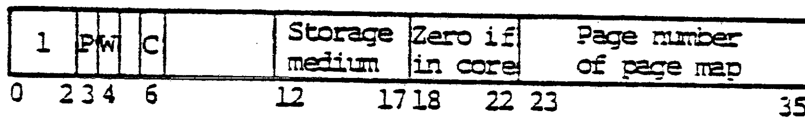
No-Access Section Pointer

Figure 14

If a memory reference leads to a no-access section pointer, then the reference is illegal. The result is a page fault, and further processing of the memory request is determined by the page fault handling software.

The capability exists primarily for the sake of generality; recall that there are map pointers in addition to section pointers, and that map and section pointers have the same format. As will be shown in section 2.2.2 there is need for no-access map pointers. Since they must be included, it was easiest to provide a section pointer that behaves the same way.

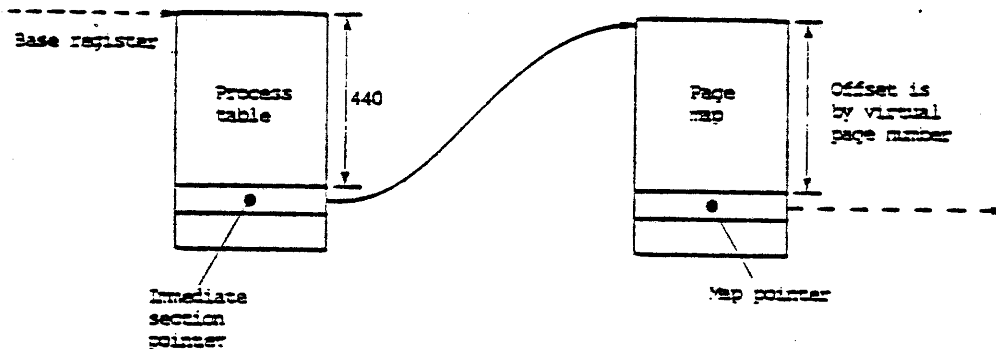
2.2.1.2 Immediate section pointers



Immediate Section Pointer Format
Figure 15

An immediate section pointer provides the address of a section's page map. The page map may be in core, in which case bits 12-17 of the pointer are zero. The page map might be on disk, however. This case is signalled by a non-zero value in bits 12-17, a condition that causes a page fault. The monitor then uses bits 12-35 as the disk address of the page map and reads it into core.

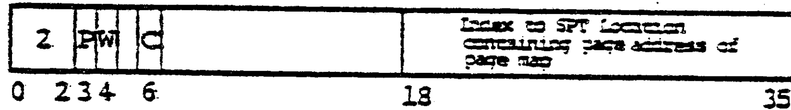
Assuming that the page map is in core (as indicated by bits 12-17=0), then the 13-bit physical page number of the page map is found in bits 23-35 of the pointer.



Immediate Section Pointer Structure
Figure 16

From the point of view of TOPS-2Ø, immediate section pointers exist for much the same reason that no-access section pointers exist; namely, as parallels to immediate map pointers.

2.2.1.3 Shared section pointers



Shared Section Pointer Format

Figure 17

In this case, the address of the desired page map is not built into the pointer. Instead, the page map's address is in the Shared Pages Table (SPT). The section pointer provides an index into the table, and the SPT location thus specified contains the desired 13-bit physical page number. The pointer need not contain the address of the SPT; the pager knows this independent of the pointer because the pager's SPT Base Register (SBR) was loaded long beforehand with the SPT's page address. The offset is found in bits 18-35 of the pointer.

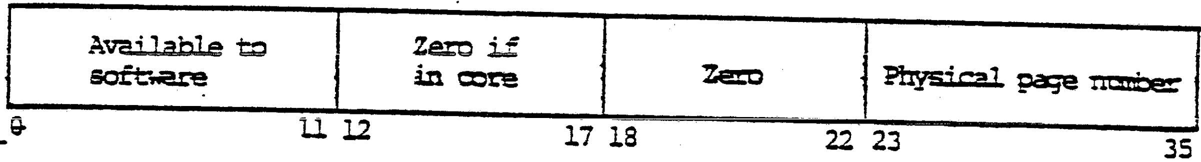
In other words, the page map's address is stored in an SPT word. The pager always knows where the SPT itself is, so the pointer only has to say which SPT word holds the data. The address is obtained by adding the 13-bit SBR to the offset from the section pointer thus:

```
  13-bit SBR        X X X X X X X X X X X X X Ø Ø Ø Ø Ø Ø Ø Ø Ø
+ 18-bit SPT index  Ø Ø Ø Ø X X X X X X X X X X X X X X X X X X
```

= 22-bit physical word address of the word that contains the page map address

It is from this address that the pager obtains the address of the page map.

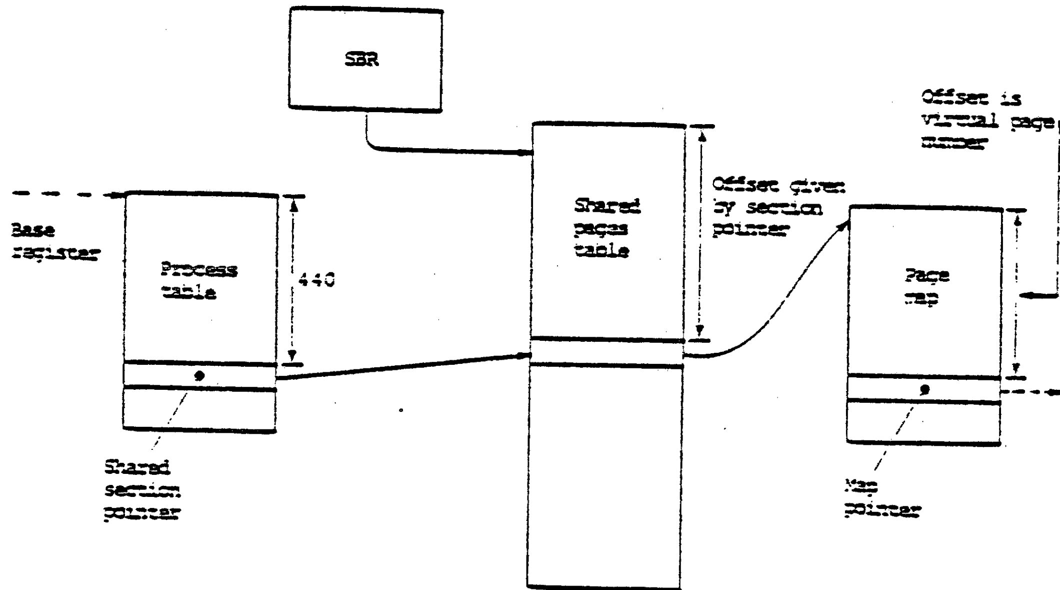
It should be stated that the SPT word consists primarily of the page map address, but not exclusively. The full format of the SPT word is:



SPT Word Format

Figure 18

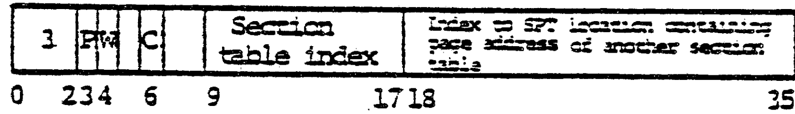
D7 0382



Shared Section Pointer Structure

Figure 19

2.2.1.4 Indirect section pointers



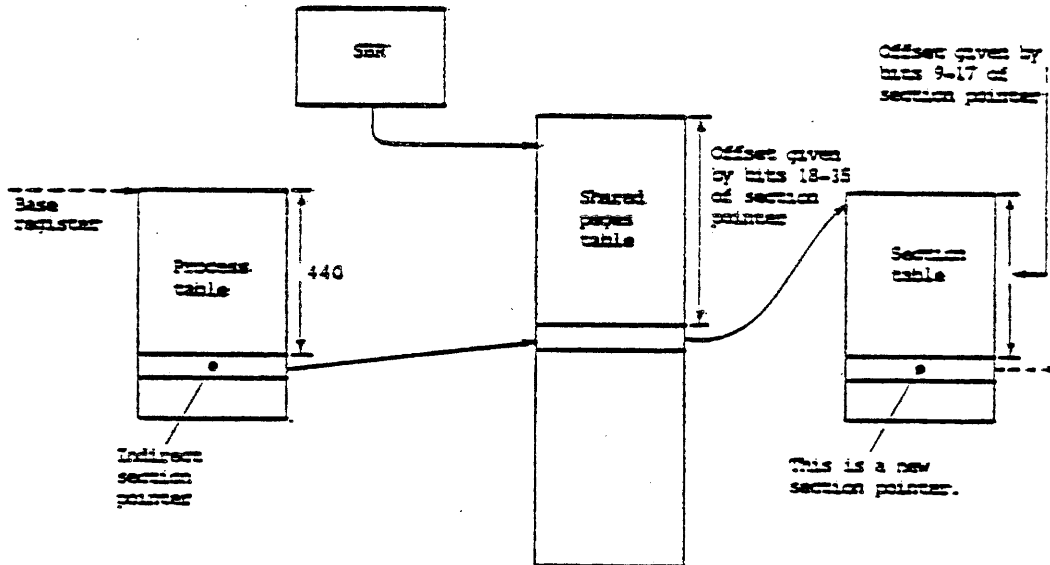
Indirect Section Pointer

Figure 20

D7 0385

Unlike either immediate or shared section pointers, indirect section pointers do not lead the pager directly to the address of a page map. Instead, an indirect section pointer results in acquisition of a new section pointer, which may in turn be no-access, immediate, shared, or indirect.

Once the pager has the indirect pointer, bits 18-35 furnish an index into the shared page table. In that location the pager finds the physical page number of a special table call a "section table." The section table may contain as many as 512 entries, each of which is a new section pointer. Bits 9 through 17 of the original section pointer furnish an index into the section table. The indicated location holds a new section pointer (no-access, shared, immediate, or indirect) which will be evaluated appropriately.



Indirect Section Pointer Structure

Figure 21

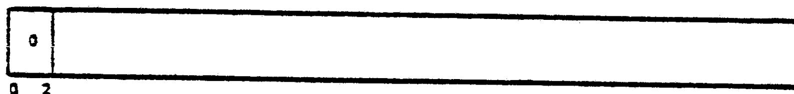
D7 0384

Map Pointers

As with section pointers, the treatment of this pointer varies depending on the first 3 bits:

if the first 3 bits are...	then the pointer is...
000	no-access
001	immediate
010	shared
011	indirect

2.2.2.1 No-access pointers



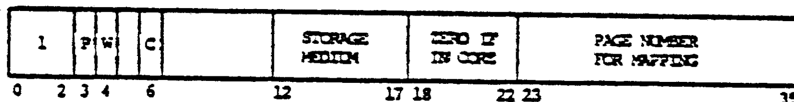
No-Access Map Pointer

Figure 22

This pointer indicates that the specified virtual page is not part of the requesting process. The result is a page fault.

No-access pointers are used to prevent a process from using illegal and unassigned pages.

2.2.2.2 Immediate map pointers

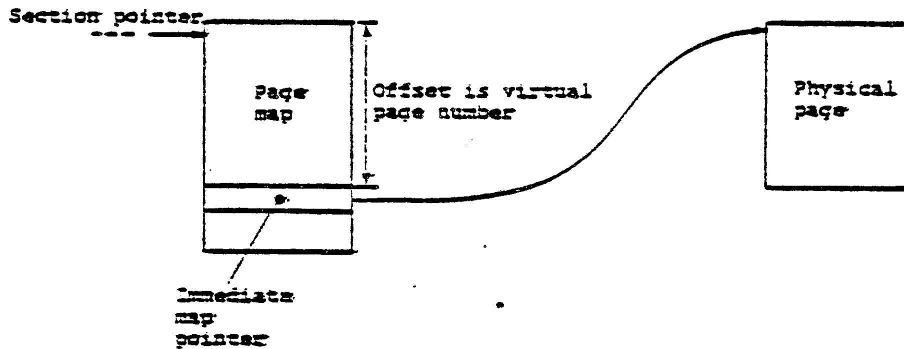


Immediate Map Pointer

Figure 23

A given page may reside either in core or on disk. If it's in core, pointer bits 12-22 are zero, and the page's physical number is held in bits 23-35 of the map pointer. These 13 bits are concatenated with the original 9 bit page index to provide the final physical address.

If the page is on disk instead, then bits 12-22 are non-zero. This condition forces the microcode to issue a page fault, in response to which the monitor uses bits 12-35 as the disk address of the desired page.



Immediate Map Pointer Structure

Figure 24

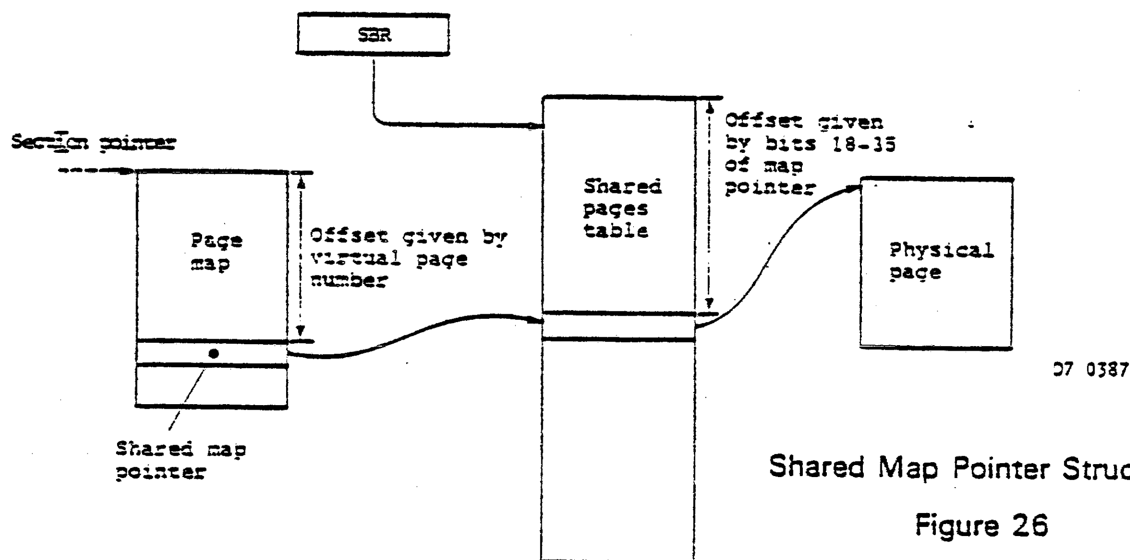
Immediate map pointers are used for private pages, i.e. pages belonging to exactly one process.

2.2.2.3 Shared map pointers

Shared Map Pointer

Figure 25

Shared map pointers provide an index into the system's SPT. The SPT location thus specified contains the 13-bit physical page number of the desired virtual page. (See the description of shared section pointers for more detail on this.)



Shared Map Pointer Structure

Figure 26

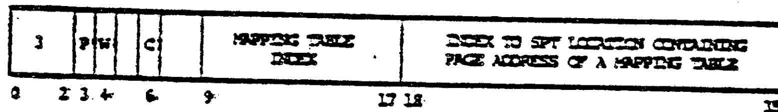
A shared map pointer is used for a page that belongs to several different processes. For instance, suppose that a particular page contains only executable code that is part of a compiler. Several different processes may be compiling at any given time, so the various page maps will each contain a shared map pointer to the shared page. By doing this, the system can swap out the page and painlessly inform all interested processes simply by changing the single SPT pointer. If immediate pointers were used instead of shared pointers, the system would be forced to find all page maps using that page and change them individually.

Another use of shared pointers arises from TOPS-2Ø'S treatment of disk I/O. When a program uses a page from a disk file, that page is considered shared between the program and the file; the file is viewed as a process. in brief, when a file is opened, its indeX Block (XB) is read into core, and its format is the same as a page map. Initially, all the XB pointers are immediate. Now suppose the user maps process page 5Ø to correspond to file page 2Ø. This results in the disk address of file page 2Ø being placed in the SPT. Then entry 5Ø of the user page-map and entry 2Ø of the file's XB are both changed to shared pointers so both now use the same SPT word. When the page is referenced, then the page is read in and the SPT entry is changed to a core address.

The beauty of this mechanism is that TOPS-2Ø uses the same copy of the XB for every process that uses the file. For instance, suppose a new process decides to use our file, specifically page 2Ø. The system need not read in the page again; instead, the new

process's page-map is given the shared pointer from the XB. When that pointer is used, the in-core copy of the page is automatically referenced, thanks to the information in the SPT.

2.2.2.4 Indirect map pointer



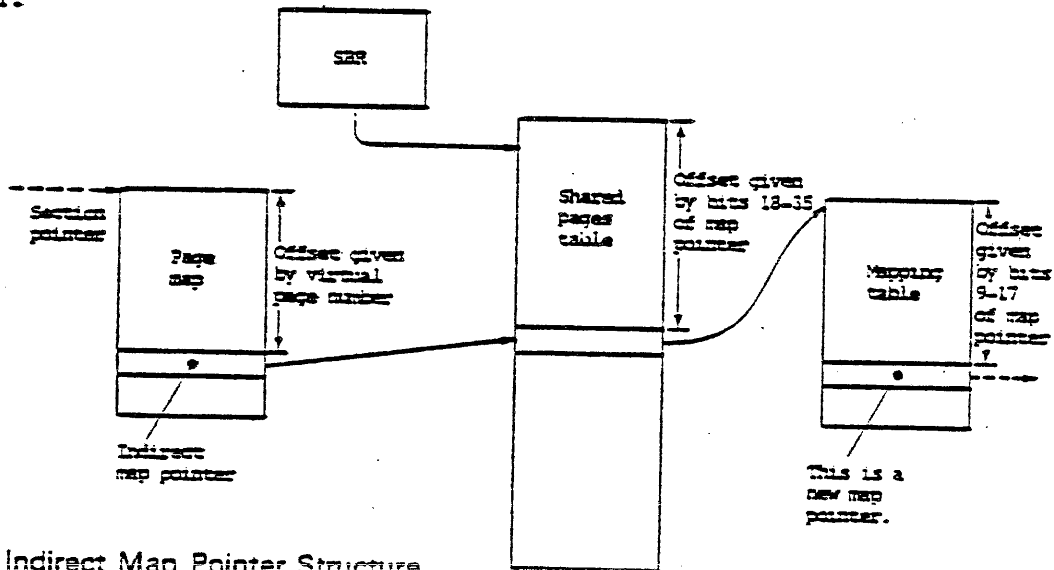
07 0400

Indirect Map Pointer

Figure 27

The pointer's SPT index directs the pager to an SPT entry, as was done with shared map pointers. In the case of indirect pointers, that entry contains the physical page address of a "mapping table".

The mapping table, in turn, contains up to 512 entries. One of these entries is selected by the 9 bit mapping table index (pointer bits 9-17). The resulting address contains a new pointer to be evaluated.



Indirect Map Pointer Structure

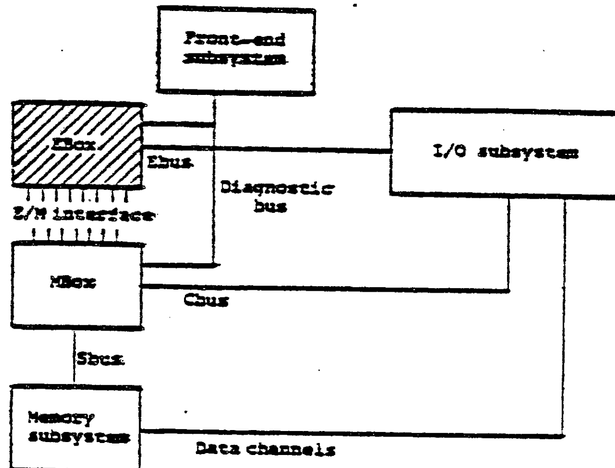
Figure 28

07 0338

There are several occasions for use of indirect pointers. One of these is the case of process A examining a page in process B. For the sake of argument, suppose A's page 100 is mapped to correspond to B's page 36. When the mapping occurs, entry 100 in A's page map becomes an indirect pointer. At the same time, an SPT word is loaded with the address of B's page-map; the SPT address of the word is put into the indirect pointer. Then bits 9-17 of the pointer are loaded with 36 (the page number of B's page).

KL Hardware

EBOX



The Ebox (Execution box) has two basic purposes. First, it must control execution of program instructions from memory. Second, it must interface non-channel I/O devices to memory.

A look at the KL configuration diagram in Figure 1 reveals that the Ebox has three links to the outside world. These are the Ebus, the E/M interface, and the diagnostic bus.

The Ebus links the Ebox to the system's I/O devices and the Front End. The Ebus carries all control information from the CPU to the output devices. Additionally, the Ebus transmits data to those devices that do not have a data-channel. Similarly, all devices send control information back to the CPU through the Ebus, and non-channel devices send data via the same route.

The E/M interface connects the Ebox to the Mbox. The information carried across this set of links is not normally of interest to the programmer, but typical signals include the 22 bits of an address desired by the Ebox, a signal indicating whether that address is virtual or physical, and signals describing the nature of an Mbox-detected page fault.

Finally, the diagnostic bus connects the Ebus to the console front end. The controlling PDP-11 uses this bus to bootstrap the KL and to gather information about the KL's health (or lack of it).

The Ebox consists of Emitter-Coupled Logic (ECL). This technology is used because of its high speed, which was gained at the expense of considerable power drain.

The following components are found in the Ebox of any KL-based system:

* arithmetic logic
* accumulator blocks
* Ebus control logic
* microcode and microprocessor
* Program Counter (PC)
* meters

The following sections of this chapter provide further details on the accumulator blocks, the microcode, the PC, and the KL meters.

Accumulator Blocks

The accumulator blocks are variously called AC blocks, fast ACs, Fast Memory (FM), or fast-memory blocks. In this text we'll refer to them as the AC blocks.

The KA10 processor was equipped with one set of 16 ACs. This meant that the monitor had to save these ACs whenever an interrupt or trap happened, since the monitor was forced to use the same accumulators as the user program.

To save this overhead the KI was given four blocks of 16 ACs. Block 0 was permanently assigned to any exec-mode program, but user programs could be given any of the four blocks. The TOPS-10 convention gave block 1 to the current user and left blocks 2 and 3 unused. (The unused blocks could, however, be used by real-time programs. If such a program wanted to use block 2, for instance, it could take advantage of its user I/O privileges to issue a DATAO PAG instruction to switch to the desired block.)

KL processors are built with eight sets of 16 accumulators. KL software can better use multiple AC blocks because the KL, unlike the KI, permits both user- and exec-mode programs to use any of the eight blocks. TOPS-10 assigns the blocks as follows:

0	most monitor operations (cf. 2 and 3)
1	current user program
2	scanner interrupt-level code
3	disk interrupt-level code
4-6	unused, available for realtime
7	reserved for use by the microcode

The TOPS-20 assignments are:

0	exec-mode programs
1	user-mode or previous context exec ACs
2-5	unused
6	KL paging
7	reserved for use by the microcode

You will periodically see references in DEC documentation to "previous context ACs" and "current context ACs." This distinction relates to the PXCT (Previous context eXeCuTe) hardware-instruction, which is similar in concept to the exec-mode XCT of the KI. PXCT is described in the Hardware Reference Manual.

Microcode

The KL's operation is governed by microcode. While there are several microstores in various parts of the machine, this discussion centers on the CRAM (Control RAM) and the DRAM (Dispatch RAM). These two RAMs (Random Access Memories) form the instruction execution logic. They are writeable semiconductor memories that are loaded by the console front end processor when the system is brought up.

The CRAM is 2048 words long, with each word 84 bits wide. It contains the microprogram that implements the DECsystem-10 or -20 instruction set, priority interrupts, etc. To give you an idea of the things controlled by the CRAM program, here is a list of some of the program modules of the microcode:

- Startup and stop handler - called at the end of each instruction to look for new PIs, etc.
- Effective address manager - computes an instruction's effective address using the instruction's I, X, and Y fields. (It does not, however, compute the corresponding physical address; virtual-to-physical mapping is done by the Mbox).
- Executor routine - contains the separate subroutines that implement specific -10 or -20 instructions (e.g. half-word moves and stack manipulation).
- Priority interrupt handler - checks if a PI has been requested. It is called from various points in EA calculation, and during some long instructions such as BLT (thus preventing lengthy operations from seriously delaying interrupt handling).
- Page fault handler - called when the Mbox can't resolve a virtual-to-physical address translation for some reason (e.g. the access-allowed bit being 0 for a virtual page).
- Input - output handler - generates any Ebus dialogue required by I/O instructions (DATAx, CONx).

The Dispatch RAM (DRAM) is 512 words long by 24 bits wide. The CRAM program uses the DRAM to decide how to process a given -10 or -20 instruction by obtaining, from the DRAM, the address of the specific CRAM routine that handles the instruction. For instance, suppose the current instruction is a MOVEI, for which the opcode is 201. The CRAM would first compute the effective address of the instruction (regardless of the fact that it is a MOVEI; the EA is the first step in processing any instruction). Then the CRAM obtains the address of the MOVEI subprogram from DRAM word 201, and jumps to it. Similarly, if the instruction was a MOVE (opcode 200), the dispatch address would come from DRAM word 200. In other words, the DRAM's entries are indexed by instruction opcode.

PC-Word

The KL PC word format is identical to that of the KI. It's described in the Hardware Reference Manual.

KL Clocks

The KL processor contains four programmable clocks. They are the:

- interval timer
- time base
- accounting meters
- performance analysis counter

The clocks are controlled by use of the three I/O device-codes TIM, MTR, and PAG. All hardware clock logic is ECL and is contained by the KL mainframe with the Ebox and Mbox.

The following presents a more detailed view of each of the KL clocks.

Interval Timer

The interval timer is similar in function to the DK10 clock. The timer can, at the programmer's option, interrupt on any desired PI level. The resolution of the clock is 10 microseconds, and the interval is programmable between 10 microseconds and 40.95 milliseconds.

The interval timer comprises a 12-bit counter and a period register. The period register is loaded by program control and reflects the desired frequency of interrupt. As mentioned earlier,

the frequency can range from 10 microseconds to 40.95 milliseconds in increments of 10 microseconds. If the program sets the period counter to four, the timer will go off every four increments, i.e. every 40 microseconds. When the timer goes off it causes a vectored priority interrupt to EPT word 514. The interrupt occurs on the clock's program-assigned PI level.

The interval timer is controlled and interrogated by use of the instructions CONO TIM and CONI TIM respectively.

Time Base

The time base is used to measure long-term elapsed time with one microsecond resolution. It offers accuracy of +-.005%, which amounts to a maximum of five seconds drift over 24 hours.

The time base is a 60-bit clock. Its length permits it to count intervals of 9140 years, after which it unfortunately overflows. The time base is incremented every microsecond. Theoretically, the 60-bit count could be maintained in the EPT and incremented there every microsecond. The increment isn't done this way; though, because this would result in a blizzard of memory references to the EPT. To hold down the overhead, the time base's count is incremented in a 16-bit register contained in the Ebox. Only when the count carries into the high order bit of this register is the count added to the 60-bit total in the EPT, after which the 16-bit register is cleared. The disadvantage of this technique is that the current time is not immediately available by looking at the EPT. For that reason, the system provides an instruction ("DATAI TIM") that produces the current time. The 60-bit quantity is held in EPMP words 510 and 511.

The time base is controlled and interrogated by the "DATAI TIM", "CONO MTR", "RDTIME" (read time-base doubleword), and "CONI MTR" instructions.

Accounting Meters

The accounting meters are, unsurprisingly, intended for job accounting. They consist of a Ebox busy meter and a memory cycle meter. As they can be programmed to shut off during PI processing, they offer an extremely reproducible way of billing users and comparing program performance.

These are two 60-bit meters. One of these, the Ebox busy meter, increments while the Ebox is executing microcode. The other meter, the Mbox cycle meter, counts the number of times the Ebox references memory through the Mbox.

The accounting meters are similar to the time base in that the Ebox contains two 16-bit registers (one for each meter), and the 60-bit values reside in memory. The Ebox busy meter is kept in UPT words 504 and 505, while the Mbox cycle meter occupies UPT words 506 and 507. In this connection, it is interesting to note that the time base is kept in the EPT (as it is a system-wide count), while the accounting meters are put in the UPT (since they contain information about a particular process).

The relevant hardware instructions are "CONO MTR", "CONI MTR", "DATAI MTR" (also called "RDEACT"), "BLKI MTR", ("RDMACT"), and "DATAO PAG" (which causes the meters to be saved on a context switch).

Performance Analysis Counter

The performance analysis counter is a built-in hardware monitor. It is designed to gather information that would be difficult or impossible to get using software probes. The performance analysis counter permits sophisticated system measurements to be made. It offers advantages not available with software monitoring. For instance, it does not interfere with system operation. Another feature is the ability to identify events happening at the sub-microsecond level.

This counter is a 60-bit counter that is maintained in EPT words 512 and 513.

Use of the counter, being rather complex, is not intended for the inexperienced. For that reason this document does not describe the counter in detail, but you might wish to note that the counter can measure combinations of the following conditions:

- User mode
- PI Level active
- cache miss
- cache writeback
- cache sweep
- Ebox-Mbox request
- microprogram event
- channel busy
- ECL probe input

The counter is controlled by "BLKO TIM" (or "WRPAE"), and "BLKI TIM" (or "RDPERF").

Complete details on the counter's use can be found in the hardware document entitled Meters-Unit Description, EK-MTR-UD-001.

*****

The following two sections describe TOPS-20 and TOPS-10 meter usage conventions.

TOPS-10 Meter Usage

The following description applies to the 6.03 monitor.

Interval timer	Provides the jiffy clock tick (every 60th of a second in 60 hz countries, every 50th of a second in 50 hz countries).
Time base	Records time of day, and optionally, job accounting if feature test switch FTEMRT is zero.

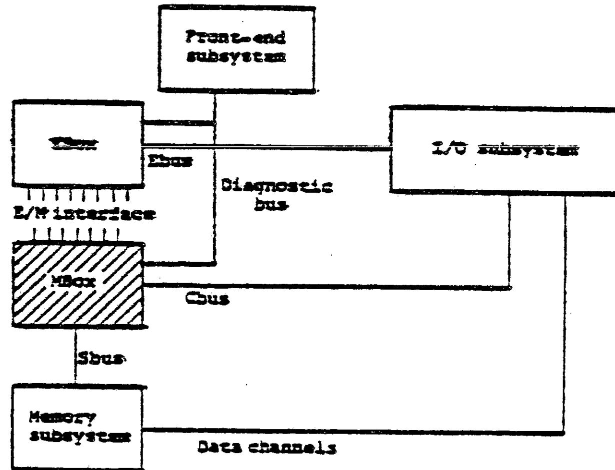
Accounting meters	Job accounting if feature test switch FTEMRT is non-zero.
Performance meter	Accessible using the PERF. monitor call.

TOPS-20 Meter Usage

As of the Release 2 monitor, the following usage prevailed.

Interval timer	Interrupts every millisecond. The interrupt handler maintains a count of the number of interrupts, and upon occurrence of the 2oth tick control is given to the monitor overhead cycle.
Time base	Used for time-of-day maintenance and job accounting.
Accounting meters	Not used
Performance meters	not used

MBOX



The job of the Mbox is to connect devices to memory. All KL-based processors require the Ebox to access memory through the Mbox. Additionally, some systems (the 1090, 2040, and 2050) replace the old DF10/10C data-channels with internal data-channels that are connected to the Mbox instead of a memory port.

The external Mbox connections shown in Figure 1 are the E/M interface, the Sbus, the Cbus (on some systems), and the diagnostic bus. The E/M interface was described in Section 3.1. The Sbus connects the Mbox to the memory subsystem. The Cbus links the Mbox to as many as 8 RH20 controllers and serves as a data-channel. The diagnostic bus permits the Front End to control Mbox operation and determine Mbox status.

The Mbox may contain the following components, depending upon the system;

- hardware page table
- user base register
- exec base register
- cache memory
- Cbus interface (internal channels)

Not all of these are present in any given system, as shown in this table. Any component not mentioned is present in all.

Mbox	1080	1090	2040	2050
Cache	Y	Y	N	Y
Internal channels	N	Y*	Y	Y

* but may also contain external channels

These components were described in the introduction to Chapter 2. Familiarity with these descriptions will help to avoid confusion resulting from use of terminology.

There are two ways for requests to enter the Mbox: through the Ebox/Mbox interface, or through the Cbus. Ebox requests are usually in the form of virtual addresses (either user or exec) which must be mapped onto physical addresses by the Mbox. Cbus requests specify physical addresses and thus bypass the pager. Cbus reads, however, may use the cache, thus involving the Mbox.

Hardware Page Table

Suppose the Ebox requests the contents of a particular virtual address. Theoretically, the Mbox must read a section pointer from a process table, probably find an SPT word,** and read a page-map entry, etc. If this procedure were used, the system would have to make many memory references whenever one was requested, thus drastically increasing access times.

The KI minimized this by having a 32-word associative memory. Built of semiconductor memory and internal to the CPU, it could hold the 32 most recently used page-map entries for rapid future access. The CPU only had to read the in-core page-map if the desired entry was not in the associative memory.

The KL has a similar but improved technique. Instead of an associative memory, the pager contains a hardware page table, which is a high-speed semiconductor memory that holds as many as 512 entries from the exec- and user-process tables (page maps for KL20s).

I'd be delighted to tell you that the virtual page number is used as an index into the hardware page-table. For instance, it would be nice if the mapping for user page 047 were found in hardware page-table word 047. Unfortunately, it is not. Moreover, it couldn't possibly be if the page-table size is 512 entries, since at any given time, the system can know about as many as 1024 virtual page mappings (512 for user, 512 for exec).

** Preceding not done in Tops-10 (uses KI-style paging)

The most direct possible solution would have been to use the virtual page number as an index into the page-table, and simply provide a status bit for each entry that indicated whether the mapping was user or exec. If that were done, however, a new problem would appear. The problem is that in any given process (whether user or exec), there are usually many references to the first few pages. If the simple scheme just described were used, it would mean that a reference to user page 000 would be written in page-table word 000. Then, if the user process issued a monitor call, it would be highly likely that a memory reference would shortly be made to exec page 000. That would cause the mapping for exec page 000 to be written over the user 000 mapping. Then when control was returned to the user, the mapping would have to be recomputed and stored back into the page-table (thus wiping out exec 000's mapping again). This wasted page-table refill activity is a component of "thrashing."

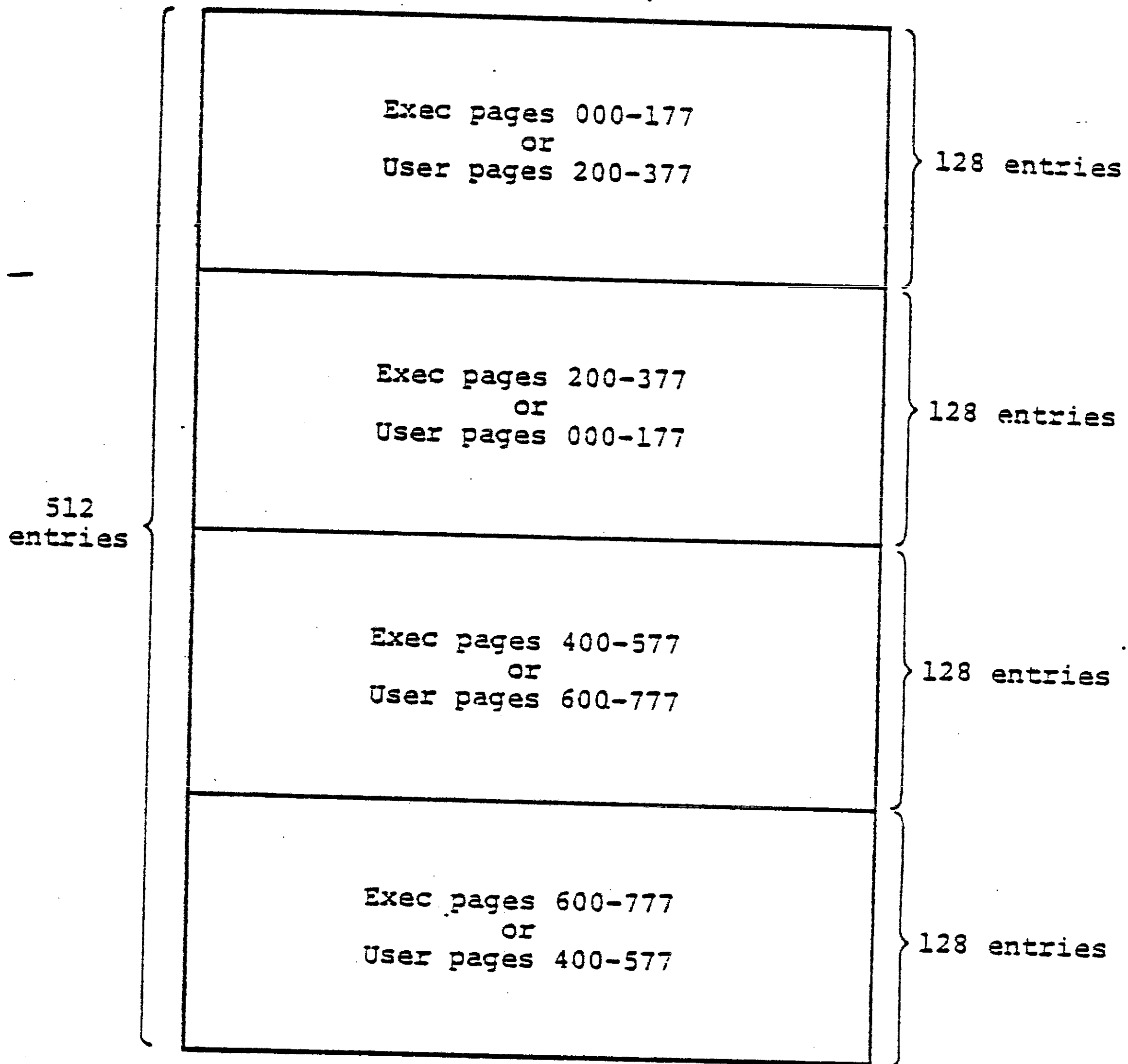
To avoid thrashing, the page-table is structured in such a way that the mapping for exec page 000 is in a different place from user page 000. The procedure used is this. When the Mbox looks for an entry in the hardware page-table, it picks up the 9-bit virtual page number from the virtual address. Next, it flips bit 19 of the page number (the second from the left as we view it) if, and only if, the virtual reference was from user space. The resulting 9-bit number is used as an index into the page-table.

For example, suppose the Mbox desires the mapping for exec virtual address 002074. The virtual page number is 002 (octal), which is 000000010 in binary. Since the address is exec, bit 19 is not changed. Therefore the map data are either in word 002 of the page-table, or the data are not in the page-table at all. In the latter case, the Mbox would have to determine the map data using the process tables and then load it into the page-table.

Alternatively, suppose the Mbox needs user-virtual address 002362. The VPN is 002 (octal), or 000000010 (binary). This time the address is user, so bit 19 gets flipped from 0 to 1, which leads to a modified index of 010000010, or 202. In this case, then, the desired data are in page-table word 202, or else not in the page-table at all.

Please note that the virtual page numbers in these two examples were the same (002). But because the addresses used were from different address spaces, the desired page-table entries were different.

Analysis of this scheme would reveal that the format of the KL's page-table is this:



KL Page Table
Figure 29

D7 0390

Cache

The purpose of cache memory is to speed up instruction execution time by substantially reducing the time needed for the average memory reference. This is accomplished by placing a high-speed semiconductor memory (the "cache") inside the Mbox. The cache can hold up to 2K words from core. Whenever the program accesses a word held in cache, the request is satisfied in 160 nanoseconds, as opposed to a microsecond or more for a core reference.

The success of this scheme depends on the quality of the algorithm used to decide which 2K from core is put into cache. Which locations are cached changes constantly with varying system demands, but the algorithm is based on the assumption that memory

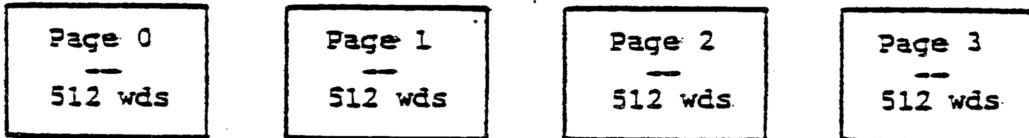
references tend to be somewhat localized. As an example, consider a typical program's structure. Usually, the flow of control is linear within a narrow scope; if an instruction has just been executed from location N, there is a good chance that there will soon be an instruction executed from location N+1.

In practice, this assumption has worked out quite well. The "hit rate" for the KL's cache memory is better than 90%. In other words, any given memory reference has nine chances in ten of being satisfied from cache, thus saving a great deal of time. The algorithm used in the KL was developed at Stanford University using extensive modelling.

Note

The cache contents are addressed by physical addresses. Thus cache comes into play only after a virtual address has been converted to physical.

As mentioned earlier, the cache can hold up to 2048 words from core memory. The cache is arranged in four pages, as follows:



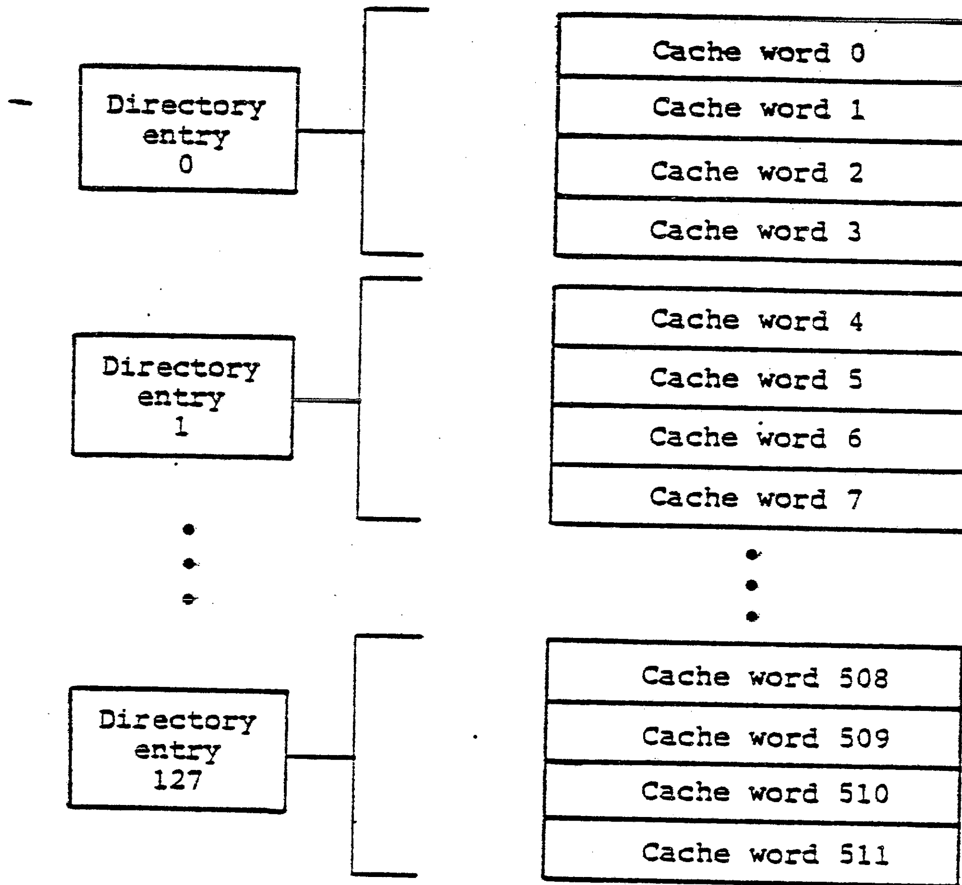
Cache Pages

Figure 30

For simplicity, let us consider one of these four pages and the format of the data stored within it. The structure to be described is identical for each of the four pages of cache.

Each cache page has a directory associated with it. A directory consists of 128 entries, each entry being 13 bits wide. A single directory entry contains information concerning four words of data within the cache page.

This gives rise to a structure that looks like this. (For convenience the diagram uses decimal arithmetic.)



Structure of the Cache Page

Figure 31

A four-word cell described by a single directory entry is called a "quadword". The 13-bit directory entry for a quadword contains the physical page number of the page in core from which the quadword came. In turn, the position of a word within a cache page is always the same as the position of the word in its original page of core.

Let's consider a specific, somewhat simplified example. Assume for the moment that we have only one page of cache and its associated directory, rather than the four that are really provided in the hardware. Suppose that the Ebox has requested the contents

of physical address 14707002, a 22-bit address. The Mbox has first to determine if it must read core location 14707002, or better, if that location is already in cache. The first step is to split the physical address into a 13-bit physical page number (in this case 14707) and a 9-bit index into the page (002). In other words, we are concerned with the 002nd word of physical page 14707. If this word is already cached, then the only place it could be in our single cache page would be in word 002 (because "the position of a word within a cache page is always the same as the position of the word in its original page"). The Mbox must therefore examine the 13-bit directory entry corresponding to the 002nd word of the page and compare it to the desired physical page number of 14707. If the directory entry holds 14707, then the 002nd cache page word is indeed the word we're looking for. If the comparison fails, then we have no choice but to read physical core.

It might be worthwhile to examine the significance of quadwords in some detail. Since there is exactly one directory entry for each quadword, it follows that all four words in the quadword must come from the same physical page. Moreover, keep in mind that a word must have the same position in the cache page that it had in the physical core page. These facts imply that the four words in a single quadword are physically contiguous in core as well as in cache.

The example just traced was simplified by the omission of three-fourths of the cache pages. In a real system with four pages (2K) of cache, a given physical word might actually reside in any one of the four pages of cache. Let us return to our example for a moment. If we need physical address 14707002, we have to keep in mind the physical page number (14707) and the index into that page (002). If the word resides in any of the four cache pages, we know it has to be in word 002 of whichever page holds it, just as we knew earlier that it had to be in word 002 of the single cache page. Therefore, the Mbox has to compare the desired physical page number of 14707 to the contents of four directory entries, one for word 002 of each of the four cache pages. If a match is found for any one, then the data is taken from the proper page. Otherwise, physical core must be read.

The system just described should serve to introduce you to the KL implementation of cache. There are several further characteristics that deserve mention.

* KL cache is not a write-through cache. If the Ebox instructs the Mbox to write a given location, the location is modified only in cache. The corresponding physical location will be updated only when the monitor instructs the Mbox to sweep cache, or when a quadword must be emptied to make room for new data. This fact has considerable importance for multi-processor KL systems.

* KL cache is organized to handle physical addresses. The cache

scheme used on some other large systems, however, is oriented to virtual addresses. Stanford's modelling demonstrated that the use of virtual addresses in the cache algorithm is less efficient than use of physical addresses.

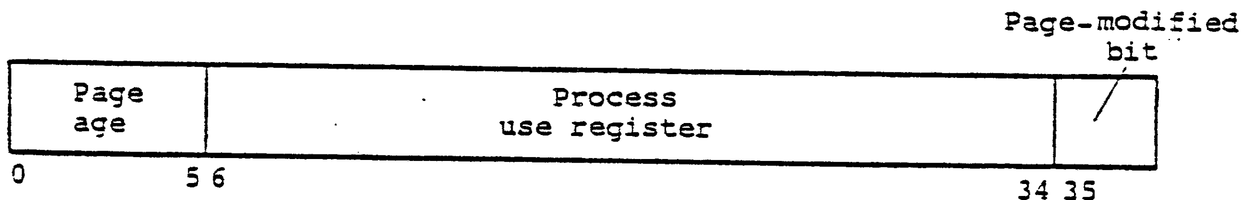
* The hardware's use of the cache is dependent upon the Mbox microcode. This microcode is normally set up to support use of all four cache pages and four-way interleaving. If desired, however, some or all of the cache can be turned off. This option is exercised when the front end is initializing the -10 at system startup.

There are three different operations to which the monitor can subject the cache: invalidation, validation, and unloading. Any of these operations can be performed on the entire cache, or on entries belonging to a single page.

To invalidate a location is simply to clear its valid and written bits, all of which has the effect of simply emptying the location. Validation of a location means that if an entry has been written since it was brought in from memory, then the modified contents must be written back into physical core. This situation arises from the fact that the cache is not a write-through cache. Finally, the unloading of a location first requires the Mbox to validate the location, then to invalidate. In other words, the location is first written into core if it has been changed since being loaded, then the location is emptied.

Core Status Table (KL-style only)

The Core Status Table (CST) is indexed by physical page number and contains one word for each physical core page. Each word has the following format:



CST Data Word

D7 0393

Figure 32

The microcode references the CST only when the pager has to get data from memory (as opposed to finding it in cache). When this happens, the CST entry for the referenced page is checked. If age stamp bits 0-5 are non-zero, the reference proceeds. However, if the age stamp is zero, a page fault occurs.

Here's why. Periodically, the monitor may decide to housekeep system storage, which results in various process pages being placed on the system free list. Theoretically, the monitor could write these pages on disk and change the pointer for that page to reflect the change. This is not good, though; there's no guarantee that all the pages just released will immediately be given away again. So if a page is not reassigned, and the last owner of the page tries to use it again, the monitor would have to read the page back from disk, even though it's still in core!

The CST gets around this problem. When a page is added to the free list, the pointer to that page is left intact. The monitor only zeroes the age stamp in that page's CST, and purges the page's data from cache. After this, two situations can arise. First, suppose the page is assigned to another process. At that time, the page's contents are written to disk (if necessary), the old pointer is changed, and the mapping proceeds. No time is lost over the scheme described earlier; things just happen later. However, suppose the page isn't reassigned, and the original owner tries to use the page again. The pager won't find the desired word in cache, because cache was flushed when the page was added to the free list. Therefore, the pager checks the CST, finds the bits zero, and generates a page fault. The monitor then takes over, determines what's happened, and gives the page back to the process by simply stamping bits 0-8 of the CST entry. Unnecessary writes and reads are avoided.

CST entries also contain a Process Use Register (PUR) and a "page modified" bit. The PUR reflects the way a page is being shared by different processes. The page modified bit is set when page data is changed. When a page must be swapped out, it needs to be written only if it's been changed; otherwise, the original copy on disk is still valid. At page-out time, the monitor decides on the need for swap-out by checking bit 35 of the CST entry for the page under consideration.

Internal Channels (Cbus)

The Cbus, and associated "internal channels", replace the older DF10/DF10C/DAS33 data-channels. Cbus features include such advantages as increased reliability and lower cost. From a system programmer's point of view, there are two principal differences.

First, the Cbus permits up to eight RH20 controllers to attach to the Mbox. Each RH20 effectively has its own data-channel in the form of a Cbus connection. And since each controller has its own channel, they can all be transferring simultaneously. In older configurations, to get the same capability would require that each

controller have its own DF10-style data-channel, which leads to considerable expense.

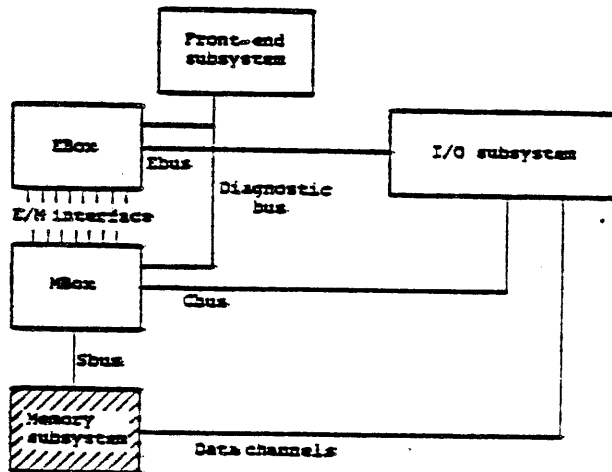
Second, the Mbox provides a sixteen-word buffer for each possible controller on the Cbus. This buffer provides protection against data overruns.

A look at page 48 reveals that the Cbus departs significantly from external channels in that the Cbus communicates solely with the Mbox, which in turn handles all transfers to or from the memory subsystem. External channels had direct connections to memory ports. Although the Mbox might seem to be a bottleneck in Cbus-equipped systems, it has been determined by testing that the Cbus runs no greater risk of overrun than external channels did.

Unfortunately, not all channel devices can be attached to the Cbus. Notable exceptions include such DECsystem-10 devices as the RH10 disk controller and the DX10 controller for TU70 tape. Systems that have these devices are equipped with internal channels where possible, and external channels when needed.

A fringe benefit of channeling data through the Cbus is that channel reads can get data from cache. This is impossible using external channels, since the data path avoids the Mbox. The advantage partially extends to output; although the Cbus cannot write cache, it does cause selective invalidation of cache words that have been changed. (Input is not directed to cache because cache would tend to be flooded in an I/O environment.) Writes using external channels required a cache sweep following the transfer.

MEMORY SUBSYSTEM



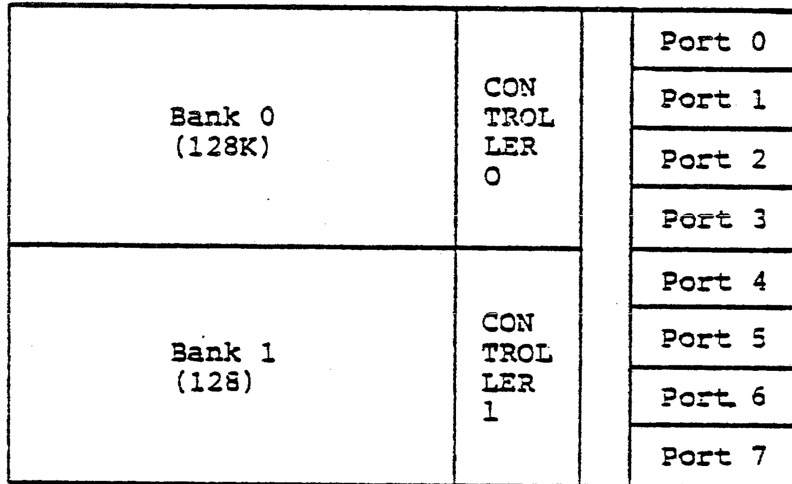
The implementation of core memory varies considerably between the DECsystem-10 computers and the DECSYSTEM-20s. The -20 line features "internal memory", while the -10 line uses memories that are external to the CPU.

Memory subsystem	1080	1090	2040	2050
Internal memory	N	N	Y	Y
DMA	Y	Y	N	N
External channels	Y	S*	N	N

* Sometimes -- see Section 3.2.4

External Memories

The external memories in use with KL systems are the MG10s and MH10s. The MG10 is normally used with 1080 systems, while 1090 systems are being shipped with MH10s. The two memories are similar internally, as follows:



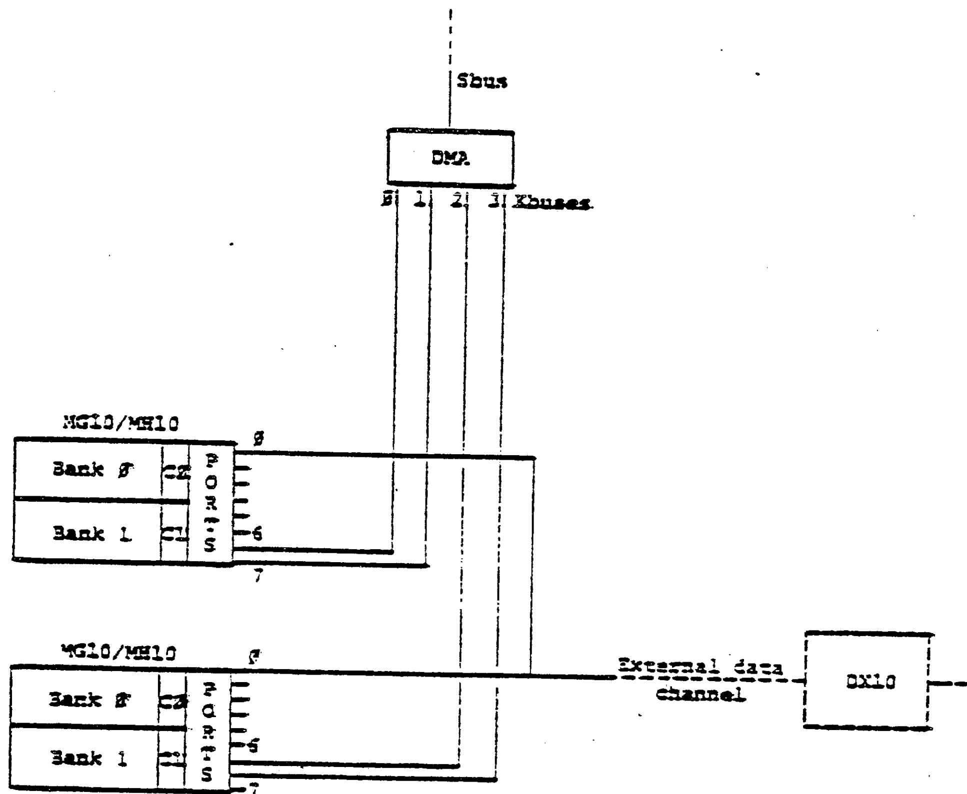
An MH10 Memory

Figure 33

A single MG10 or MH10 consists of two banks of 64/128k words, each bank having its own controller. Thus any given memory location can be serviced by exactly one controller. Since a controller can handle at most one request at a time, simultaneous requests for two locations within a single bank will result in one of the requests waiting until the other is complete. On the other hand, the two controllers operate completely independently of one another. Therefore, simultaneous requests can be made and serviced as long as the two locations needed are in different banks.

Note also that the memory has eight ports. These are priority ordered with ports 0 and 1 sharing highest priority, ports 2 and 3 sharing second priority, and ports 4 through 7 having the lowest. A request coming in on any port can be sent to whichever controller is required.

The following diagram represents a typical 1090 memory configuration.



Typical 1090 Memory Configuration

Figure 34

Note that the use of external memories dictates the presence of a DMA. This box interfaces the Sbus (one word wide) to the four Kbuses, each of which is also one word wide. In this way the system effectively has a four-word data path into the DMA and a one-word data path between the DMA and the Mbox. The four Kbuses are important to the correct operation of interleaving, which is normally four-way. This is best illustrated by means of an example. Consider the following sequence of events:

1. The Ebox asks the Mbox for the contents of a memory location. For the sake of example, suppose that the location needed is physical address 170Ø.

2. The Mbox attempts to satisfy the request by looking in cache. Frequently the desired data will already be in cache, in which case no reference to physical core need be made. Suppose, however, that location 170Ø is not in any of the four cache pages. This leads to Step 3.

3. Now the Mbox will read location 1700 from physical core into cache. In fact, not only will location 1700 be cached, but so will the other three words in the quadword. Thus the Mbox needs to read words 1700, 1701, 1702, and 1703. To do this, the Mbox requests the DMA (Direct Memory Access) to read the desired four words and pass them across the Sbus to the Mbox for caching.

4. The DMA proceeds to issue four simultaneous requests, one for each of its four Kbuses. The memories were configured for four-way interleaving when the system was first brought up, which guarantees that word 1700 will reside in memory 0 bank 0, word 1701 will be in memory 0 bank 1, 1702 will be in memory 1 bank 0, and 1703 will be in memory 1 bank 1. Since no two of these words are in the same bank, the four requests will be handled by four different controllers, in parallel. Note that the first request issued, and thus the first to be honored, is for the address originally needed. In this way, further processing can take place while the rest of the quadword is being filled.

5. As the data is sent back to the DMA from the memories, the DMA passes the information along to the Mbox. Thus the quadword is filled in cache, and ultimately the original Ebox request is satisfied.

6. This concludes our examination of the Ebox request. However, it is worth noting that in many cases the Ebox will shortly request the word adjacent to the original word, in this case 1701. If that happens, the Mbox will find that 1701 is in cache, thus obviating most of the work outlined above with considerable saving of time.

It is apparent from this example that four-way interleaving on a KL system is powerfully tied to the concept of cache quadwords. It is for this reason that system throughput suffers on caching systems whose memories are configured for either two-way or no-way interleaving.

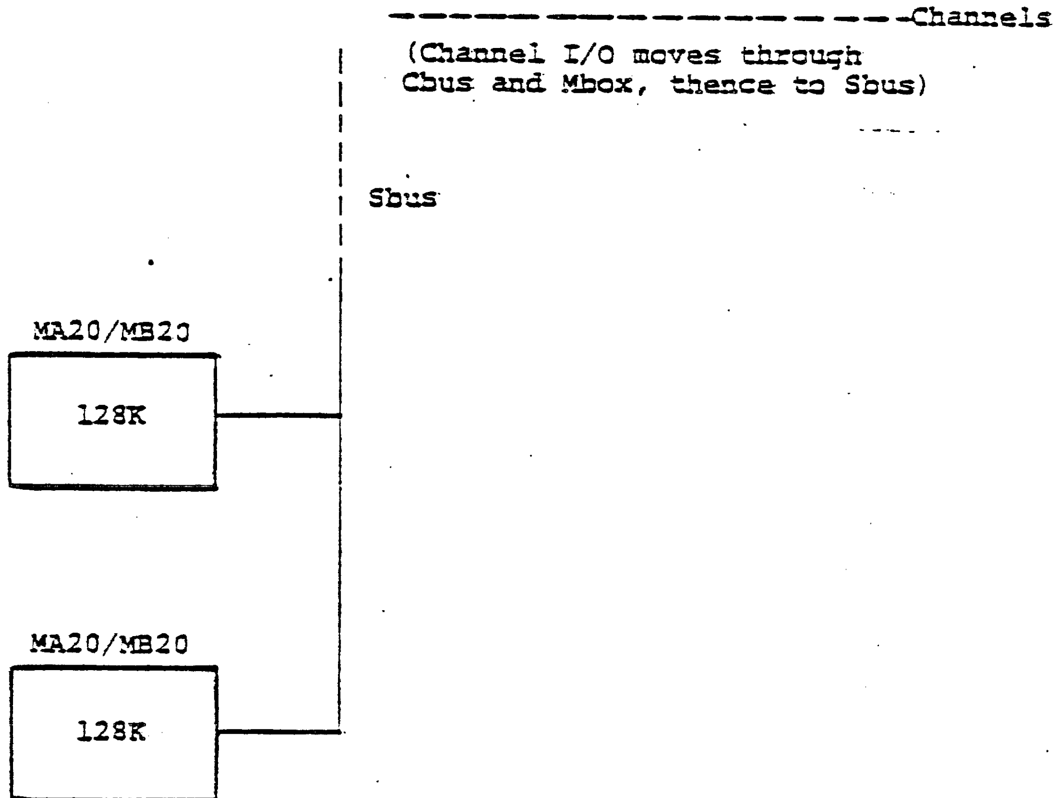
As a final note, it should be mentioned that memory configuration depends on the program in the Mbox microcode. The choice of configuration is made when the system is brought up.

Internal Memories

DECSYSTEM-2Ø machines feature internal memories. These offer improved reliability and lower cost than external memories. So far, the only internal memories offered have been the MA2Ø and the MB2Ø.

A close look at an internal memory reveals that the MA2Ø is similar to the MG1Ø. Like the MG1Ø, an MA2Ø has two memory banks, each with its own controller. The two controllers operate independently of each other, thus providing the ability to overlap within a single unit of MA2Ø memory. However, the MA2Ø contains no ports like those of the MG1Ø. There is no need for them, as -2Ø systems support no devices having external data-channels, so all memory requests are handled by the Mbox. These requests, in turn, are fed back and forth through the Sbus. By the same token, the MB2Ø is analogous to the MB1Ø.

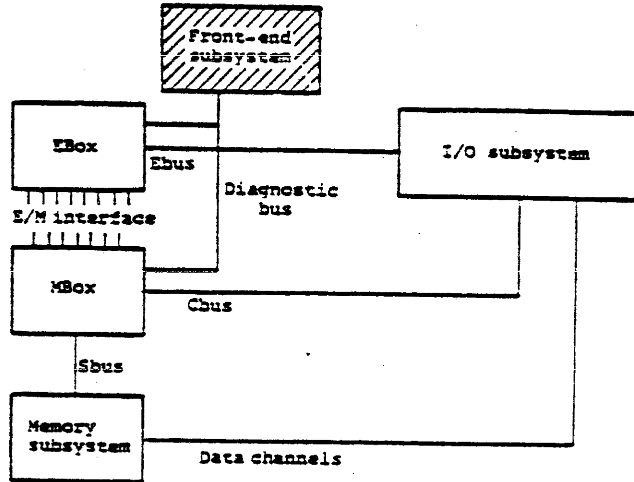
Neither is there a DMA on -2Ø system. Instead, the various memory controllers communicate directly with the Mbox.



Typical 2040 Memory Configuration D7 0396

Figure 35

FRONT-END SUBSYSTEM



Front-end	1080	1090	2040	2050
Unit-record equipment	N	N	Y	Y
Communications gear	N	S*	Y	Y

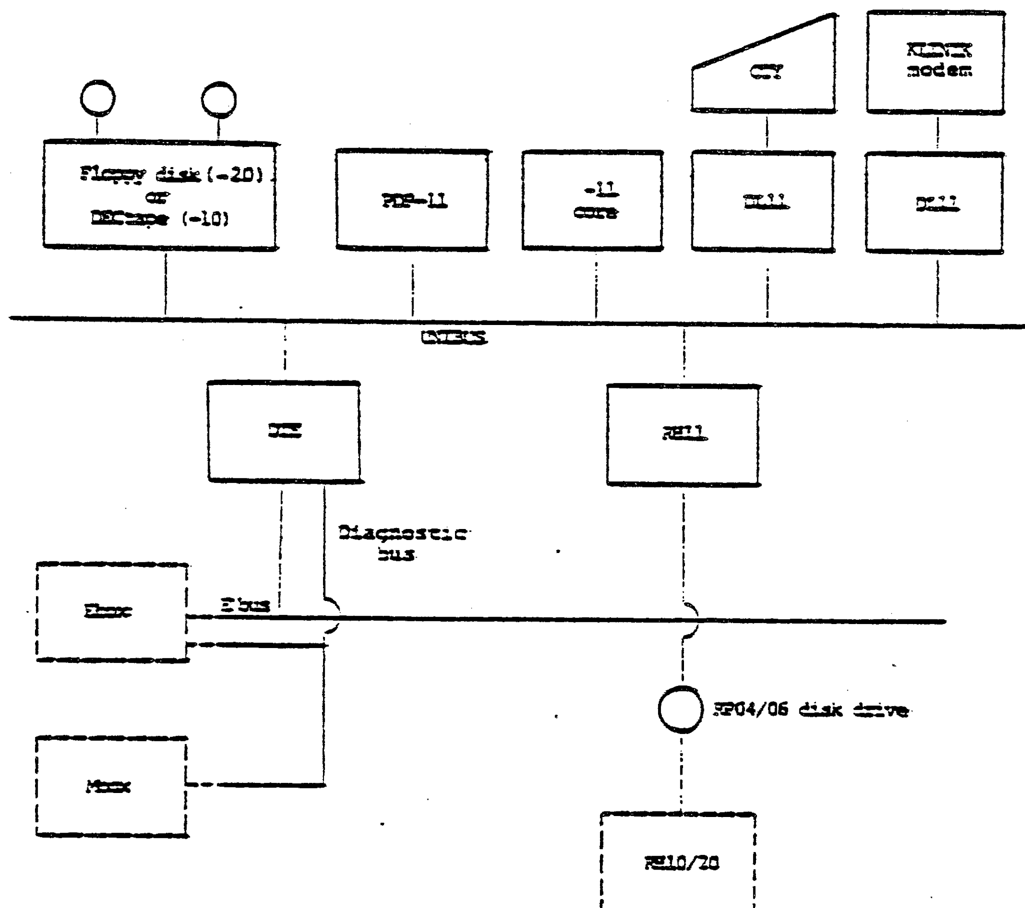
* Sometimes

The principal ingredient of the front-end subsystem is the PDP-11 computer. Like any PDP-11, it is connected to its devices by its UNIBUS as shown in Figure 36.

The DTE is the interface between the Front End and the KL CPU. The primary purpose of the DTE is to permit the Front End to control and monitor the operation of the KL CPU. The KL can support up to four DTEs.

The DTE provides the following functions:

- examine or deposit of words in specified areas of KL memory
- high speed, simultaneous, two-way data transfer (so-called "byte transfers")
- doorbell interrupts: the -11 can interrupt the KL and vice versa



Generalized Front-End Subsystem

Figure 36

DT 0399

Additionally, a specially enabled DTE can:

- examine or deposit words into any area of KL memory regardless of protection;
- control and obtain status from from the KL CPU;
- let the -11 bootstrap the KL;
- let the KL bootstrap the -11.

The DTE has two operating modes: restricted and privileged. This is determined by the setting of a manual switch on the DTE. PDP-11 attached to a restricted DTE can perform the first set of functions listed on Page 52, while a privileged DTE/-11 pair can do everything listed on Page 52. Normally only the master -11 (usually attached to DTEØ) is privileged.

There are two different ways the DTE lets an -11 communicate with KL memory. First, the -11 can use the examine/deposit feature, which permits the -11 to read or write a single KL word. The other way is with byte transfers, in which the DTE is responsible for transferring a string of data to or from KL memory without tying up either the KL or -11 CPU.

-- Examines and deposits can always be made to any address within windows defined in KL memory. The windows are specified by the KL's exec process table. There are two windows, known as the to-KL area and the to-11 area. These differ in their availability to the two processors, as follows:

	Can KL write?	Can -11 write?
to-KL area	Y	Y
to-11 area	Y	N

A restricted front end cannot examine or deposit outside of the windows. This permits the KL to protect itself from a wayward -11. However, a privileged front end can examine and deposit anywhere in KL memory, without regard for protection.

The other transfer mechanism is the byte transfer. It has the following characteristics:

- Permits transfers to or from anywhere in KL memory;
- Byte size can be eight or sixteen bits, at the programmer's option;
- Supports simultaneous to-11 and to-KL transfers.

Once the transfer has been initiated, the DTE handles it without further intervention from either CPU at the program level. In other words, the KL monitor will not be interrupted until the transfer is complete. The DTE can recognize the end of the transfer either by the transfer of a null byte or by expiration of a byte counter. Transfer completion results in an interrupt on the assigned PI level.

It is important to understand how the byte transfer is being handled internally. It was stated in the preceding paragraph that the KL monitor does not see an interrupt from the DTE until the transfer is complete. This is true, but please note that the Ebox is internally interrupted by the DTE for every byte passed across the DTE. The interrupt comes through on PI level Ø, which does not cause an interrupt that is visible to the operating system. The effect of the level Ø interrupt is to force the Ebox to move a byte between the DTE and the Mbox. Thus, every byte transferred through the DTE results in a small amount of CPU overhead, but does not require monitor action.

Byte transfers are not limited to the windows. This does not represent a security problem since in the case of a to-KL byte transfer the KL, and not the PDP-11, specifies the byte pointer and thus the destination address in KL memory.

Both forms of transfer require use of the Ebox, ~~which implies that the microcode must be running. If the microcode~~ is inoperable, a privileged -11 can use the DTE's diagnostic bus to access KL memory.

Both types of transfer (examine/deposit and byte transfer) are controlled in part by locations in the exec process table. These locations begin at octal 140:

Location	Contents
140 + 8*N	To 11 byte pointer
141 + 8*N	To 10 byte pointer
142 + 8*N	DTE-20 interrupt instruction
143 + 8*N	Reserved for DEC hardware
144 + 8*N	Examine protection word
145 + 8*N	Examine relocation word
146 + 8*N	Deposit protection word
147 + 8*N	Deposit relocation word

where N is in the range 0-3 and denotes the DTE under consideration.

Here is a more detailed description of these locations.

- To-11 byte pointer -- a byte pointer, set up in standard KL format, that tells the DTE what data to transfer to the -11. The pointer directs the DTE to exec-virtual addresses. The length of the string is determined either by a count or by the presence of a null byte at the end of the string, at the option of the programmer.
- To-10 byte pointer -- same as to-11 pointer, with the obvious exception that this pointer is used on to-KL transfers from the -11.
- DTE-20 interrupt instruction -- contains the instruction that will be performed as an interrupt instruction when the DTE interrupts the KL. The DTE is a vectored-interrupt device, so it does not interrupt through EPT locations 40+2N and 41+2N as many older devices do. Instead, the interrupt instruction is taken from this location.

  Please note that the interrupt causing this instruction to be executed will be caused by events such as transfer complete and inter-CPU doorbell. Level 0 interrupts arising from byte transfers will not go through this location; indeed, they will not produce an interrupt visible to the operating system at all.
- Examine protection word -- contains the length of the to-11 window. The length is expressed in 36-bit words.
- Examine relocation word -- contains the beginning physical address of the to-11 window.
- Deposit protection word -- contains the length of the to-KL window.

- Deposit relocation word -- the physical address of the to-KL window.

Certain front-end operations and equipment are found in all forms of KL system while others are not. Section 3.4.1 describes those features common to all KLs, while Sections 3.4.2 and 3.4.3 describe the -10 and -20 Front Ends, respectively.

Common Front-End Operations

Please study Figure 36 as you read this section.

All KL-based systems rely on the Front End for at least two basic functions. First, the -11 is responsible for initiating KL CPU operations from a dead stop. This process involves setting up KL status, loading all microstores, configuring KL memory, and starting the monitor bootstrap. These operations are conducted primarily across the diagnostic bus which is shown in Figure 36 connecting the Front End to both the Ebox and the Mbox. The second job of the -11 is to support the console terminal by which an operator can control the system. It is this terminal that governs the -11 operating system and the tasks running under it. In addition to controlling the -11 operating system, the console terminal can talk directly to TOPS-10 or TOPS-20, thus acting as a terminal as well.

The key element of these jobs is the PDP-11's operating system. The systems used vary somewhat with processor type, but all are based on the RSX operating system. The currently supported front-end monitor is RSX-20F. This system runs multiple tasks. One task is the "command parser," which is the program that recognizes commands typed in on the console terminal. Other tasks include KLINIT, which oversees initialization of the KL processor, and KLINIK, which provides a telephonic link that permits diagnosis and control of the KL from remote locations.

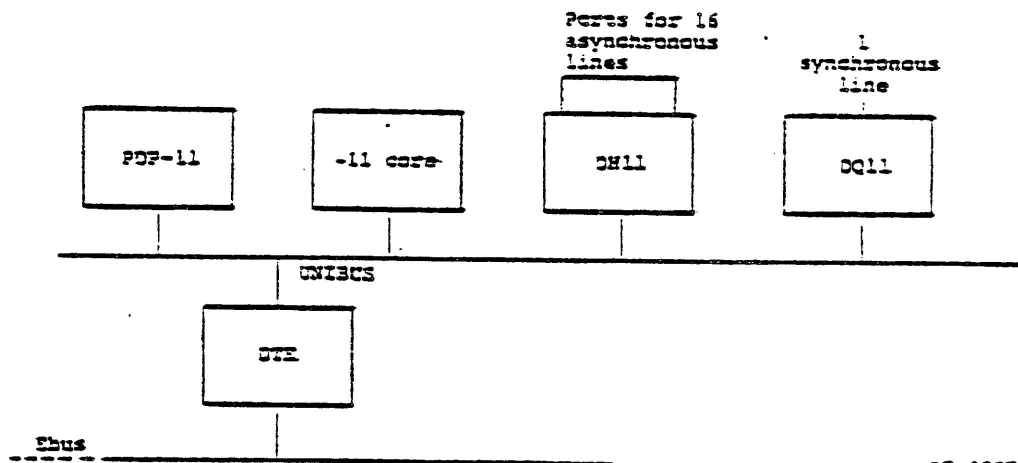
Devices associated with the Front End include the RH11 disk controller, which supports RP04/06 disk drives. Current front-end operations require the RH11 to be connected to a dual-ported disk of which the other port is connected to a KL controller (RH10 or RH20). There are both software and hardware interlocks to prevent the KL and the -11 from interfering with one another. The disk used has several tracks formatted in PDP-11 format, while the rest of the disk is KL formatted. In addition to the RH11, the Front End has either a floppy disk drive or a DECtape drive. These are used as an alternate bootstrap device if, for some reason, the disk cannot be used, or contains obsolete data.

KL systems can be attached to up to four PDP-11s. Only one of these, however, can be the controlling front end. In order to prevent conflicts between different -11s, the operations described in this section can only be done using a "privileged" DTE. A DTE is made "privileged" or "restricted" by the setting of a manual switch located on the DTE. Restricted DTEs can still move data between the Kl and the -11; such transfers require the -11 to communicate and cooperate with the KL using a software protocol, however, which presupposes that it is already running correctly. Only the privileged Front End can alter the KL state without permission from the KL itself.

DECsystem-10 Front-End

The configuration of the -10 Front End depends upon the use intended for it. The controlling Front End (i.e. that attached to a privileged DTE) will have only those devices shown in Figure 36, and does no more than what was mentioned in Section 3.4.1.

Some -10s have multiple DTEs. One of these will be the controlling Front End, and the others are part of a DN87S communications unit. Here is the structure of the DN87S;



DN87S Configuration

Figure 37

The DN87S includes such basics as -11 memory, the DTE, and the PDP-11, since the machine couldn't function otherwise. In addition it has DH11 and DQ11 line interfaces. The DH11 handles as many as 16

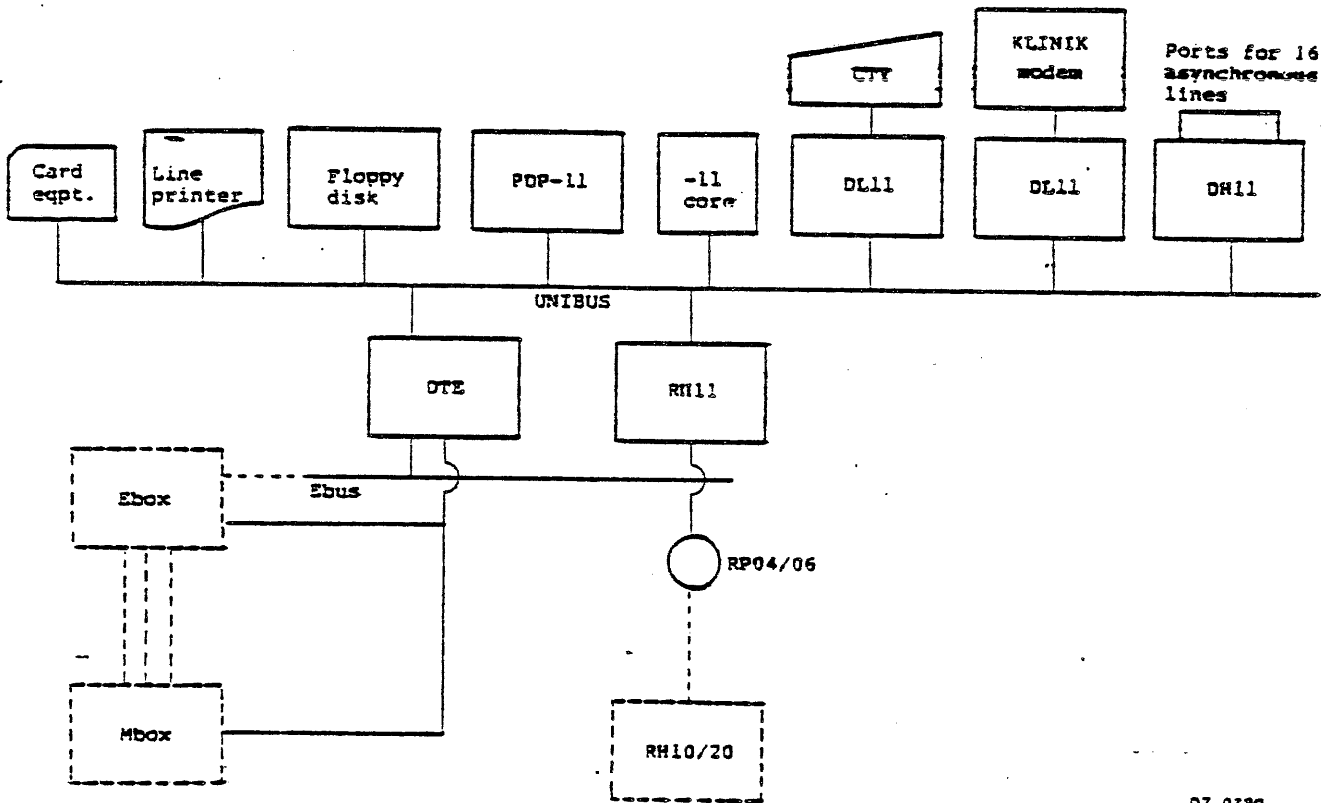
asynchronous lines. The DQ11 provides space for a single synchronous line for a remote station or link. The total capacity of the DN87S is diagramed below. This permits attachment of up to 112 asynchronous lines (using seven DH11s), or twelve synchronous lines (using 12 DQ11s), or any combination. For instance, one could run 64 asynchronous and 4 synchronous lines on a single DN87S. Combinations other than those shown are also allowed.

Note that the diagnostic bus is absent from Figure 37.

LINE MAXIMUMS PER DN87S

Max. No. of Sync. Lines	Max. No. of Async. Lines
0	112
4	64
8	32
12	0

3.4.3 DECSYSTEM-20 Front End

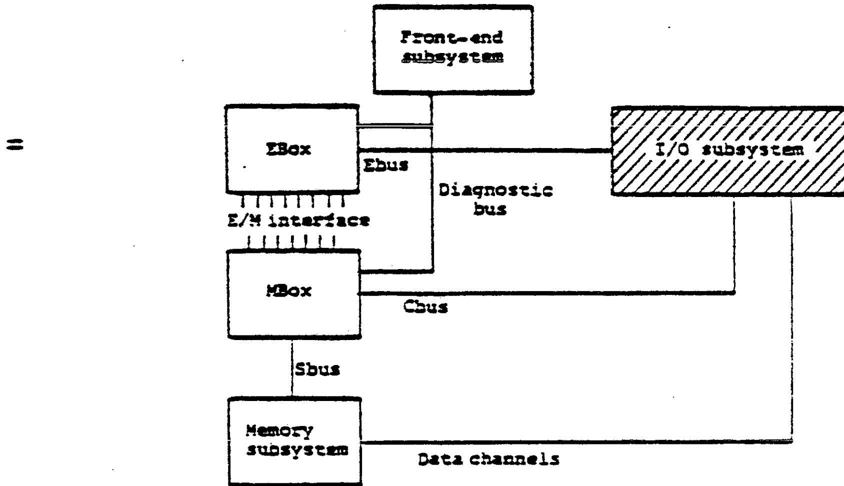


Typical DECSYSTEM-20 Front-End

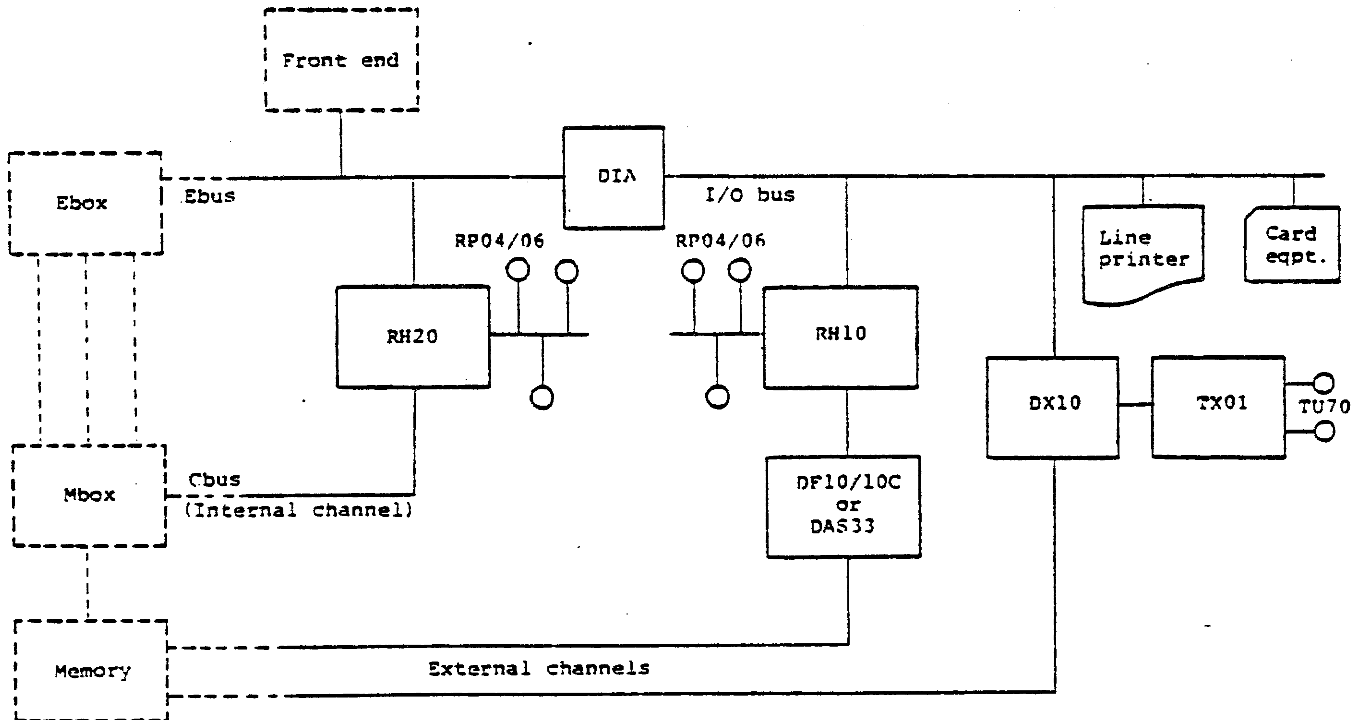
Figure 38

The -20 Front End is a much busier system under TOPS-20. In addition to handling those functions described in 3.4.1, DTE-based PDP-11s are responsible for handling all communications traffic and all unit-record equipment.

I/O SUBSYSTEM



I/O subsystem	1080	1090	2040	2050
DIA	Y	Y	N	N



I/O Subsystem

Figure 39

D7 0398

The I/O subsystem has three possible links to the rest of the system.

- Ebus -- this connects all I/O devices to the Ebox. It is through the Ebus that devices receive control signals from the CPU and return device status to the CPU.

- Cbus -- acts as a data-channel between RH2Ø controllers and the Mbox. For a complete description please read Section 3.2.4.

- External channels -- data-channels between memory and controllers for those controllers not able to use the Cbus.

The only component per se of the I/O subsystem, other than I/O devices themselves, is the DIA. The need for the DIA arises from the fact that the KL's Ebus uses a different hardware protocol than the KA/KI I/O bus, even though the basic purpose of both is the same. Only on -1Ø systems, it is necessary to connect older I/O devices to KL systems; devices that were designed to use the I/O bus protocol. To solve this problem, -1Ø systems are equipped with the DIA, which accepts a conventional I/O bus on one side and the KL Ebus on the other. The DIA is not needed on the -2Ø because the kinds of devices that need the I/O bus (e.g. unit-record equipment) is not connected directly to the KL at all on -2Øs; instead, they are attached to the Front End -11.

Previous Context Execute

Normally, an instruction's address references are handled completely within the current context. I.e., if an instruction is issued in user mode, then all its address references are handled as user-virtual addresses. There are situations, however, where it is convenient to cause an exec instruction to reference user-virtual addresses. For instance, suppose a user process issues a monitor call (UUO or JSYS) that involves an argument block at user address 770. The monitor cannot read the first argument word by saying "MOVE AC,770"; that would result in the acquisition of exec word 770, not user word 770. Theoretically, the monitor can set up a new page-map entry to point to the desired user page, but this procedure requires many instructions, and would adversely affect the operation of the pager.

The problem is solved on KL systems by use of the PXCT (Previous Context eXeCuTe) instruction. PXCT, like a conventional XCT, loads an instruction from the location specified by the PXCT's effective address. Unlike a conventional XCT, the instruction XCT'd will be treated, in whole or part, as an instruction performed in the "previous context"; that is, in the processor mode the processor was in when the most recent monitor call occurred. Normally, the previous context will be user mode (public or concealed). It could, however, be an exec mode.

Reconsider our earlier example. The monitor wishes to read user address 770. Rather than juggle page maps, the monitor would issue this instruction:

```
PXCT    14,[MOVE AC,770]
```

This instruction results in <u>user</u> address 770 being put into <u>exec</u> "AC". (Note that the number "14" in the instruction does not refer to AC 14; rather, the bit pattern 1100 in the AC field of the PXCT determines the treatment of the MOVE. This matter is discussed shortly.)

PXCT has the same opcode as XCT; an XCT becomes a PXCT when:

- the XCT is performed in exec mode, and
- the XCT's AC field (instruction bits 9-12) is non-zero.

By way of example, the instruction "XCT 0,anything" would be a conventional XCT, with the target instruction being treated as belonging to the current context. However, the instruction "XCT 5,anything" would be handled in the fashion described below.

It should be noted that the instruction name PXCT is an exact synonym for XCT; the distinction betweem the two names is purely mnemonic. Proper operation of the PXCT depends on the programmer setting up the PXCT's AC field.

Correct PXCT behavior requires that the hardware know what the previous context was. Previous context is completely defined by the following 3 items:

- previous context AC block number (0-7)
- previous context mode (user or exec)
- previous context protection (public or concealed)

All instructions, not just PXCT, require the translation of several virtual addresses. For example, suppose the CPU has just processed an instruction, and a new instruction is to be fetched and performed. The instruction will be fetched from the address given in the processor's PC word. PC addresses are virtual, so that address must be converted to physical before the new instruction can even be found. Now think ahead to the point where the instruction has been found and brought into the Ebox. The effective address must be computed , and that process involves translation, too. Consider the instruction "MOVE 5, 1043". The address 1043 must be translated to physical. In addition to all this, the system must also figure out which of the eight possible AC blocks is to be used. Otherwise the right AC5 cannot be found.

For most instructions, all such memory references will be treated as belonging to the current context. In using PXCT, however, the programmer has a choice regarding the way <u>some</u>, but not all, memory references are treated. The following types of instruction reference will always be exec mode:

- Fetch of the PXCT itself. This is only natural, since until the instruction has been fetched, the system doesn't even know it's a PXCT.

- Resolution of the effective address of the PXCT; i.e., the address of the target instruction is always an exec address. This too is a necessary function of the way the hardware operates: effective addresses are computed before the instruction opcode is looked at.

- AC field in the target instruction; This is <u>not</u> to say all AC references by the target instruction are exec. For instance, "PXCT ?,[MOVE 5,1000]" would always move 1000 of the previous context into exec AC 5, because the number 5 is in the target instruction's AC field (bits 9-12). By contrast, "PXCT ?,[MOVE 5,6]" might either move user 6 into exec 5, or exec 6 into exec 5, depending on the value of PXCT bits 9-12. The option exists in the latter instruction because the number 6 appears in the target instruction's Y field, not the AC field.

Other references may be either in user space or exec space, at the programmer's option. This choice is exercised using PXCT bits 9-12. The meaning of a "1" in any of these bits varies somewhat according to the target instruction, so we will treat three different classes: general, BLT, and EXTEND. A "1" in a position

signifies that the corresponding sort of reference is treated as a previous context address.

4.1 General Instructions

Bit position	
9	Effective address calculation for target instruction
10	Memory operands specified by E, whether fetch or store. (E.g. source address in MOVE or PUSH, destination in ADDM).
11	Not applicable - must be 0.
12	Applicable only to PUSH and POP -- address of stack as reflected by stack pointer.

4.2 BLT And XBLT

Bit position	
9	EA calculation of BLT
10	Destination address (from BLT AC right half)
11	Not applicable -- must be 0
12	Source address (from BLT AC left half)

For example, this instruction sequence will copy a 50 word block from user address 460 to exec address 702.

```
MOVE   AC,[460,,702]     ;SET UP BLT AC WITH [SOURCE,,DESTINATION]
PXCT    1,[BLT AC,751]   ;EFFECTIVE ADDRESS OF BLT DENOTES LAST
                         ;...LOCATION TO BE WRITTEN
```

Only PXCT bit 12 is set, causing only the BLT source address to be treated as a user address.

4.3 EXTEND

Bit position	
9	EA calculation of both instruction words. Also EXTEND EA calculation of source pointer if bit 11=1, and of destination pointer if bit 12=1.
10	Memory reference of second instruction word.
11	EA calculation of source, and EA calculation of source pointer if bit 9=1.
12	Destination, and EA calculation of destination pointer if bit 9=1.

APPENDIX A

KL PAGING GLOSSARY

Term	Definition
EBR	Exec Base Register.
Exec Base Register	An internal Mbox register that holds the physical page number of the Exec Process Table.
Private page	A page belonging to the address space of exactly one process. Pointed to by an immediate page map pointer; it is not pointed to by a shared pages table entry.
Section pointer	One word of data, residing in the exec process table or user process table word 440, that describes the location of a page map.
Special/Shared Pages Table	A single in-core table comprising a series of physically contiguous pages: it contains the addresses of those pages being shared between - process and a file, and addresses of special pre-process data base tables maintained by the monitor.
SPT	Shared Pages Table.
UBR	User Base Register
User Base Register	An internal Mbox register that holds the physical page number of the current User Process Table.
Section table	A one page table used in conjunction with indirect section pointers.

INDEX

-1Ø-style paging 7
-2Ø-style paging 14

Accounting meters . . . 35
Accumulator blocks
  31
  TOPS-1Ø assignments . 31
  TOPS-2Ø assignments . 31
Address
  examples of uses . . . 6
  exec virtual 4
  physical 4
  system flow 6
  types of 4
  user virtual 5
Associative memory . . . 39

Base registers 11
Byte transfers 55

Cache
  . . . . 7, 41
  quadword 43
Cbus 4, 46, 62
Channels
  external 62
Clocks
  33
  accounting meters . . 35
  interval timer . . . 33
  performance analysis counter 36
  time base 34
  TOPS-1Ø usage 36
  TOPS-2Ø usage 37
Components
  front end subsystem . 53
Control RAM 32
Core status table . . . 45
CRAM 32

Data channels
  external 4
  internal 4
DIA 62
Diagnostic bus 4
Dispatch RAM 32
DMA 5Ø

DN87S 58
DRAM 32
DTE
. . . . 2, 54
byte transfers 55
DN87S 58
EPT locations 56
examine/deposit . . . 55
functions 54
operating modes . . . 54
privileged 54
restricted 54

Ebox
accumulator blocks . . 31
clocks 33
components 31
description 1, 30
memory requests . . . 5
microcode 32
Ebus 62
Examine/deposit 55
Exec base register . . . 11
Exec process table . . . 56
Exec process table
KI-style 10
KL-style 7
Exec virtual address
description 4
External channels . . . 4, 62
External memories . . . 49

Front end
-10 operations 58
-20 operations 60
description 2
operations on -10 and -20 57
Front end subsystem
. . . . 53
DTE 54
PDP-11 54
UNIBUS 54

Hardware page table . . 39

I/O subsystem
. . . . 61
description 2
DIA 62
Interleaving 50
Internal channels
. . . . 4, 46

Internal memories . . . 52
Interval timer 33

Kbus 50
KI-style paging 7
KL clocks 33
KL configuration 1
KL page table 41
KL process tables . . . 16
KL subsystems
optional components . 2
summary 1
KL-style address translation process 18

MA20 52
Map pointers 25
MB20 52
Mbox 4
Mbox
. . . . 38
base registers 11
cache 7, 41
Cbus 46
components 38
core status table . . 45
description 2
exec base registerr . 11
internal channels . . 46
Kbus 50
memory subsystem . . . 48
Sbus 50
User base register . . 11
Memories
MG10 49
MH10 49
ports 49
Memory subsystem
. . . . 4, 48
description 2
DMA 50
external memories . . 49
interleaving 50
internal memories . . 52
Kbus 50
MA20 52
MB20 52
Sbus 50
MG10
. . . . 49
configuration using . 49
MH10 49

Microcode. 32
  routines 32

Page table 41
Paging
  -10-style 7
  -20-style 14
  associative memory . . 39
  general description . 4
  hardware page table . 39
  KI associative memory 39
  KI-style 7
  KI-style algorithm . . 11
  KL algorithm 18
  KL-style 14
  map pointer details . 25
  section pointer details 20
  VPN 8
PC usage 5
PDP-11 54
Performance analysis counter 36
Physical addresses
  description 4
  uses 4
Previous context execute 63
Privileged DTE 54
Process tables
  exec 7
  KI-style 7, 10
  KL detail 16
  KL-style 15
  user 7
PXCT 63

Quadword 43

Real-time 5
Restricted DTE 54
RH20 4

Sbus 4, 50
Section pointers 20
Subsystems
  optional components . 2
  summary 1
System address flow . . 6

Time base 34

UNIBUS 54
User base register . . . 11
User process table
  KI-style 7, 10
User virtual address
  description 5

VPN 8

PART 3

KL SYSTEM OPERATIONS

Chapter 3
KL10 System Operations

The information presented in this chapter is primarily for Digital's own system programmers, for their use in writing the Monitor and other software. However it is also needed by anyone who wishes to write his own operating system, to some extent by users who handle their own IO, and by programmers in a situation where all the facilities of a system are dedicated to a single large task.

WARNING

KL10 functions are implemented in microcode, which can be changed much more easily than hardware. Although user operations are deliberately kept as compatible as possible from one machine to the next, Digital will change the KL10 system microcode whenever such change will result in greater speed, efficiency or effectiveness. Therefore anyone writing system software should make sure to use the most recently updated version of this documentation, and before embarking on any project as enormous and critical as an operating system, to check with Large Systems Engineering for any changes not yet documented.

Programming for the system as a whole is programming in executive mode. Only the kernel program is without instruction restrictions, and only it can, if needed, access physical memory unpaged. The supervisor program labors under the same instruction restrictions as the user and has no way of bypassing them, although it can read but not alter concealed pages (the kernel program can supply data tables to the supervisor program, and the latter cannot affect them).

The amount of useful work done by the system depends upon how efficiently and effectively the executive manages the system. This means selecting which processes will run when, managing their working sets, responding to their needs, and even reacting to error situations or perhaps downright unacceptable behavior on the part of a user. The kernel program accomplishes these objectives by handling all in-out for the system, setting up page maps, trap locations, interrupt locations and the like for both itself and the users, handling user accounts, communicating with the front end, and so forth. In other words, except for handling in-out, the activities of an operating system are the topics covered in this chapter. Of course the system programmer must also be quite familiar with all of the material presented in the preceding chapters. In particular he must fully understand the architecture of the system as discussed in Chapter 1, and must be especially well versed in the use of the JRST instruction, MUUOs, and IO instructions (§§2.9, 2.16, 2.18).

System information for other processors is given in Chapters 4 and 5. The present chapter is devoted solely to the KL10, but contains two sections on paging, only one of which is applicable to a given system. §3.3 describes the paging used with the TOPS–10 Monitor; this paging is similar to that of the KI10. §3.4 treats the paging associated with the TOPS–20 Monitor. Both kinds of paging employ essentially the same hardware — the difference lies principally in the microcode.

Much of the material presented here is related to the DTE20s, the channels, and the DIA20. Although the chapter does describe all activities of the microcode undertaken for these devices (e.g. the front end functions in §3.7), the descriptions of the devices themselves are not included.

3.1 Priority Interrrupt

The DECSYSTEM–20 is essentially a system of processors clustered around the E bus. The various controllers and interfaces are subsidiary to the PDP–10, but maintain a considerable degree of independence from it. Each RH20 Massbus controller operates from its own command list in memory and handles all data transfers via the channels; but it must reach the Ten program to start a new list or if something should go wrong. Each PDP–11 is a whole computer with its own internal program; but for handling IO equipment or acting as the system console, it must communicate with Ten memory via the E bus (to which it is interfaced by a DTE20), and the peripheral computer must reach the Ten program for setting up mutual operations. Basically the priority interrupt system allows the other processors to interrupt the central processor at various levels of priority, so that all can operate simultaneously. The hardware also allows conditions internal to the PDP–10 to signal its own program by requesting an interrupt.

In a DECsystem–10, the PDP–11 is limited to use as a system console and diagnostic facility, and the unit-record peripheral equipment is organized around a KI10-type IO bus connected to the E bus via a DIA20 IO bus interface. If the system lacks internal channels, Massbus controllers must

be of the RH10 type, which the program controls via the IO bus. For data purposes an RH10 is connected to external memory by a separate memory bus. It is recommended that those who program a DECsystem–10 read both this section and the first few pages of the discussion of the KI10 interrupt[1] (§5.2).

Interrupt Requests

Interrupt requests are handled on eight levels arranged in a priority sequence. Levels are numbered 0–7, with 0 having highest priority. Level 0 is quite unlike the others, however, in that it is available only to the front end processors for simulating console functions and handling byte transfers. Moreover level 0 is always active — it cannot be turned off even by inactivating the interrupt system. The program does control the enabling of level 0 in the DTE20s, but the master front end can even override that. Assignment of devices[2] to the remaining levels is entirely at the discretion of the programmer. To assign a device to a level, the program sends the number of the level to the device control register as part of the conditions given by a CONO (usually bits 33–35); a zero assignment disconnects the device from the interrupt levels altogether. Any number of devices can be placed on the same level.

When a device requires service, it sends an interrupt request signal on its assigned level over the bus to the processor. A request is recognized by the processor if the level is active — meaning that both the interrupt system and the individual level[3] have been turned on. But the processor can accept no requests while it is processing a request or starting an interrupt at any level, or holding an interrupt on the same level or on a level with higher priority than those on which requests have been recognized (in other words, if the current program is a higher priority interrupt routine). The request signal remains on the bus however until turned off by an appropriate response from the processor: either given by the program (CONO, DATAO, or DATAI, depending on the device), or generated automatically by the hardware. Thus if a request is not recognized or accepted when made, it will be when the necessary conditions are satisfied. A single level will even shut out all others of lower priority if every time its service routine dismisses the interrupt, a device assigned to it is already waiting with another request.

---

[1] On the Ten side of the DIA20, the interrupt works as described here. But on the other side it acts more like the KI10 interrupt, with seven programmable levels, second-order priority determined by proximity to the DIA20, etc. Of course the processor activities and interrupt functions available are those of the KL10.

[2] As explained in §2.18, the program treats all E bus controllers, internal subsystems, and IO bus peripherals as IO devices. In other words, it monitors and controls them by means of IO instructions using appropriate device codes. For a PDP–11, the device is the DTE20.

[3] Remember that level 0 is always active, even when the interrupt system is off. In other respects this discussion applies to all levels.

The request signal is generally derived from a flag that is set by various conditions in the device. Often associated with these flags are enabling flags, where the setting of some device condition flag can request an interrupt on the assigned level only if the associated enabling flag is also set. The enabling flags are in turn controlled by the conditions supplied to the device by a CONO. For example, a device may have half a dozen flags to indicate various internal conditions that may require service by an interrupt; by setting up the associated enabling flags, the program can determine which conditions shall actually request interrupts in any given circumstances.

Processing a Request. The processor handles only one request at a time. When it is ready, it accepts the highest priority request currently recognized, provided that request is on a level higher than the current program (all levels are higher than a noninterrupt program). To process a request the hardware sends an interrupt service demand to the devices on the E bus to determine which ones are currently requesting an interrupt on the accepted level. Note that at this point the processor is accepting not an individual request, but rather a class of requests: namely all those being made on the same level. Should the bus be busy, the demand is sent as soon as it becomes available, taking precedence over any IO instruction that may also be waiting (note that in this situation the program actually stops). From among the devices that respond to the demand on the accepted level, the processor selects the one of highest priority[4] according to this schedule:

Devices in Order of Decreasing Priority	*Physical Device Numbers*[5]
Interval counter	
Other internal requests — processor error flags, program initiated request	
Channels 0–7	0–7
DTE20s 0–3	10–13
DIA20 — i.e. any device on the IO bus	17

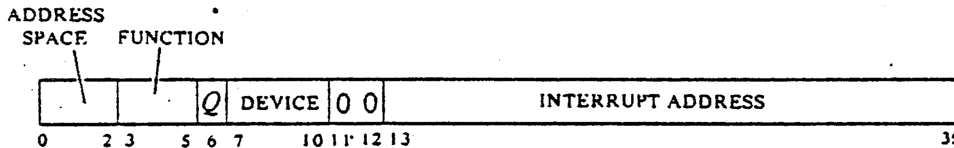
[4] There are therefore two orders of priority associated with an interrupt: first the level, and then for all devices requesting interrupts simultaneously on the same level, physical device number. These physical numbers are not the device codes used in the IO instructions; they are just for interrupt priority purposes and depend on position on the backplane (the RH20s are ordered opposite from the slot numbers).

[5] Physical numbers 14–16 are not used.

If the device selected is internal, no further processing of the request is required. Otherwise the hardware sends a function demand to the selected device (by specifying its physical number along with the interrupt level), and the device responds by returning an interrupt function word. In either case, once all necessary information about the request has been gathered, the interrupt system waits for the interrupt to start. The microcode checks frequently for a waiting request, and upon discovering one departs from its normal routine to start an interrupt. At such time PC points to the interrupted instruction, so a correct return can later be made to the interrupted program.

Interrupt Functions and Instructions

The action taken by the microcode to start an interrupt depends upon the function specified by the function word returned to the processor. Two fixed locations in the executive process table are associated with each level, locations 40 + 2*N* and 41 + 2*N*, where *N* is the level number. Level 1 uses locations 42 and 43, level 2 uses 44 and 45, and so on to level 7 which uses 56 and 57. The processor starts a "standard" interrupt for level *N* by executing the instruction in the first interrupt location for the level, i.e. location 40 + 2*N*. This type of interrupt is performed for a processor error or program-initiated request, for an external device whose function word specifies a standard interrupt, and also for an IO bus device that returns no function word. The fixed locations however need not be used. The interrupt function word sent by the device may specify an equivalent interrupt using a pair of locations selected by the function word, or some other interrupt function entirely. The function word has this format.



The microcode acts from a function word whether there is one or not; its absence is taken as a zero function. The DIA20 returns the word supplied over the IO bus or simulates a zero word. Bits 7–10 identify the device by its physical number, but this is supplied by the interrupt hardware, not the device. The meanings of the other bits in the word are as follows.

0–2 In unrestricted examine and deposit functions, codes given in these bits select the space in which the address supplied in bits 13–35 is interpreted.

0	Executive process table
1	Executive virtual address space
4	Physical address space

Remaining codes are reserved.

3–6 Interrupt function (bits 3–5), sometimes qualified by *Q* (bit 6). When unspecified, *Q* is irrelevant. The microcode handles functions 4–6 even when it is in the halt loop.

0 Internal device or zero word: for the interval counter perform a vector interrupt (see function 2); otherwise perform a standard interrupt (see function 1).

1 Standard interrupt — execute the instruction in location 40 + 2*N* of the executive process table.

2 Vector interrupt — action depends on device type as follows:

  Interval counter — execute the instruction in location 514 of the executive process table.

  DTE20 — execute the instruction in location 2 of the corresponding DTE20 control block.[6]

  Channel — execute the instruction in the executive process table location specified by bits 27–35.

  DIA20 — dispatch interrupt: execute the instruction in the executive virtual location specified by bits 13–35.

3 Increment — depending on whether *Q* is 0 or 1, add 1 to or subtract 1 from the contents of the executive virtual location specified by bits 13–35.

4 Examine — send the contents of the specified location to the selected DTE20. If *Q* is 0, select the location according to bits 0–2 and 13–35. If *Q* is 1, use bits 14–35 as a physical address and restrict the function to the communication area defined in the DTE20 control block.[6] The examine is effected by performing a DATAO to the DTE20.

5 Deposit — load the word supplied by the selected DTE20 into the specified location. If *Q* is 0, select the loaction according to bits 0–2 and 13–35. If *Q* is 1, use bits 14–35 as a physical address and restrict the function to the communication area defined in the DTE20 control block.[6] The deposit is effected by performing a DATAI to the DTE20.

6 Byte transfer — increment the byte pointer for the direction specified by *Q* (0 out, 1 in) from the control block for the selected DTE20, and then move a byte between Ten memory and the DTE20 according to the altered pointer.[6]

7 Reserved (produces a standard interrupt at present).

CAUTION

Because of the special cycle in which it is executed, an interrupt function that uses virtual addressing cannot employ indirect pointers in its paging procedure (§3.4).

---

[6] For further information on front end interrupt functions, refer to §3.7.

13–35	The bits among these that supply the address when the function requires one depend on the address space.

Executive process table	27–35
Executive extended virtual address space	13–35
Executive unextended virtual address space	18–35
Physical address space	14–35

Regardless of what mode the processor is in when an interrupt occurs, the interrupt operations are performed in kernel mode, and are therefore in executive virtual address space unless the particular function selects some other form of addressing. A page failure that occurs in an interrupt operation is never trapped; instead it sets the In-out Page Failure flag, which requests an interrupt on the level assigned to the processor (§3.8). These considerations of course do not apply to a service routine called by an interrupt instruction.

Interrupt Instructions. An instruction executed in response to an interrupt request and not under control of PC is referred to elsewhere in this manual as being "executed as an interrupt instruction." Some instructions, when so executed, have different effects than they do when performed in other circumstances. And the difference is not due merely to being performed in an interrupt location or in response (by the program) to an interrupt. To be an interrupt instruction, an instruction must be executed in the first or second interrupt location for a level, in direct response by the hardware (rather than by the program) to a request on that level. These locations may be the fixed ones for a standard interrupt or those given by the function word for a vector interrupt. §2.17 describes the two ways a BLKO is performed. If a BLKO is contained in an interrupt routine called by a JSR, it is not "executed as an interrupt instruction" even in the unlikely event the routine is stored within the interrupt locations and the BLKO is executed by an XCT. There are two types of interrupt instructions executed in a standard or dispatch interrupt; the effects of all other instructions are undefined.

BLKI, BLKO. If the pointer count is not zero, the processor dismisses the interrupt and returns immediately to the interrupted program (i.e. it returns control to the unchanged PC). If the count is zero, the processor executes the instruction contained in the second interrupt location.

XPCW, JSR. The processor holds an interrupt on the level, takes the next instruction from the location specified by the jump (as indicated by the newly changed PC), and enters either kernel mode or the mode specified by the new flag word of the XPCW. Hence the instruction is usually a jump to a service routine handled by the Monitor. XPCW is the preferred instruction on the extended KL10.

The most important point of which the programmer must be aware is that even while User is set, the interrupt instructions are not part of the user program. They are executed in kernel mode and are therefore subject only to kernel mode restrictions. Regardless of the current PC section, the address part of an interrupt instruction is interpreted as referencing sec-

tion 0, except in a dispatch interrupt, where it references the section specified by the interrupt function word. As an interrupt instruction, JSR automatically clears both User and Public to jump to a kernel mode service routine. An XPCW should be set up to produce the same result. The XPCW control block must be in section 0 unless the interrupt is a dispatch.

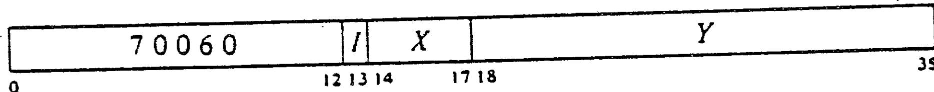
CAUTION

> Because of the special cycle in which an interrupt instruction is executed, the paging procedure for it cannot employ indirect pointers (§3.4).

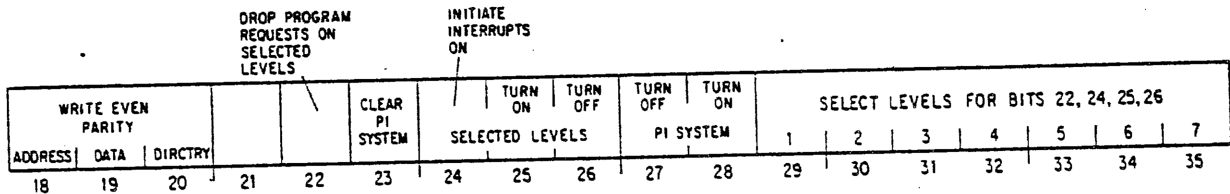
Interrupt Programming

The program can control the priority interrupt system by means of condition IO instructions. The device code is 004, mnemonic PI.[7]

CONO PI, **Conditions Out, Priority Interrupt**



Perform the functions specified by the effective conditions E as shown[8] (a 1 in a bit produces the indicated function, a 0 has no effect).



22 On levels selected by 1s in bits 29–35, turn off any interrupt requests made previously by the program (via bit 24).

23 Turn off the priority interrupt system, turn off all levels, drop all program-set requests, and dismiss all interrupts that are currently being held.

24 Request interrupts on levels selected by 1s in bits 29–35, and force the processor to recognize them even on levels that are off. The request remains indefinitely, so as soon as an interrupt is completed on a given level another is started, until the request is turned off by a CONO that selects the same channel and has a 1 in bit 22.

---

7 Data instructions with device code PI are unassigned and execute as MUUOs. The block instructions are used for error and diagnostic purposes (§3.8).

8 Bits 18–20 are for test purposes only. They are used to force errors and are discussed in §3.8.

Remember that the processor allows the program to continue while it processes a request. Thus when this bit forces recognition of a request, many additional program instructions may be performed before the interrupt, even on the highest priority level. Moreover if the request is allowed to remain, additional instructions may be performed between successive interrupts. For other than the highest priority level, the greater the number of higher levels active, the greater the amount of program time available both initially and between successive interrupts. If the program forces an interrupt on the lowest level when all are active, there can be a very long time between CONO PI, and its interrupt.

25 Turn on the levels selected by 1s in bits 29–35 so interrupt requests can be recognized on them.

26 Turn off the levels by 1s in bits 29–35, so interrupt requests cannot be recognized on them unless made by a CONO PI, with a 1 in bit 24.

27 Turn off the interrupt system so no requests can be recognized.

28 Turn on the interrupt system so the hardware can process requests.

CONI PI, **Conditions In, Priority Interrupt**

70064	*I*	*X*	*Y*
0–12	13	14–17	18–35

Read the status of the priority interrupt (and several diagnostic bits) into location *E* as shown.

											PROGRAM REQUESTS ON LEVELS						
											1	2	3	4	5	6	7
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

WRITE EVEN PARITY			INTERRUPT IN PROGRESS ON LEVELS							PI SYSTEM ON	LEVELS ON						
ADDRESS	DATA	DIRCTRY	1	2	3	4	5	6	7		1	2	3	4	5	6	7
18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35

Levels that are on are indicated by 1s in bits 29–35; 1s in bits 21–27 indicate levels on which interrupts are currently being held; and 1s in bits 11–17 indicate levels that are receiving interrupt requests generated by a CONO PI, with a 1 in bit 24. A 1 in bit 28 means the interrupt system is on, and 1s in bits 29–35 therefore indicate active levels.

The remaining conditions read by this instruction have nothing to do with the interrupt. Bits 18–20 reflect several diagnostic functions discussed in §3.8.

Dismissing an Interrupt. Unless the interrupt operation dismisses the interrupt automatically, the processor holds an interrupt until the pro-

gram dismisses it, even if the interrupt routine is itself interrupted by a higher priority level. Thus interrupts can be held on a number of levels simultaneously, but from the time an interrupt is started until it is dismissed, no interrupt request can be accepted on that level or any of lower priority.

A routine dismisses the interrupt by using an instruction that restores the level on which the interrupt is being held at the same time it returns to the interrupted program. The proper instruction is XJEN (JRST 7,) in an extended KL10, otherwise JEN (JRST 12,). Once the level is restored, the hardware can again accept requests and start interrupts on it and lower priority levels. These instructions also restore the flags: XJEN from the flag-PC doubleword if the routine was called by an XPCW; JEN from the left half of the PC word if the routine was called by a JSR in section 0. XJEN also restores the previous context section if the return is being made to an executive program.

CAUTION

> An interrupt routine must dismiss the interrupt when it returns to the interrupted program, or its level and all levels of lower priority will be disabled, and the processor will treat the new program as a continuation of the interrupt routine.

Timing. The maximum time a device may wait for an interrupt to start depends on how many active devices are of higher priority and how long their service routines are. When a given request is of highest priority, its device need never wait longer than 10 μs.

Special Considerations. When an interrupt occurs, PC points to the interrupted instruction (or to an XCT that executed it), unless the interrupt occurred in an overflow trap instruction, in which case PC points to the instruction that overflowed. After taking care of the interrupt, the processor can always return to the interrupted instruction. Either *a*) the instruction did not change anything; *b*) the interrupt was in the second part of a two-part instruction, where First Part Done being set prevents the processor from repeating any unwanted operations in the first part; or *c*) the interrupt occurred at some point in a multipart instruction where the microcode rigged the various pointers and other quantities so the processor actually restarts the instruction where it stopped, rather than from the beginning. However, in a BLT and in byte manipulation, the very mechanism that facilitates the return results in special properties of which the programmer must be aware.

An interrupt can start following any transfer in a BLT. When one does, the BLT puts the pointer (which has counted off the number of transfers already made) back in AC. Then when the instruction is restarted following the interrupt, it actually starts with the next transfer. This means that if interrupts are in use, the programmer cannot use the accumulator that holds the pointer as an index register in the same BLT, he cannot have the BLT load AC except by the final transfer, and he cannot expect AC to be the same after the instruction as it was before.

An interrupt can also start in the second effective address calculation in a two-part byte instruction. When this happens, First Part Done is set. This flag is saved as bit 4 of a flag word, and if it is restored by the interrupt routine when the interrupt is dismissed, it prevents a restarted ILDB or IDPB from incrementing the pointer a second time. This means that the interrupt routine must check the flag before using the same pointer, as it now points to the next byte. Giving an IDLB or IDPB would skip a byte. And if the routine restored the flag, the interrupted IDLB or IDPB would process the same byte the routine did.

Programming Suggestions. The Monitor handles all interrupts for user programs. Even if the User In-out flag is set, a user generally cannot reference the interrupt locations to set them up. Procedures for informing the Monitor of the interrupt requirements of a user program are discussed in the Monitor manual.

For those who do program priority interrupt routines, there are several rules to remember.

- Use interrupt instructions in a manner consistent with the special effects and conditions applicable to such instructions as described above.
- No request can be accepted, not even on higher priority levels, while a request is being processed or an interrupt is starting. Therefore do not use lengthy effective address calculations in interrupt instructions.
- To prevent a device from hanging up a level, the programmer must be aware of — and satisfy — whatever requirements the device has for dropping the request.
- The interrupt instruction that calls the routine should be an XPCW on an extended KL10, otherwise a JSR. In either case the paging for the instruction *must not* use indirect page pointers.
- The principal function of an interrupt routine is to respond to the situation that caused the interrupt. Computations and any other time-consuming activities that can possibly be performed outside the routine should not be included within it.
- Never turn off the interrupt system in a routine unless it is absolutely necessary, and then always turn it back on again as soon as possible. If one or more levels can be turned off in place of the entire system, always do that instead.
- If the routine uses a UUO it must first save the contents of the locations that will be changed by it in case the interrupted program was in the process of handling a UUO of the same type (§2.16).
- The routine must dismiss the interrupt (with an XJEN or JEN) when returning to the interrupted program. Flags and UUO locations should be restored.

3.2 Cache Management

For the user, the cache is transparent: any program simply gets information from memory and stores information in memory. But use of a cache as part of the memory subsystem reduces program time, since the cache is faster than the storage modules, and also reduces storage use by the pro-

gram, making a larger percentage of total storage cycles available to other parts of the system. As explained in §1.7, transfers between processor and memory are in four-word groups: storage references are to four locations at a time.[9] The cache contains representations of a selection of such location groups. One may view the cache as 2048 general purpose registers, organized in sets of four, which substitute temporarily for the most frequently referenced physical storage location groups. The cache serves this function not only for the program, but for all microcode references, including those for handling interrupts, traps, page refills, and other automatic operations. The way the hardware handles the cache depends upon whether the initial processor reference to a location in a particular group is read or write.

When the first processor reference to a group is to read the contents of one of its locations, memory control retrieves the entire four-word group containing the referenced location. The single word requested is supplied to the program, but all four are placed in the cache and are validated, i.e. they are tagged as words that do represent the true contents of memory. Subsequent references, read or write, to the same group are made to the cache, not to storage. If the processor modifies the contents of a location in the group, the new word supplied is substituted for the one in the cache location, which is tagged as written. Thus the cache word is different from storage but still valid — i.e. it represents what the storage location should contain.

When the first reference to a group is for writing, there is no call to storage at all. Instead the hardware sets aside a location group in the cache, with the one word in it tagged as both valid and written. Further reads or writes of the same location are handled solely with the cache, and subsequent writes to other locations in the same group are handled just like the first. But a read to a location that has not been written produces a storage reference. The requested word is given to the processor, and all words in the group that do not already have written representations in the cache are inserted into the group entry.

When storage is being updated or a group entry that is not in use is replaced by another, words just valid can be thrown away. But written words must eventually be sent to a storage module.

Cache Structure. The 2048 locations in the cache are contained in 128 lines of sixteen each. The lines are identified by the possible group numbers in a single page, 0–177. Each line contains four group entries for the given number. Each group entry in turn comprises the number of the physical page[10] containing the storage group corresponding to the entry and representations of the four locations in the group, each with valid, written and parity bits.

---

[9] Of course memory control does not blindly request four storage cycles for every group even when it is known that some are unnecessary. Fewer references are made when some locations in a group already have valid representations in the cache, or the first or last transfer in a channel block is for part of a group.

[10] The list of all page numbers makes up the cache "directory." For many hardware functions the cache is organized in four quadrants. A quadrant contains 128 group entries, one from each line.

The hardware also includes a mechanism for keeping track of the use of the various group entries. Whenever the processor references a group whose corresponding line in the cache already contains valid entries from four other pages, the hardware puts the new group representation in place of the least recently used entry in the line. But in doing so it also updates from any representations tagged as written in the displaced group entry.

Internal Channels. The channels are expected in general to deal with the storage modules, but if the cache contains any valid words for a page being handled through the channels, the hardware acts as follows:

> In an output operation, any valid representations at locations addressed by a channel are taken from the cache instead of storage.
>
> In an input operation, all data is sent to storage. However any entries that are in the cache for locations addressed by the channel are invalidated.

The reasons for this behavior are apparent. For output any valid words left in the cache might as well be taken since that is faster than going to storage. Furthermore some valid entries may have been written, and it is assumed that storage will certainly not be more up to date than the cache. Anything brought in via a channel is assumed to be the correct copy, and it should therefore go to storage as the page cannot be in use at the same time it is being loaded. Any valid entries left over in the cache must be from some previous operation, and they should therefore be invalidated, so any future references to those locations will go to storage for the correct copy. Should any of the valid leftovers be tagged as written, it is assumed the Monitor would have swapped out the modified page before bringing in the new. Of course a page used as temporary storage, or to hold counters and control words, albeit modified, can just be thrown away.

Cache Programming

The operations the program can perform on or for the cache are three: to invalidate, to validate, and to unload. Any of these operations may be carried out for all entries in the cache or for all entries of a single page. To invalidate a location is simply to clear its valid and written bits so it no longer represents anything. To validate or unload means to update storage, i.e. to write a cached word into storage if it is tagged as written, and to clear the written bit. Otherwise validating storage leaves the validity of the cache entries unchanged, whereas unloading invalidates all entries, written or not, in the groups being processed (all those in a single page or the entire cache).

Following power turnon in any system, the cache use tables must be initialized and the cache invalidated, as its initial state is indeterminate. Beyond this, a system with a single central processor and internal channels requires no cache programming, as everything is handled adequately by the hardware. However if a system contains facilities that bypass the processor to deal directly with external memory, whether such facility be an external channel or another central processor, then the Monitor must actually manage the relationship between storage modules and cache.

As an example of such management and to illustrate the difference in use between validation and unloading, consider the situation in which a program is through with the data in a particular (modified) page and it is to be swapped via an external channel with new data brought into the same physical page for later use. The page must be unloaded into storage so that subsequently the program will go there for the new data. On the other hand suppose a program has created some code in a page, and the system is both to go ahead and execute it immediately and place it in a library. Now validation is the proper procedure: while the storage copy is being filed, the program can continue execution from the cache.

For initialization and management, there is one instruction that initializes the use tables and six that sweep the cache to perform the above three operations for a single page or all pages. Note that a sweep of the entire cache is always necessary, even for handling a single page, as there is no prior way of knowing whether any given line contains a group from any given page. Sweeping for a single page does however take less time than sweeping for all pages. In the latter case the sweeper must check all 512 group entries, whereas the former requires checking only every line to see if it contains an entry for the specified page, and there can be at most one such entry. Moreover sweeping for all pages can usually be expected to require more storage references than sweeping for a single page. In this light it should be noted that the sweep instructions simply initiate operations which are then carried forward by the cache sweeper. The program can continue while the sweep is going on, but this can be expected to slow down the sweep as the cache and program would then compete for storage references. That a sweep is in progress is indicated by the Sweep Busy flag being on, and at completion the sweeper clears Busy and sets Sweep Done. The program can check both of these flags among what are otherwise the processor error conditions, and it can enable the latter to request an interrupt on the level assigned to the processor (§3.8).

These are IO instructions wherein the cache sweeper has device code 014, mnemonic CCA. But the instructions have their own mnemonics since they bear no relation to the standard IO operations. Six of the eight are used: the BLKI and CONO also sweep, doing nothing but wasting cache cycle time. The single instruction that initializes the use tables is discussed at the end of the section.

SWPIA **Sweep Cache, Invalidate All Pages** (DATAI CCA,)

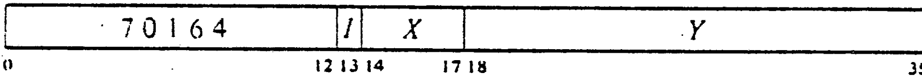
70144 (0–12)	*I* (13)	*X* (14–17)	*Y* (18–35)

E is not used.[11]

Set Sweep Busy, and clear the valid and written bits in all cache entries. At the completion of the sweep, clear Sweep Busy and set Sweep Done, requesting an interrupt on the level assigned to the processor.

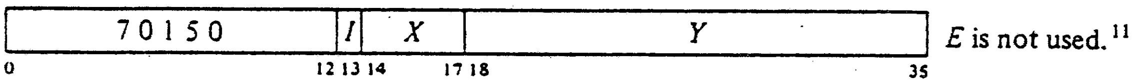
[11] *I*, *X* and *Y* are reserved and should be zero.

SWPIO **Sweep Cache, Invalidate One Page** (CONI CCA,)



Set Sweep Busy, and clear the valid and written bits in all cache entries for the physical page specified by bits 23–35 of *E*. At the completion of the sweep, clear Sweep Busy and set Sweep Done, requesting an interrupt on the level assigned to the processor.

SWPVA **Sweep Cache, Validate All Pages** (BLKO CCA,)

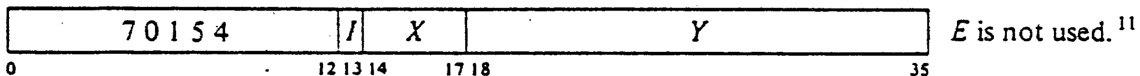


Set Sweep Busy, and write into storage all cached words whose written bits are set. Clear all written bits but do not change the validity of any entries. At the completion of the sweep, clear Sweep Busy and set Sweep Done, requesting an interrupt on the level assigned to the processor.

SWPVO **Sweep Cache, Validate One Page** (CONSZ CCA,)

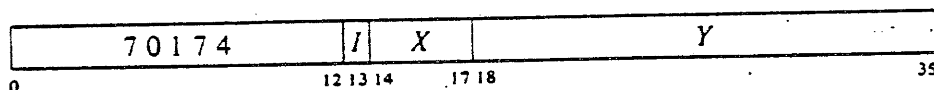
Set Sweep Busy, and write into storage all cached words whose written bits are set and which are found in entries for the physical page specified by bits 23–35 of *E*. Clear the written bits associated with those words sent to storage, but do not change the validity of any entries. At the completion of the sweep, clear Sweep Busy and set Sweep Done, requesting an interrupt on the level assigned to the processor.

SWPUA **Sweep Cache, Unload All Pages** (DATAO CCA,)



Set Sweep Busy, and write into storage all cached words whose written bits are set. Invalidate the entire cache, i.e. clear all valid and written bits. At the completion of the sweep, clear Sweep Busy and set Sweep Done, requesting an interrupt on the level assigned to the processor.

SWPUO **Sweep Cache, Unload One Page** (CONSO CCA,)



Set Sweep Busy, and write into storage all cached words whose written bits are set and which are found in entries for the physical page specified by bits 23–35 of E. Invalidate all entries for the specified page, i.e. clear both their valid and written bits. At the completion of the sweep, clear Sweep Busy and set Sweep Done, requesting an interrupt on the level assigned to the processor.

Management of the cache is relatively straightforward. With external channels the program must simply be sure always to update storage pages before having them sent out, and to invalidate the cache entries for pages being brought in so processor references will go to storage for the new data.

The same procedures are used for a multiprocessor system, but here a problem arises when different processors are allowed to reference the same page at the same time, if either is allowed also to modify the page. Without modification the cache copies in both processors will remain valid; but if a processor modifies the page, the other cannot expect to get up-to-date data from cached words. To handle this situation, the pager includes mechanisms for bypassing the cache. Each page mapping[12] contains a cache bit for determining whether cache use is allowed for the given page. This cache bit applies only to an individual page, and has no effect at all unless cache use is enabled by the cache look bit. Analogous to the mapping cache bit is a load bit that applies to all unpaged references (such as pager references to the process tables). The look and load bits are among the conditions the Monitor provides to the pager. The way these "cache strategy" conditions govern cache use is as follows.

Look

0 The cache is disabled — go to storage for all references.

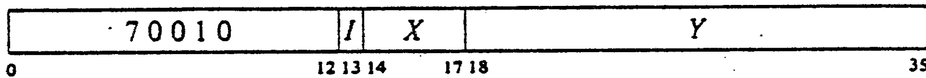
1 Look in the cache for all references. This means always use the cache (reading or writing) for any locations that already have valid representations. Furthermore when there is no valid representation for a reference, load the cache (reading or writing) if either the reference is unpaged and the load bit is 1, or the reference is paged and the cache bit in the mapping for the page is 1.

[12] For information on page mapping refer to §3.3 or §3.4 depending on whether the system uses respectively the TOPS–10 or TOPS–20 Monitor. Instructions for handling the pager are discussed in §3.5.

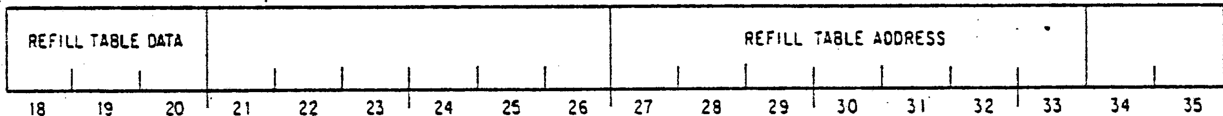
Timing. Simple invalidation takes little time, and it interferes minimally with the program since it requires no storage references. Otherwise an average sweep requires on the order of several hundred microseconds, but varies widely depending on the number of references required. Allowing the program to run simultaneously slows down the sweep because of competition for storage cycles, but program time is saved nonetheless.

Initializing the Cache. The use logic contains two tables each with 128 entries. Each entry in the use table identifies the use history — from most to least recently used — of the group entries in the corresponding cache line. With each reference, the use entry for the line must be updated. But instead of containing complex computational logic, the hardware has a refill table that supplies new use entries as a function of the previous use history of a given line and the group entry currently being accessed in the line. Following power up the program must initialize the use logic by giving this instruction 128 times to load every 3-bit location in the refill table.

WRFIL **Write Refill Table** (BLKO APR,)



Load the refill data given by bits 18–20 of *E* into the refill table location specified by bits 27–33.[13]



After filling the refill table by stepping through locations 0–177 (values of *E* that are multiples of 4 from 0 to 774), the program should give an SWPIA to invalidate the indeterminate initial contents of the cache. During the sweep the normal monitoring of cache access by the use logic initializes the use table from the refill table. The way the use table gets set up depends on the data pattern — the "refill algorithm" — loaded into the refill table, and the pattern selected depends on the use strategy desired for the cache. To limit cache use to a single quadrant, simply load the quadrant number (0–3) into the entire refill table. The usual use strategy is to allow equal use of all quadrants and to start with a presumed use history of most to least recently used corresponding to the numerical order of the quadrants. To implement this strategy,[14] load the following data pattern.

[13] The refill locations are selected by bits 27–33 to make use of the same lines that supply group numbers to address entries in the use table.

[14] For information on refill algorithms for other use strategies, refer to the writeup of MAINDEC10–DDQDA–L–D(SUBRTN).

	0	1	2	3	4	5	6	7
000	0	1	2	3	4	5	6	7
010	3	1	2	3	2	1	2	3
020	7	1	2	7	1	1	2	7
030	6	5	6	7	5	5	6	7
040	0	3	2	3	0	2	2	3
050	0	1	2	3	4	5	6	7
060	0	7	7	7	0	0	0	7
070	4	6	6	6	4	4	6	4
100	3	1	3	3	1	1	1	3
110	0	7	7	7	0	0	0	7
120	0	1	2	3	4	5	6	7
130	4	5	5	7	4	5	4	7
140	0	1	2	2	0	1	2	1
150	0	5	6	6	0	5	6	0
160	4	5	6	5	4	5	6	4
170	0	1	2	3	4	5	6	7

3.3 TOPS–10 Paging and Process Tables

General information about the machine modes and paging procedures is given in §1.3. Here we treat in detail the structure of the process tables and certain hardware procedures — paging and page failures — a knowledge of which is necessary for an understanding of executive programming. This section covers these topics relative to a machine that uses the TOPS–10 Monitor. The next section presents equivalent information for the TOPS–20 Monitor. Instructions through which the Monitor controls the pager and otherwise exercises overall management of the program environment are the same whether the system uses TOPS–10 or TOPS–20, and are described in §3.5.

With paging turned on, the program considers all of its dealings with memory to be in its virtual address space, and interrupt functions and instructions reference executive virtual address space except in special cases where a function specifically calls for physical references. A virtual address is any address given in virtual space except those for fast memory, which are treated as physical. The pager maps only virtual addresses, but it is involved in all references to the extent that it responds to error situations. Physical references include those made by the pager-microcode to carry out the mapping procedure, and also microcode references to retrieve interrupt instructions, handle traps and UUOs, and service the meters and front end.

Paging

All of memory both virtual and physical is divided into pages of 512 words each. The virtual memory space addressable by a program is 512 pages; the locations in virtual memory are specified by 18-bit addresses, where the left nine bits (18–26) specify the page number and the right nine (27–35) the location within the page. Physical memory can contain 8192 pages and requires 22-bit addresses, where the left thirteen bits (14–26) specify the page number. The hardware maps the virtual address space into a part of the physical address space by transforming the 18-bit addresses into 22-bit addresses.[15] In this mapping the right nine bits of the virtual address are not altered; in other words, a given location in a virtual page is the same location in the corresponding physical page. The transformation maps a virtual page into a physical page by substituting a 13-bit physical page number for the 9-bit virtual page number. The mapping procedure is carried out automatically by the hardware, but the page map that supplies the necessary substitutions is set up by the kernel mode program. Each word in the map provides information for mapping two consecutive pages with the substitution for the even numbered page in the left half, the odd numbered page in the right half.

The pager contains two 13-bit registers that the Monitor loads to specify the physical page numbers of the user and executive process tables. To retrieve a map word from a process table, the pager uses the appropriate base page number as the left thirteen bits of the physical address and some function of the virtual page number as the right nine bits. For example, the entire user space of 512 virtual pages at two mappings per word requires a page map of just half a page, and this is the first half page in the user process table. Thus locations 0–377 in the table hold the mappings for pages 0 and 1 to 776 and 777. To find the desired substitution from the 9-bit virtual page number, the hardware uses the left eight bits to address the location and the right bit to select the half word (0 for left, 1 for right).

The executive virtual address space is also 256K, but the page map for it is in three parts. The map for the first 112K (pages 0–337) is in executive process table locations 600–757. The map for the second half of the virtual address space uses the same locations in the executive process table as are used in the user process table for the user map (locations 200–377 for pages 400–777). The map for the remaining 16K in the first half of the executive virtual address space is in the *user* process table, the mappings for pages 340–377 being in locations 400–417. This means the Monitor can assign a different set of thirty-two physical pages (the per-process area) for its own use relative to each user. Hence when switching from one user to another, the Monitor need change only the user process table, this single substitution making whatever change is necessary in the executive address space for a particular user.

[15] For paging purposes page 0 has only 496 locations using addresses 20–777, as addresses 0–17 reference fast memory, which is unrestricted and available to all programs. (In general a user cannot reference the first sixteen storage module locations in his virtual page 0.) Throughout this discussion it is assumed that all references are to storage.

Figure 3.1: TOPS–10 Virtual Address Space and Process Table Layout

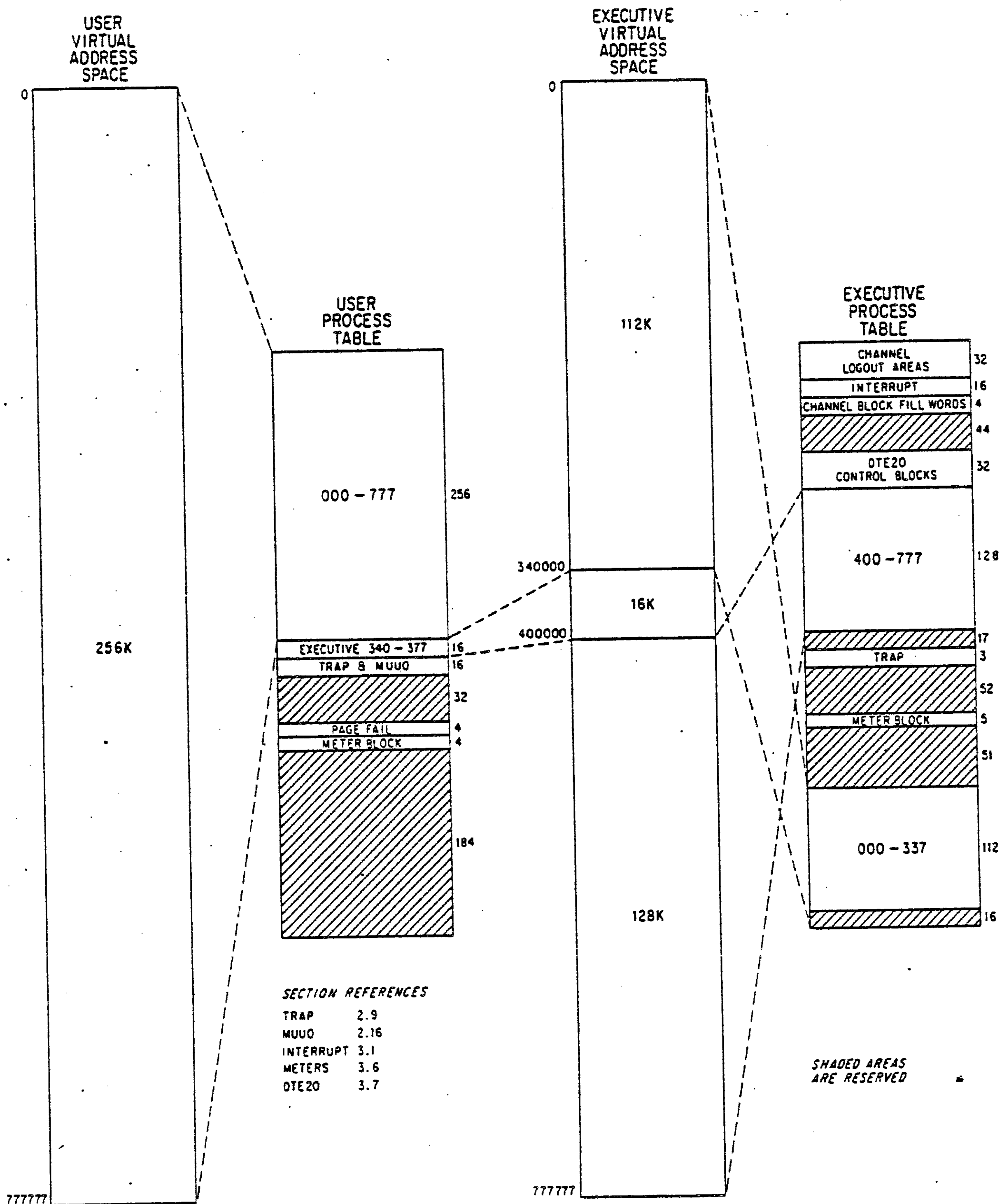


Figure 3.2: TOPS-10 Process Table Configuration

USER PROCESS TABLE

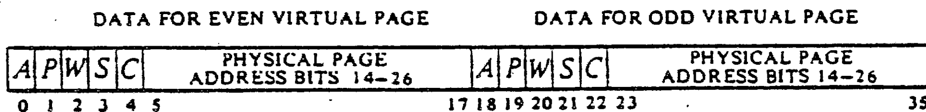
Address	Contents	
0	USER PAGE 0	USER PAGE 1
377	USER PAGE 776	USER PAGE 777
400	EXECUTIVE PAGE 340	EXECUTIVE PAGE 341
417	EXECUTIVE PAGE 376	EXECUTIVE PAGE 377
420	RESERVED	
421	USER ARITHMETIC OVERFLOW TRAP INSTRUCTION	
422	USER STACK OVERFLOW TRAP INSTRUCTION	
423	USER TRAP 3 TRAP INSTRUCTION	
424	MUUO STORED HERE	
425	MUUO OLD PC WORD	
426	MUUO PROCESS CONTEXT WORD	
427	RESERVED	
430	KERNEL NO TRAP MUUO NEW PC WORD	
431	KERNEL TRAP MUUO NEW PC WORD	
432	SUPERVISOR NO TRAP MUUO NEW PC WORD	
433	SUPERVISOR TRAP MUUO NEW PC WORD	
434	CONCEALED NO TRAP MUUO NEW PC WORD	
435	CONCEALED TRAP MUUO NEW PC WORD	
436	PUBLIC NO TRAP MUUO NEW PC WORD	
437	PUBLIC TRAP MUUO NEW PC WORD	
440–477	RESERVED	
500	PAGE FAIL WORD	
501	PAGE FAIL OLD PC WORD	
502	PAGE FAIL NEW PC WORD	
503	RESERVED	
504–505	USER PROCESS EXECUTION TIME	
506–507	USER MEMORY REFERENCE COUNT	
510–777	RESERVED	

EXECUTIVE PROCESS TABLE

Address	Contents	
0–37	EIGHT CHANNEL LOGOUT AREAS EACH: 0 INITIAL CHANNEL COMMAND 1 GETS CHANNEL STATUS WORD 2 GETS LAST UPDATED COMMAND 3 RESERVED	
40–41	RESERVED	
42–57	STANDARD PRIORITY INTERRUPT INSTRUCTIONS	
60–63	FOUR CHANNEL BLOCK FILL WORDS	
64–137	RESERVED	
140–177	FOUR DTE20 CONTROL BLOCKS	
200	EXECUTIVE PAGE 400	EXECUTIVE PAGE 401
377	EXECUTIVE PAGE 776	EXECUTIVE PAGE 777
400–420	RESERVED	
421	EXECUTIVE ARITHMETIC OVERFLOW TRAP INSTRUCTION	
422	EXECUTIVE STACK OVERFLOW TRAP INSTRUCTION	
423	EXECUTIVE TRAP 3 TRAP INSTRUCTION	
424–507	RESERVED	
510–511	TIME BASE	
512–513	PERFORMANCE ANALYSIS COUNT	
514	INTERVAL COUNTER INTERRUPT INSTRUCTION	
515–577	RESERVED	
600	EXECUTIVE PAGE 0	EXECUTIVE PAGE 1
757	EXECUTIVE PAGE 336	EXECUTIVE PAGE 337
760–777	RESERVED	

Figures 3.1 and 3.2 show the organization of the virtual address spaces, the process tables and the maps for both user and executive. The first illustration gives the correspondence between the various parts of the address spaces and the corresponding parts of the page maps. The second illustration lists the detailed configuration of the process tables as determined by the hardware. Any table locations not used are reserved for future use by the hardware or for use by the Monitor for software functions. Note that the numbers in the half locations in the page map are the virtual pages for which the half words give the physical substitutions. Hence location 217 in the user page map contains the physical page numbers for virtual pages 436 and 437.

Although the virtual space is always 256K by virtue of the addressing capability of the instruction format, the Monitor usually limits the actual address space for a given program by defining only certain pages as accessible.[16] The Monitor also specifies whether each page is public or not, writable or not, and cacheable or not. The cache bit has an effect only if cache use is enabled as the current cache strategy (§3.2); in this case a 1 in the cache bit allows loading the cache for the physical page when referenced as this particular virtual page, whereas a 0 limits cache use to look but do not load. Each word in the page map has this format to supply the necessary information for two virtual pages.



Bits 5–17 and 23–35 contain the physical page numbers for the even and odd numbered virtual pages corresponding to the map location that holds the word. The properties represented by 1s in the remaining "page use" bits are as follows.

Bit	*Meaning of a 1 in the Bit*
A	Access allowed
P	Public
W	Writable (not write-protected)
S	Software (not interpreted by the hardware)
C	Cacheable

Page Table. If the complete mapping procedure described above were actually carried out in every instance, the processor would require two memory references for every reference by the program. To avoid this, the

[16] There is no requirement that the accessible space be continuous — it can be scattered pages. The convention however is for the accessible space to be in two continuous virtual areas, low and high, beginning respectively at locations 0 and 400000. The low part is generally unique to a given user and can be used in any way he wishes. The (perhaps null) high part is a reentrant area, which is shared by several users and is therefore write-protected.

pager contains a page table, in which it keeps a large assortment of mappings for both the executive and the current user. In a manner analogous to the way the cache is organized to handle word groups of four, the pager handles mappings in sets of eight. A page set is eight consecutively numbered pages beginning with one whose number is a multiple of 10_8. Each page set consists of those pages whose mappings are contained in a single word group in the page map. The 512 locations in the page table are contained in sixty-four lines, each of eight locations holding the mappings for the eight pages of a set. The lines are identified by the possible page-set numbers in an address space, 0–77, and the individual locations are accessed by means of the virtual page numbers, 0–777. Each location has a parity bit and the complete mapping (i.e. map half word) for the virtual page that identifies it, including the physical page number and the five page use bits. Associated with each line are a bit that indicates whether the specified page set is in the user or executive address space, and a bit that indicates whether the set of mappings is valid or not (it is not suitable to clear a line as zero is a perfectly valid mapping, albeit for an inaccessible page). The user and validity bits for all lines collectively constitute the page table directory.

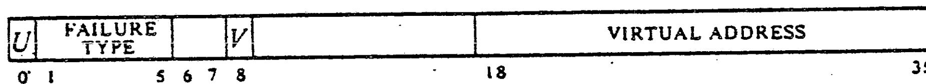
When the program references a page contained in a page set whose mapping entry is tagged as valid and in the program address space, the 13-bit physical number from the mapping location for the virtual page is used as the left thirteen bits in the physical address for the memory reference (provided of course that the reference is allowable according to the *A*, *P* and *W* bits). If however the mapping set is invalid or is not for the correct address space, the pager makes a memory reference (referred to as a "page refill cycle") to get the word group containing the mapping for the specified virtual page from the page map. Even when there is no cache, all eight mappings from the word group are entered into the page table, filling and validating the line for the page set. This means the mappings will also be in the table for subsequent references to pages in the same set, although some may require a trap to the Monitor to make them accessible.

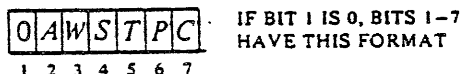
Note that all the mappings in an entire line of the page table are for a single space, user or executive. Since most programs are written beginning at page 0 (and often page 400 for a pure part), a mechanism is built into the table to avoid excessive refills due to switching between user and executive. In the numbers actually used to select lines in the table, the value of address bit 19 is inverted in user address space. For a given page number, this causes a difference of 200 in the line selection number for user space as against executive space. Suppose the executive uses pages 0–37 and 400–437, and also uses the per-process area, pages 340–377. Then if the user is limited to pages 0–137, 240–577 and 640–777, no conflict will ever occur between them in the page table.

Page Failure

When for any reason the pager is unable to make a desired memory reference, an event known as a "page failure" occurs. For this the pager terminates the instruction immediately, without disturbing PC or storing any

results in memory or the accumulators, and executes a page fail trap.[17] The trap operation makes use of three locations in the user process table: it places a page fail word in location 500, identifies the failed state of the processor by placing the current PC word in location 501, and sets up the flags and PC according to a new PC word in location 502. The processor then resumes operation in the new state at the location now addressed by PC. The page fail word supplies this information.





Whether the violation occurred in user or executive address space is indicated respectively by a 1 or 0 in bit 0; and a 1 or 0 in bit 8 indicates whether or not a virtual address was given for the reference. If bit 1 is 1, bits 6 and 7 are indeterminate, and the number in bits 1–5 ($\geqslant$ 20) indicates the type of "hard" failure as follows.

21 Proprietary violation — an instruction in a public page has attempted to reference a concealed page, or a public program has attempted to fetch an instruction from a concealed page at an illegal entry point (one not containing a PORTAL). The failure for an illegal entry (which forces bit 8 to 0) occurs at the next reference, after the instruction is decoded, so the fail address is meaningless.

22 Page refill failure — this is a hardware malfunction. The pager found no mapping for the virtual page in the page table, so it refilled the line from the page map but still could not find it.

23 Address failure — this is caused by the satisfaction of an address condition selected by the program. It is used for debugging purposes, such as to find an instruction that is maliciously wiping out a memory location, and is explained in §3.5 with the description of the DATAO APR, instruction that sets it up. Bit 8 is forced to 0 by this failure.

25 Page table parity error — the pager has encountered a page table mapping with incorrect parity.

36 AR parity error — the processor has detected incorrect parity in a word read into AR from a storage module, the cache, or the E bus, and has saved the word with correct parity in AC 0, block 7. When the source is a storage module, the MB Parity Error flag is also set (CONI APR, bit 27).

---

[17] A page failure that occurs during an interrupt instruction does not act this way. Instead it places a page fail word in AC 2, block 7, and sets the In-out Page Failure flag (CONI APR, bit 26), requesting an interrupt on the level assigned to the processor.

37 ARX parity error — the processor has detected incorrect parity in a word read into ARX from a storage module or the cache, and has saved the word with correct parity in AC 1, block 7. When the source is a storage module, the MB Parity Error flag is also set (CONI APR, bit 27).

If the failure is not one of these, then bits 1–7 have the format shown above, where *A*, *W*, *S*, *P* and *C* are simply the corresponding bits taken from the mapping for the page specified by bits 18–26, and *T* indicates the type of reference in which the failure occurred — 0 for a read-only reference, 1 for any reference involving writing. The type of reference per se implies nothing about the cause of failure — it indicates only the reason the failed reference was being made. Of course *T* being 1 in conjunction with *W* being 0 certainly implies the cause of failure.

For a page fail trap, the new PC word is set up by the Monitor to transfer control to kernel mode. After rectifying the situation, the Monitor returns to the interrupted instruction, which starts over again from the beginning or from the stopping position in a multipart instruction. Even a two-part instruction that has been stopped by a failure in the second part is redone properly, provided the Monitor restores First Part Done. The mechanism for making a correct return and the effects it produces on a BLT are the same as for an interrupt, and are described under the special considerations given at the end of §3.1.

Note that a soft failure[18] seldom implies that anything is "wrong" — unless a program has attempted to write in a truly write-protected area. Consider a typical case where the Monitor has, for example, ten or twenty pages of a user program in core; these would be the virtual pages indicated as accessible. When the user attempts to gain access to a page that is not there (a virtual page indicated in its mapping as inaccessible), the Monitor would respond to the page failure by bringing in the needed page from the disk, either adding to the user space or swapping out a page the user no longer needs.

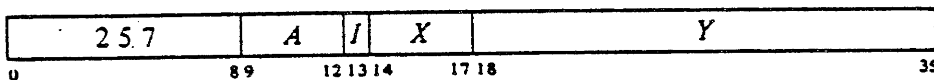
The same situation exists for writability. When bringing in a user program, the Monitor would ordinarily indicate as writable only the buffer area and other pages that will definitely be altered, distinguishing those that must be revised on the disk at the end from those that can be thrown away by setting the software bit. Then in response to a write failure, the Monitor makes the page writable and sets the software bit to indicate to itself that that page has in fact been altered and must be saved. When the user is done, the Monitor need write back onto the disk only those pages for which both *W* and *S* are set.

[18] In a soft page failure or page table parity error, the line containing the mapping for the page is invalidated on the assumption the Monitor will change it. When the instruction is restarted, the pager must go to the page map to get new information for the table.

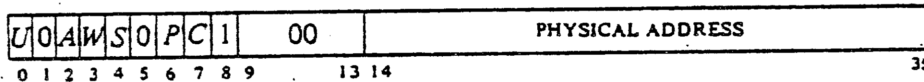
The Map Instruction

It is often helpful for the Monitor or a debugging package to be able to determine how the pager would respond to a particular reference without actually chancing a page failure. It may also be useful to determine where a particular virtual page is in physical memory, e.g. to set up a channel command list. For such purposes the processor has this instruction, which unlike all other instructions described in this chapter, is not an IO instruction even though it is subject to the same restrictions.

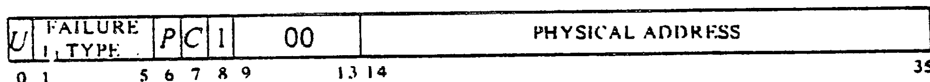
MAP **Map an Address**



If the pager is on and the processor is in kernel or user IO mode, map the page number of the virtual effective address *E* and place the resulting physical address and other map data in AC. The information loaded into AC for a true mapping is of the form



where bits 14–26 are the physical page number the pager supplies for *E*, bit 0 is 1 or 0 depending on whether the paging is done in user or executive address space, and *A*, *W*, *S*, *P* and *C* are the page use bits from the mapping as explained above. If however there is a parity error in the page table entry, or the paging is done in user mode public but the page, while accessible, is private, AC receives



The failure code can be only 21 or 25 for a proprietary or parity error, where in the latter case those bits supplied by the mapping, 6, 7 and 14–35, are meaningless.

This instruction cannot be performed in a user program unless User In-out is set, nor in a supervisor program. Instead of mapping the address, it executes as an MUUO. If the pager is off, the result is undefined.

Notes. The instruction itself cannot fail because it does not actually reference memory: it just translates the address and gets other mapping data. However the effective address calculation could fail, and getting the mapping may require a refill, in which a hard failure could occur.

3.5 Memory Management

In order properly to manage memory, the kernel program must select the kind of paging and the cache strategy, set up process tables and page maps for itself and the various users, oversee the operation of the page table, and select the fast memory block to be used by each program (usually block 0 for itself). At any given time, accumulator, index register and fast memory references are made to that AC block that is assigned as "current." Given a particular processor mode (user or executive, public or private) and an appropriate process table and page map, the Monitor effectively defines the address space for a process (which may be itself) by specifying the base address for the process table and selecting the current AC block.

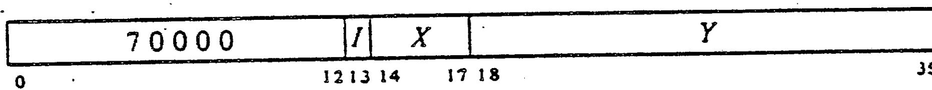
When a user program calls the Monitor it is usually to request some activity, which may often require the executive to gain access to the user address space. To facilitate the crossover from one address space to another, the same instruction through which the Monitor assigns its own current AC block also allows assignment of an AC block and section for the "previous context" — i.e. the context of the process that made the call. These quantities, together with flags that indicate the mode of the caller, allow execution of instructions in the previous context (more about this subject

later). At any point in time, the previous context is essentially the circumstances in which the previous process was running. Note that the previous context need not be the user; the same techniques can be exploited following a call from one level of the Monitor to another.

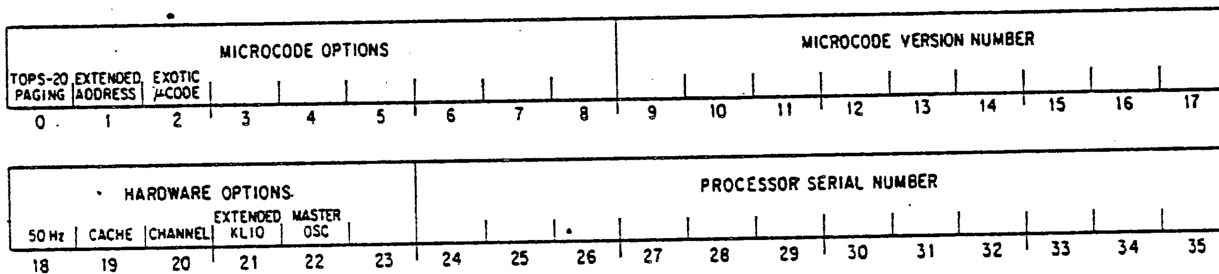
For initial setup, the kernel program must be cognizant of certain fundamental characteristics that can vary from one system to another. For this purpose the instructions for basic management include not only those that address the pager, but also one that addresses the processor to discover what those characteristics are.

The device code for the pager is 010, mnemonic PAG.[33]

APRID **Arithmetic Procesor Identification**



Read the microcode version number, the processor serial number, and a listing of the fundamental characteristics of the system into location E as shown.



0 The microcode implements paging for the TOPS–20 Monitor; 0 indicates TOPS–10 paging.

1 The microcode handles extended addresses.

2 The microcode differs in some way from the standard version.

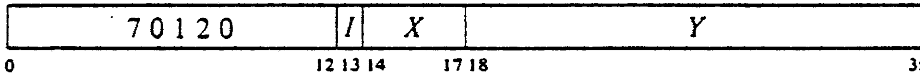
18 Line power frequency is 50 Hz rather than the standard 60 Hz.

21 The processor is an extended KL10; 0 indicates a single-section KL10. The microcode options must of course be consistent with the processor type.

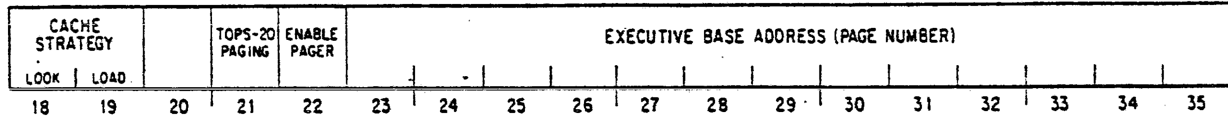
22 The system has a master oscillator, which is available as an external clock source. In a system containing MOS memory, the software must select this source (CPU clock source 2) from the PDP–11.

[33] BLKI PAG, is unassigned and executes as an MUUO.

CONO PAG, Conditions Out, Pager



Set up the system-oriented characteristics of the pager according to the effective conditions E as shown.



Load bits 23–35 into the executive base register to select the executive process table. If bit 22 is 1 enable overflow trapping and enable the pager for the type of paging selected by bit 21: 1 TOPS–20, 0 TOPS–10. The paging selected *must* be the same as that implemented by the microcode as indicated by APRID bit 0. A 0 in bit 22 prevents traps and disables paging so all memory references are to physical locations unpaged.[34]

CAUTION

> Paging can be disabled only for executive mode. A user mode program will not run correctly unless the pager is turned on.

Select the cache strategy according to bits 0 and 1 as follows:

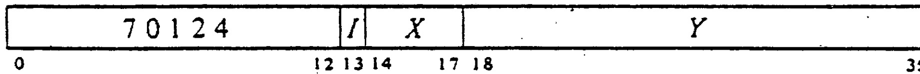
0x Disable the cache.

10 Look for all references, but do not load physical references; for virtual references act as directed by the cache bit in the mapping for the page.

11 Make complete use of the cache for physical references; for virtual references act as directed by the cache bit in the mapping for the page.

Invalidate the entire page table by setting the invalid bits in all lines.

CONI PAG, Conditions In, Pager

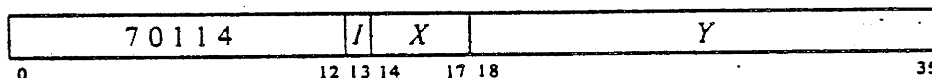


Read the system status of the pager into the right half of location E. The information read is the same as that supplied by a CONO.

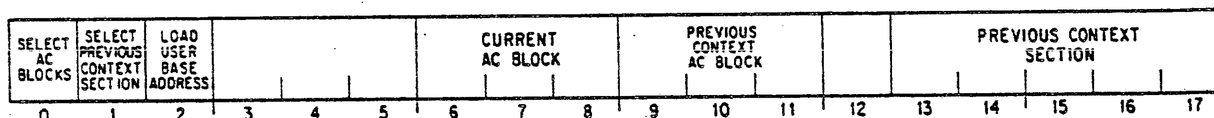
---

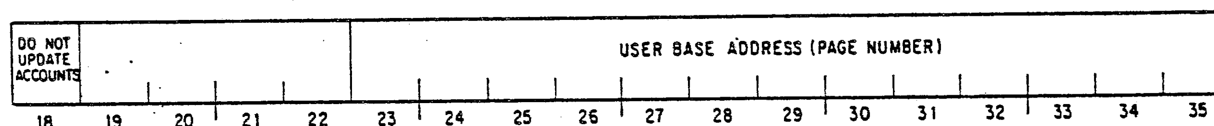
[34] Note that disabling the pager does not mean there can be no page failures, as these can be caused by conditions having nothing to do with paging, i.e. with translating virtual to physical addresses.

DATAO PAG, **Data Out, Pager**



Set up the process-oriented elements of the pager according to the contents of location E as shown.





Bits 0–2 are change indicators for parts of the data word: when a bit is 0, the corresponding part of the word is ignored, and the equivalent value supplied by a previous DATAO remains in effect.

If bit 0 is 1, select as the current and previous context AC blocks those specified by bits 6–8 and 9–11, respectively. If bit 1 is 1, select as the previous context section that specified by bits 13–17 (which must be zero in a single section processor). If bit 2 is 1, perform these functions:

If bit 18 is 0, update the user accounts as explained in §3.6.

Load bits 23–35 into the user base register to select the user process table.

Invalidate the entire page table by setting the invalid bits in all lines.

DATAI PAG, **Data In, Pager**

Read the process status of the pager into location E. The information read is in the same format as that supplied by a DATAO (bits 0–2 are 1s and bit 18 is 0). Note however that only the AC block designations and user base address are necessarily the same information supplied by a previous DATAO. When an MUUO stores its own context as given by the DATAO that set up the process containing it, it changes the designation of the previous context section to that in which the program is currently running. Hence following a call by an MUUO, a DATAI PAG, in the called program will see as the previous context section that specified by PC at the time the MUUO was performed.

CLRPT **Clear Page Table Entry**

TOPS–20	*TOPS–10*
Invalidate the page table mapping entry for the page referenced by *E*.	Invalidate the page table line (eight entries) containing the mapping for the page referenced by *E*.

At power turnon the contents of the cache and page table are indeterminate, the processor is in kernel mode, paging is disabled, the cache is off, and the current AC block is 0 by default. After the front end loads the microcode, it then loads the initializing kernel program. This program, running unpaged in physical memory, should give an APRID to determine system characteristics and an SWPIA to invalidate the cache. The unpaged program ends with a CONO PAG, that selects the cache strategy, selects and enables paging, specifies the executive base address, and invalidates the page table. From this point the kernel program runs paged and must set up the first user or users, loading the user process tables and page maps, bringing in whatever parts of user programs and data that are consistent with good working-set management, and setting up the timing and accounting meters. Finally the Monitor gives a DATAO PAG, to assign the base address and current AC block for the first user, and then transfers control to the user program via an XJRSTF or JRSTF. The initial DATAO PAG, should have a 1 in bit 18 to inhibit updating accounts before any user has run.

On a call from the user via an MUUO, give a DATAI PAG, to determine the context of the user, i.e. his AC block and section. Then give a DATAO PAG, that assigns block 0 as current for the Monitor, assigns the user AC block and section as previous context for accessing user space, but leaves the base address alone so the right paging is still available for such access. To return to the same user, reassign the AC block without changing the base address. Leaving the base address alone also avoids unnecessary updating of user accounts. Note that on the transfer to a user program no previous context values need be given as the user cannot employ PXCTs. For switching from one user to another, give a DATAO PAG, that updates the first user's accounts in his process table, as specified by the old base address, and then loads a base address for the new user. The transfer to a user is done with a JRSTF or XJRSTF; the latter also restores the previous context section when used to return from a higher to a lower level within the executive.

The usual procedure for administering AC blocks is to assign some to individual user programs on a semipermanent basis for special applications

and to assign block 1 to all other users.[35] In this way the Monitor need not store their blocks when the special users are not running, and it need not store block 1 when it takes control from an ordinary user temporarily. If the Monitor shared block 0 with any users, it would have to store the user accumulators even when taking control only temporarily. When switching from one ordinary user to another, the Monitor usually stores the first user's accumulators in his process table or shadow area — this is locations 0–17 in user virtual page 0, an area not generally accessible to the user at all — and loads the new user's accumulators from his process table or shadow area, where they were stored after the last time the new user ran.

On a change from one process to another the entire page table must be invalidated, but this is done automatically by the instruction that assigns the new user base address. If the system uses shared or indirect pointers, or several virtual page numbers point to the same physical page, then the table must be invalidated whenever a page is removed from memory or a pointer is removed from a user section table or page map. On the other hand deletion of a page with a unique mapping requires only that a CLRPT be given to invalidate the line containing it. In multiprocessor operation all page tables must be cleared whenever one is. CST entries can be used to communicate paging information from one processor to another.

Previous Context Execute

Ordinarily an instruction in a user program is performed entirely in user address space, and an instruction in the executive program is performed entirely in executive address space. But to facilitate communication between Monitor and users, the executive can execute instructions in which selected references cross over the boundary between user and executive address spaces. This feature is implemented by the previous context execute, or PXCT, instruction. The mnemonic PXCT is for convenience only and has no meaning to the assembler; it is used simply to indicate an XCT with nonzero *A* bits. A PXCT is an XCT. Although the PXCT is given by a program in the current context, some of the references made by the executed instruction can be in the previous context. A PXCT can be given only in executive mode, but the previous context may be the user, as following a call to the Monitor by the user. The previous context can however be the executive, to allow communication between one level of the executive program and another, as when the Monitor gives an MUUO to itself. (Note: it is not intended that PXCT be used by the Monitor for unsolicited references to a user program.)

It is very important to understand just which operations are affected by a PXCT and which are not. The only difference between an instruction executed by a PXCT and an instruction performed in normal circumstances is in the way certain of its memory and index register references are made. To work as a PXCT, an XCT must be given in executive mode, and the bits in its *A* field (9–12) must not all be 0 (in user mode *A* is ignored. But there is otherwise no difference in the way the XCT itself is performed: everything in the PXCT is done in the current (executive) context, and the in-

[35] It may be worthwhile to assign a separate AC block for the sole use of interrupt routines.

struction to be executed by the XCT is fetched in the current context. Moreover in the executed instruction, all accumulator references (specified by bits 9–12 of the instruction word) are in the current context. (Remember that the executive can always access a user accumulator simply by addressing it as a fast memory location.) If the instruction makes no memory operand references, as in a shift or immediate mode instruction, and it has no indexing or indirection (i.e. the instruction word gives E directly), then its execution differs in no way from the normal case. The only difference is in memory and index register references.

The previous context is specified by four quantities. Following a call by an MUUO, the section in which the calling program was running (its PC section) and the fast memory block assigned to it appear as the previous context section and current context AC block in the word read by a DATAI PAG,. For the called program, these two quantities can then be assigned as the previous context by a DATAO PAG,. The current AC block of the calling program also appears in the process context word supplied by the MUUO. Various levels of the Monitor may all use fast memory block 0; or a separate block may be assigned to that part of the Monitor that uses PXCTs in handling MUUO calls from other parts of the Monitor.

Just as the current mode is indicated by the User and Public flags, the mode in which the calling program was running is indicated by Previous Context User and Previous Context Public.[36] At a call these flags may be set up automatically or they may be set up by a flag-PC doubleword or a PC word. Note that the restrictions on references made in the previous context are those of the previous context — not those of the context in which the PXCT is given — with the single exception that if the current program is running in section 0, the previous context is also limited to section 0. Suppose the executive executes an instruction that references the concealed user area. Such a reference would fail if Previous Context Public were set.

Which references in the executed instruction are made in the previous context is determined by 1s in the A portion of the PXCT instruction word as follows.

Bit	*References Made in Previous Context if Bit is 1*
9	Effective address calculation of instruction, including both instruction words in EXTEND (index registers, address words by indirection); also EXTEND effective address calculation of source pointer if bit 11 is 1 and of destination pointer if bit 12 is 1
10	Memory operands specified by E, whether fetch or store (e.g. PUSH source, POP or BLT destination); byte pointer; second instruction word in EXTEND
11	Effective address calculation of byte pointer; source in EXTEND; effective address calculation of EXTEND source pointer if bit 9 is 1

[36] Previous Context User and Previous Context Public are in the same flag bits that are used for User In-out and Overflow in user mode. The former has no meaning in executive mode, and the latter is not really necessary as the executive program is not ordinarily interested in performing extensive mathematical procedures.

12 Byte data; stack in PUSH or POP; source in BLT; destination in EXTEND; effective address calculation of EXTEND destination pointer if bit 9 is 1

Previous context referencing is useful and reasonable in some instructions but inapplicable to others. There is no trap of any kind, and the effect of using the feature with an instruction to which it does not apply is simply undefined.

Applicable	*Inapplicable*
Move, XMOVEI	LUUO, MUUO
EXCH, BLT, XBLT	AOBJN, AOBJP
Half word, XHLLI	JUMP, AOJ, SOJ
Arithmetic	JSR, JSP, JSA, JRA, JRST
Boolean	PUSHJ, POPJ
Double move	XCT, PXCT
CAI, CAM	Shift-rotate
SKIP, AOS, SOS	String (except MOVSLJ)
Logical test	IO
PUSH, POP, ADJSP	
Byte	
MOVSLJ (extended KL10 only)	
MAP	

Note that no jumps can use previous context referencing. Even among the instructions to which such referencing is applicable, only a limited number of the sixteen possible bit combinations is useful or meaningful. Doing an effective address calculation in the previous context (selected by bit 9 or 11) makes sense only if the corresponding data access is also in the previous context (as selected by bit 10 or 12 except 11 or 12 in EXTEND). Only these combinations are permitted.

Instructions	*9*	*10*	*11*	*12*	*References in Previous Context*
General	0	1	0	0	Data
	1	1	0	0	*E*, Data
Immediate	1	0	0	0	*E*

NOTE

An *A* of 1000 is the "correct" configuration for a PXCT of an immediate mode instruction, but it inadvertently allows use of the current context section rather than the previous context as would be desired in say the PXCT of an XHLLI. To get the previous context section in the extended KL10, use 1100 instead.

	0	0	0	1	Source
BLT	0	1	0	0	Destination
	0	1	0	1	Source, destination
	1	1	0	0	*E*, destination
	1	1	0	1	*E*, source, destination

XBLT	0	0	1	0	Source
	0	0	0	1	Destination
	0	0	1	1	Source, destination
Stack	0	0	0	1	Stack
	0	1	0	0	Memory data
	0	1	0	1	Memory data, stack
	1	1	0	0	*E*, memory data
	1	1	0	1	*E*, memory data, stack
Byte	0	0	0	1	Data
	0	0	1	1	Pointer *E*, data
	0	1	1	1	Pointer, pointer *E*, data
	1	1	1	1	*E*, pointer, pointer *E*, data
MOVSLJ	0	0	0	1	Destination
(extended KL10 only)	1	0	0	1	*E* (= *Y*), destination pointer, destination
	0	0	1	0	Source
	1	0	1	0	*E* (= *Y*), source pointer, source
	0	0	1	1	Source, destination
	1	0	1	1	*E* (= *Y*), pointers, source, destination

Execution of a BLT by a PXCT is limited to these three cases:

Where all operations, regardless of context, are in section 0.

Where the previous context fast memory block is being saved in or restored from the current context, which may be any section. (But remember that regardless of context a BLT-given in-section address in the range 0–17 always refers to fast memory. Hence an AC block can never be saved in or restored from the first sixteen storage locations in any section.)

Where all operations are confined to a single section in the previous context, as would be the case when clearing a user page.

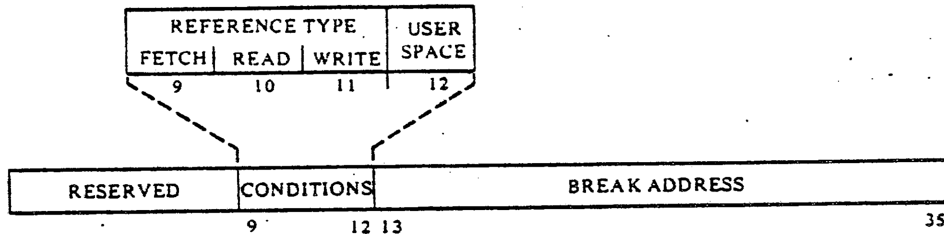
In all other circumstances XBLT must be used instead.

Address Debugging

The address failure, or address break, feature of the pager implements the traditional program debugging technique of catching a particular type of memory reference to a selected location (it does not catch fast memory references). It may be used to determine whether a given program is modifying a particular location, is executing a particular piece of code, or is simply using a particular block of data. This instruction uses the processor device code to specify the circumstances in which a break shall occur.

DATAO APR, Data Out, Arithmetic Processor

Select the break address and the break conditions according to bits 9–35 of location *E* as shown (a 1 in a condition bit selects the condition indicated, a 0 makes no reference selection or selects the opposite address space).



The break conditions selected by 1s in bits 9–12 are as follows.

9 A normal fetch of an instruction in the program under control of PC.

10 Any reference that reads except the normal fetch of an instruction. This includes retrieval of operands, address words in an effective address calculation, or an instruction to be executed by an XCT or user LUUO.

11 Any reference that writes.

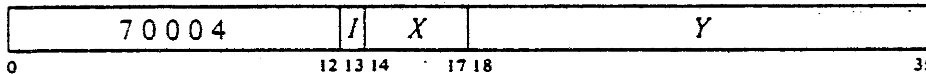
12 A reference made in user virtual address space (0 selects executive space). The break mechanism operates only for virtual address space. It does not catch microcode physical references, such as to the process tables.

Whenever the processor attempts one of the selected types of reference to the location specified by the break address in the selected virtual address space, a page failure results[37] unless the Address Failure Inhibit flag is set. This flag, which is bit 8 of the program flags and can be set only by an instruction that restores them, prevents an address failure during the next instruction — the completion of the next instruction automatically clears it. If an interrupt or trap intervenes, the flag has no effect and is saved and cleared if the flags are saved with PC. If it is not saved, it affects the instruction following the interrupt or trap. Otherwise it affects the instruction following a return in which it is restored with PC. Using the inhibit flag, the Monitor can return to a user instruction that caused an address failure and "get by it."

Since this feature is entirely under the control of the above IO instruction, it can be used quite flexibly for the executive to debug its own routines, or to debug a single user program without bothering either the executive or other users. The break conditions in effect at any time can be ascertained by giving this instruction.

37 Executive conditions also catch virtual references in interrupt functions, but the page failure sets the In-out Page Failure flag instead of resulting in a trap for an address failure.

DATAI APR, Data In, Arithmetic Processor

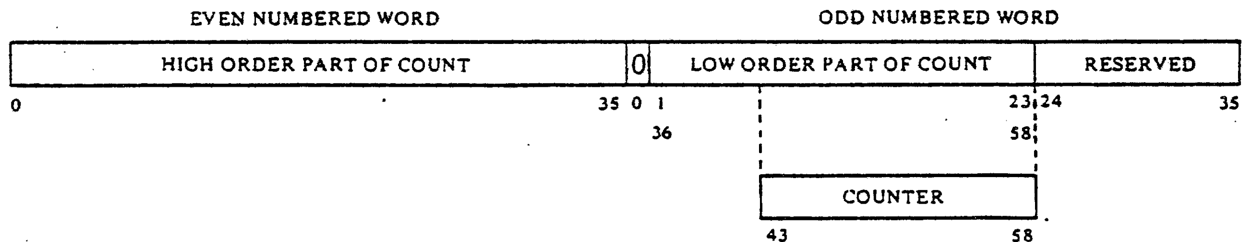


Read the current break conditions into bits 9–12 of location E. The information read is the same as that supplied by the last DATAO. (Note that the break address cannot be read.)

3.6 Timing and Accounting

The processor includes a subsystem with elements for keeping track of time, use of system facilities, and use of individual system features. One element is a standard 12-bit interval counter that is set up by the program to interrupt when the count reaches a preset value. The others are meters for keeping a 59-bit count, wherein only the low order sixteen bits are implemented in hardware. In each case the actual counting is done in a 16-bit hardware counter, while the overall count is kept in a doubleword in a process table. A count is updated from its counter by a procedure that is performed periodically by the microcode and whenever appropriate to an operation requested by the software. In the update procedure the contents of a counter are added into the corresponding count and the counter is cleared. Whenever the microcode checks for interrupt requests it updates any count whose counter is more than half full, i.e. whose MSB is 1. The current user accounts are generally updated when the Monitor switches to a new user.

A doubleword count is a 59-bit unsigned quantity whose format and relationship to the hardware counter are as shown here. The entire first word comprises the high order thirty-six bits, and the low order twenty-



three are in bits 1–23 of the second word.[38] Reserving bits for expansion at the low order end guarantees format compatibility with future machines that may be much faster (and therefore require bits for counting smaller time units). Altogether there are four meters that use this counter-doubleword format. One is a straightforward time base that counts at 1 MHz. Two keep track of process execution time and number of memory references for purposes for user accounting. Last is a mechanism for analyzing system performance by investigating the use of individual system fea-

[38] Remember, it is a property of twos complement arithmetic that the sign can be used as an extra magnitude bit in an unsigned number. But since the hardware is set up for signed arithmetic, bit 0 of any lower order word must be skipped.

tures, either by counting the number of times particular events occur or measuring the duration of time particular procedures are in progress.

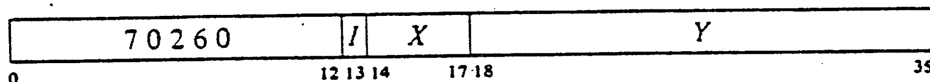
The program controls the various subsystem elements through two sets of IO instructions using device codes 20 and 24, mnemonics TIM and MTR.[39] In general the meter code is for handling the accounting meters and the timer code is for the other elements, but the MTR conditions are for both. Data instructions read updated doubleword counts, but affect neither the counts nor the counters. Condition bits (in a CONO) directly affect only the 16-bit hardware counters. Of course a counter being enabled does mean updating of the doubleword count will probably occur. But to reset a count, the program must not only clear the hardware counter but separately clear the corresponding pair of locations in the process table.

System Timing

For regular system use, the processor provides a time base and an interval counter. The time base is a doubleword count (of the type described above) kept in locations 510 and 511 of the executive process table. It counts elapsed time in microseconds (a rate of 1 MHz). Drift is guaranteed to be less than 5 seconds per day for at least the first six years of use. To maintain day-to-day accuracy, the Monitor can reset the time base once each day from the line frequency clock in the front end processor (although a line frequency clock has quite low resolution, it has very high long-term accuracy.)

The interval counter is a 12-bit hardware counter that counts in 10 μs increments (100 kHz). It can therefore count, and signal completion of, any interval from 10 μs to 40.95 ms; and it can also be read at any time to determine how long some particular operation or procedure has taken. The counter can be used for any purpose by the software, but it is employed principally to signal the Monitor should a user tie up the system too long. Associated with the counter are two flags, Interval Done and Interval Overflow. Done sets when the counter reaches the value the program specifies as its period or reaches its maximum (all 1s); Overflow sets only if the counter reaches its maximum without ever matching its period.[40] Setting Done requests an interrupt on the level assigned to the counter, and the processor responds by executing the instruction in location 514 of the executive process table.

CONO MTR, Conditions Out, Meters



Assign the interrupt level specified by bits 33–35 of the effective conditions *E* and perform the functions specified by bits 18–26 as shown.

[39] Unassigned instructions using these codes are DATAO TIM,, BLKO MTR, and DATAI MTR,. They execute as MUUOs.

[40] Overflow can occur only if at some time during the count, the program changes the period to a value less than the current counter value.

3.8 Error and Diagnostic Instructions

The first part of this section explains the instructions through which the software handles the error flags and identifies the source of a hardware error. The second part discusses a special instruction the Monitor uses to set up the memory system and to get diagnostic and configuration information directly from individual memory controllers. The objective of this treatment is to complete the definition of all KL10 instructions and to give the programmer what he needs to identify sources of hardware error for purposes of software recovery. For information on diagnosing equipment ills, the reader must turn to maintenance documents. Note that this section does not touch on diagnostic functions the front end can execute in the KL10 without the KL10 microcode running; that subject is treated in the maintenance documentation.

Error Monitoring and Investigation

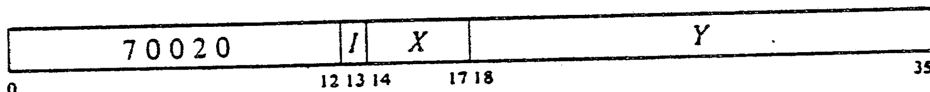
A few hardware errors — specifically a parity error in the page table or in a word brought into AR or ARX from memory — are detected by the pager and produce a page failure. Other hardware errors detected in the processor or on the S bus are indicated by flags that can request an interrupt on a level assigned to the processor. Several of these flags also lock information about the bad reference into the error address register ERA. The program can read this register, and it continues to hold the same information, even should subsequent errors occur, until the flag that locked it is cleared.

The error conditions are generally regarded as important enough to be assigned to the highest priority level. However for conditions that may be associated with user instructions (a parity error or unanswered memory reference), the common practice is for the error interrupt to switch over to the lowest priority level by means of a program-set request. Then the time

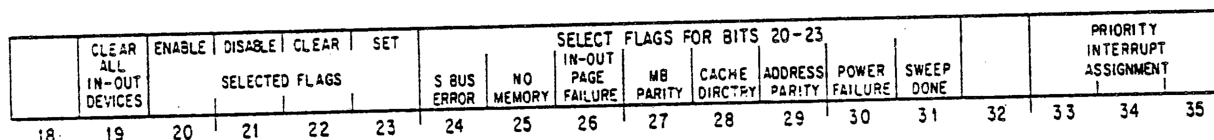
taken to handle the situation, which may well be considerable, cannot interfere with high priority events.

Error flags are handled by two condition IO instructions that address the processor, which has device code 000, mnemonic APR.[44] These instructions also handle the sweep flags for the cache (§3.2). The instruction that reads ERA uses the interrupt device code.

CONO APR, Conditions Out, Processor Flags



Assign the interrupt level specified by bits 33–35 of the effective conditions E and perform the functions specified by bits 19–31 as shown (a 1 in a bit produces the indicated function, a 0 has no effect).

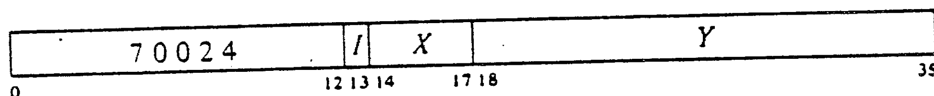


A 1 in bit 19 generates the IO reset signal, which clears the control logic in all of the peripheral equipment (but affects none of the internal devices, such as the pager or the processor flags).

Bits 20–23 select flag functions: 1s in these bits produce the indicated effects on the processor flags selected by 1s in bits 24–31. A 1 in bit 20 enables the setting of any selected flag to request an interrupt on the level assigned to the processor; a 1 in bit 21 disables the selected flags from requesting interrupts. Similarly a 1 in bit 22 or 23 clears or sets the selected flags. The result of putting 1s in both bits 20 and 21 or 22 and 23 is indeterminate.

Notes. Setting flags has of course no relation to what the flags represent; the function is used only to check out the flag logic.

CONI APR, Conditions In, Processor Flags



Read the status of the processor error and sweep flags into location E as shown (asterisks indicate bits that can cause interrupts).

[44] The processor device code is also used in several instructions for the pager and the cache.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
						FLAGS ENABLED TO INTERRUPT											
						S BUS ERROR	NO MEMORY	IN-OUT PAGE FAILURE	MB PARITY	CACHE DIRCTRY	ADDRESS PARITY	POWER FAILURE	SWEEP DONE				
						*	*	*	*	*	*	*	*				

18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35
	SWEEP BUSY					S BUS ERROR	NO MEMORY	IN-OUT PAGE FAILURE	MB PARITY ERROR	CACHE DIRCTRY PARITY ERROR	ADDRESS PARITY ERROR	POWER FAILURE	SWEEP DONE	INTRUPT REQUEST	PRIORITY INTERRUPT ASSIGNMENT		

6–13 A 1 in any of these bits indicates that setting the listed flag will request an interrupt on the level assigned to the processor by bits 33–35 of the CONO.

19 The cache is currently undergoing a sweep.

24 A storage controller has signaled the processor that it has detected an error in its own operation or in information it has received over the S bus or from one of its storage modules. If the type of error is not identified by there also being a 1 in bit 25, 27 or 29, then the condition is either an incomplete cycle or a parity error in data sent to the memory (all data received by memory is written, even if bad). Controller flags for some of these conditions can be read by the diagnostic instruction discussed in the second part of this section.

25 The processor attempted to access a memory that did not respond within a preset time. This time is 68 μs on an extended KL10, 82 μs on a single-section KL10. The setting of this flag locks information about the attempted reference into ERA. Since a nonexistent memory supplies zero data, on read this error should be accompanied by a 1 in bit 27.

26 A page failure has occurred in an interrupt instruction, or a word with even parity has been received at AR from the E bus (the latter can be recognized only if the transmitting device generates a parity bit). An interrupt failure caused by an address break sets this flag instead of producing an address failure (§3.5).

NOTE

> A page failure in an interrupt instruction is regarded as a fatal error, and causes an interrupt instead of a page failure trap. The kernel program is expected to set up the interrupt instructions so that a software page failure simply cannot occur.

27 The buffer (MB) in memory control has received a word with even parity. The setting of this flag locks information about the reference into ERA.

28 A physical page number with even parity has been encountered in the cache directory. The setting of this bit turns off the cache, and it remains off until the flag is cleared by giving a CONO APR, with 1s in bits 22 and 28.

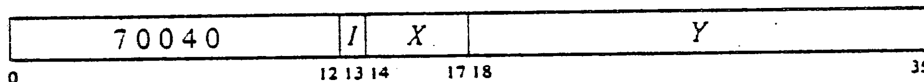
29 A storage controller has signaled that it has received an address with even parity from the processor. The parity check actually encompasses both the address and the control signals that accompany it on the S bus. The setting of this bit locks information about the attempted reference into ERA.

30 Ac power has failed. The program should save PC, the flags, mode information and fast memory in storage, update the accounting meters, validate the entire cache, and halt the processor. Note that PC may point to an interrupt routine rather than the main program. After power is restored the front end must reboot the system, and the Monitor must reestablish the operating environment (§3.5).

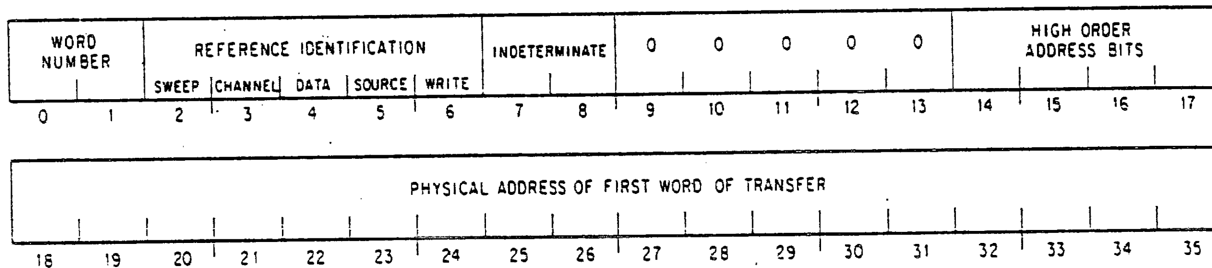
31 A cache sweep has been completed.

32 Some processor flag is currently requesting an interrupt, i.e. some flag in bits 24–31 is set and has been enabled to interrupt as indicated by a 1 in the corresponding position in bits 6–13.

RDERA **Read Error Address Register** (BLKI PI,)



Read the contents of the error address register into location E. If No Memory, MB Parity Error or Address Parity Error is set, ERA contains information about the reference corresponding to the first of those flags to be set as shown.



Bits 0–1 and 14–35 identify the physical location of the reference in which the error occurred. Bits 14–35 are the address of the specific memory reference made by the program or whatever. If the reference required only a single transfer, that address is the error address. But if the reference triggered a group transfer, bits 14–35 are the address of the first reference chronologically in the group, and bits 0 and 1 give the number of the word on which the error actually occurred. Note that word numbers are in physical, not chronological, order.

Information given in bits 2–6 identifies the reference. A 1 in bit 2 or 3 respectively means the reference was made for a cache sweep or a channel transfer. Bit 6 indicates the memory function being performed for the reference, where the read and write parts of a read-pause-write are separately

indicated by 0 and 1. Bits 4, 5 and 6 together identify the source of the data for the transfer or attempted transfer (on write the word is always going to storage).

Bits 4–5	*Source with 0 in bit 6*	*Source with 1 in bit 6*
00	Storage for any read or read-pause-write	Channel status
01		Channel data
10		AR
11	Cache for channel read or TOPS–10 page refill	Cache writeback

ERA retains the same information until the program clears the locking flags by giving a CONO APR,2260*P*. Of course only flags that are set actually need be cleared, and the routine that responds to errors should consider and clear all set flags. To facilitate diagnosis from the front end, the master reset does not clear ERA. Hence if need be, the front end can give diagnostic functions that reset the KL10 and then read ERA.

The processor includes provision for forcing bad parity to check the error detection logic. Bits 18–20 of a CONO PI, (§3.1) respectively cause even parity to be generated for an address sent to memory, a data word available from AR, and a page number entered into the cache directory. Where the data error shows up depends on where the word is sent from AR. Which errors are being forced can be seen by checking the flags in the same bits of a CONI PI,.

Programming Cautions. When handling parity error or nonexistent memory interrupts, the programmer should beware of the following.

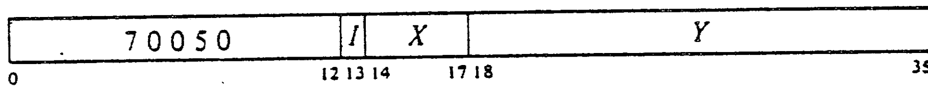
- An incorrect word from memory to AR or ARX can result in both a page failure and an interrupt. In general the page fail trap to the Monitor can be expected to occur slightly ahead of the interrupt.
- Should an error flag be set while another interrupt request is being processed, the system would handle the lower priority interrupt before getting to the processor interrupt. This means PC may be pointing to a lower level interrupt routine rather than the program level at which the error occurred. Remember that during request processing, the interrupt system is otherwise static and the program continues.
- Even without inadvertent interference from another level, it is quite likely the processor will perform one or perhaps two more instructions between the time the error flag sets and its interrupt starts. Hence even though PC is at the correct program level, it may well be pointing to the first or second instruction following the one in which the error occurred.
- A processor error interrupt that switches over to a lower priority level should not return to the interrupted program, as the error may simply recur, producing a second processor interrupt before the error-handling interrupt for the first. This could happen because PC is actually pointing to the offending instruction, but beyond that, one error often begets another

— consider the case of PC counting into a nonexistent memory. In any event, it is generally not worthwhile to return to any program without first finding out what went wrong.

S Bus Diagnostic Cycle

Ordinarily the S bus is used for the processor to reference memory. But the S bus also has a diagnostic cycle that allows the processor to communicate with the memory controllers rather than to access a particular location. The diagnostic cycle is initiated by the processor giving a special instruction that sends a function word to a controller and receives a word of error and diagnostic information back from it.

SBDIAG **S Bus Diagnostic Function** (BLKO PI,)



Send the contents of location E as a function word over the S bus to the controller specified by bits 0–4, and read the return word for the function from that controller into location E+1. Which function a word represents is indicated by its code in bits 31–35.

PART 4

603A SCHEDULER/SWAPPER PLM

SCHED

Program Logic Manual for Scheduler and Swapper

Date: February 1978
File: SCHED2,4,6,7.RNO
Version: 1

The information in this document is subject to change without notice and should not be construed as a commitment by Digital Equipment Corporation. Digital Equipment Corporation assumes no responsibility for any errors that may appear in this document.

The software described in this document is furnished under a license and may be used or copied only in accordance with the terms of such license.

Digital Equipment Corporation assumes no responsibility for the use or reliability of its software on equipment that is not supplied by DIGITAL.

Copyright © 1978 by Digital Equipment Corporation

CONTENTS

Page

CHAPTER 1 INTRODUCTION 1-1

1.1 OPERATION MODES 1-1
1.2 OBJECTIVES 1-2

CHAPTER 2 OVERVIEW OF SCHEDULER OPERATION 2-1

2.1 PROCESSOR QUEUES 2-1
2.2 LONG-TERM WAIT QUEUES 2-2
2.3 SPECIAL QUEUES 2-3
2.4 TIME SLICE 2-4
2.4.1 PQ2 Time Slice, Round Robin Mode 2-4
2.4.2 PQ2 Time Slice, Class Scheduler Mode 2-5
2.4.3 PQ1 Time Slice 2-6
2.4.4 HPQ Time Slice 2-7
2.5 SCHEDULING SCAN AND ASSIGNMENT OF SHARABLE RESOURCES 2-7

CHAPTER 3 DETAILED DESCRIPTION OF THE SCHEDULER 3-1

3.1 SCHEDULER ASSEMBLY 3-1
3.2 CALLING THE SCHEDULER 3-1
3.3 NXTJOB TO NXTJBX SECTION 3-2
3.4 NXTJB1 TO CKJB0A SECTION 3-3
3.5 CKJB0A TO CKJB5 SECTION 3-4
3.6 CKJB5 TO CKJB7 SECTION 3-4
3.7 CKJB7 TO SCHED SECTION 3-5
3.8 SCHED TO QREQ SECTION 3-5
3.9 QREQ TO QCHNG SECTION 3-7
3.10 SUBROUTINE QCHNG 3-9
3.11 SUBROUTINE SETIPT 3-9
3.12 SUBROUTINES ZERIPT, CLRIPT, CLRIP1 3-10
3.13 SUBROUTINE ASICPT 3-10
3.14 SUBROUTINE TOBACK 3-10
3.15 SUBROUTINE QARNDT 3-10
3.16 QXFER TO DICLNK SECTION 3-10
3.17 SUBROUTINE DICLNK 3-15
3.18 SUBROUTINE IICLNK 3-15
3.19 SUBROUTINE DCCLNK 3-15
3.20 SUBROUTINES ICCLNK AND ICSLNK 3-16
3.21 SUBROUTINE INOLST 3-16
3.22 SUBROUTINE DLOLST 3-16
3.23 SUBROUTINE DLJILS 3-16
3.24 QSCAN THROUGH FSQFOR SECTION 3-16
3.25 FSQFOR THROUGH BQFOR SECTION 3-17
3.26 BQFOR THROUGH ISSFOR SECTION 3-18
3.27 ISSFOR THROUGH OSSFOR SECTION 3-19
3.28 OSSFOR THROUGH ILFOR SECTION 3-20
3.29 ILFOR THROUGH SAVSUM SECTION 3-20

CONTENTS (CONT.)

			Page
CHAPTER	4	DETAILED DESCRIPTION OF SWAPPER	4-1
	4.1	SWAP TO SWAP1 SECTION	4-2
	4.2	SWAP1 TO FININ0 SECTION	4-3
	4.3	FININ0 TO INERR SECTION	4-3
	4.4	INERR TO FINOUT SECTION	4-4
	4.5	FINOUT TO SWP1 SECTION	4-5
	4.6	SWP1 TO FIT1 SECTION	4-5
	4.7	FIT1 TO OUTERR SECTION	4-7
	4.8	OUTERR TO SWPREC SECTION	4-8
	4.9	SWPREC SUBROUTINE	4-9
	4.10	SWPRC1 SUBROUTINE	4-9
	4.11	ZCKZPN TO SCNJOB SECTION	4-9
	4.12	SCNJOB TO FORCE0 SECTION	4-10
	4.13	FORCE0 TO SWAP0 SECTION	4-12
	4.14	SWAP0 TO NOFIT SECTION	4-13
	4.15	NOFIT TO NOFITZ SECTION	4-14
	4.16	NOFITZ TO ZERFIT SECTION	4-15
	4.17	ZERFIT TO NOFORC SECTION	4-15
	4.18	NOFORC TO SWAP1 SECTION	4-15
	4.19	CHGSWP TO CHG1 SECTION	4-15
	4.20	CHG1 TO UNSWAP SECTION	4-16
	4.21	UNSWAP TO RTNDSP SECTION	4-16
	4.22	RTNDSP TO GIVBKH SECTION	4-16
	4.23	GIVBKH TO XPAND SECTION	4-16
	4.24	XPAND TO XPANDH SECTION	4-17
	4.25	XPANDH TO SCHED. SECTION	4-17
CHAPTER	5	SCHEDULING PARAMETERS	5-1
	5.1	PROCESSOR QUEUE TIME SLICES	5-1
	5.1.1	PQ1 Time Slice	5-2
	5.1.2	PQ2 Time Slice	5-3
	5.1.3	HPQ Time Slice	5-3
	5.2	SWAPPING AND SCHEDULING FAIRNESS COUNTS	5-3
	5.3	IN-CORE FAIRNESS FACTOR	5-4
	5.4	CLASS QUOTAS AND MICROSCHEDULING INTERVAL	5-4
	5.5	BACKGROUND BATCH PARAMETERS	5-5
	5.6	RESPONSE FAIRNESS FACTOR	5-5
	5.7	AVERAGE SWAP TIME	5-6
	5.8	JOB CLASS	5-6
	5.9	CLASS RUNTIME	5-6
CHAPTER	6	DETERMINATION OF PARAMETERS FOR SCHEDULER	6-1
	6.1	PROCESSOR QUEUE TIME SLICES	6-2
	6.1.1	PQ1 Time Slice	6-3
	6.1.2	PQ2 Time Slice	6-4
	6.1.3	HPQ Time Slice	6-5
	6.2	SWAPPING AND SCHEDULING FAIRNESS COUNTS	6-5
	6.3	IN-CORE FAIRNESS FACTOR	6-7
	6.4	CLASS QUOTAS AND MICROSCHEDULING INTERVAL	6-7
	6.5	BACKGROUND BATCH PARAMETERS	6-7
	6.6	RESPONSE FAIRNESS FACTOR	6-7
	6.7	AVERAGE SWAP TIME	6-8

CONTENTS (CONT.)

			Page
	6.8	JOB CLASS	6-8
	6.9	CLASS RUNTIME	6-8
CHAPTER	7	DETAILED DESCRIPTION OF SCHED. MONITOR CALL	7-1
	7.1	SCHED. TO SCDQTA SECTION	7-1
	7.1.1	Function 0	7-1
	7.1.2	Function 2	7-1
	7.1.3	Function 3	7-2
	7.1.4	Function 5	7-2
	7.1.5	Function 6	7-3
	7.1.6	Function 7	7-3
	7.1.7	Function 9	7-3
	7.1.8	Function 10	7-3
	7.1.9	Function 11	7-3
	7.1.10	Function 12	7-3
	7.1.11	Function 13	7-3
	7.1.12	Function 14	7-4
	7.1.13	Function 15	7-4
	7.1.14	Function 16	7-4
	7.1.15	Function 17	7-4
	7.1.16	Function 18	7-4
	7.1.17	Function 19	7-4
	7.1.18	Function 20	7-5
	7.1.19	Function 21	7-5
	7.1.20	Function 22	7-5
	7.1.21	Function 23	7-5
	7.1.22	Function 24	7-5
	7.2	SCDQTA TO SCDQT7 SECTION	7-5

FIGURES

FIGURE	3-1	In-core Protect Time (.PDIPT)	3-2

TABLES

TABLE	2-1	Wait-State Codes	2-2
	2-2	Long-Term Wait Queues	2-3
	2-3	Special Queues	2-4
	3-1	Unrunnable Bit Settings	3-4
	3-2	Primary and Secondary Scan Tables	3-5
	3-3	QBITS TABLE	3-8
	3-4	Example of Codes Set For Transfer Tables	3-12
	3-5	Possible Codes	3-17
	4-1	Important Flags and Data Items	4-1
	4-2	Primary and Secondary Scan Swapping Tables	4-6
	5-1	Default Values of Swapping and Scheduling Fairness Counts	5-3
	6-1	In-Core Protect-Time Parameter Values	6-2
	6-2	PQ1 Quantum Runtime Parameter Values	6-2
	6-3	PQ2 Quantum Runtime Parameter Values	6-3

CONTENTS (CONT.)

		Page
6-4	HPQ Quantum Runtime Parameter Values	6-3
6-5	Percent of PQ1 Jobs Blocking to Long-Term Wait as Function of Time	6-4
6-6	Example of Effect of Incorrect Parameters	6-5
6-7	Default Values for Swapping and Scheduling FairnesC Counts	6-5
6-8	Example of Effect of Fairness Counts	6-6

CHAPTER 1

INTRODUCTION

The DECsystem-10 scheduler provides several response levels for long-term, short-term, and high-priority computing needs. Also, numerous scheduling parameters and two different modes of operation provide flexibility in the scheduling policy.

1.1 OPERATION MODES

There are two ways to operate the scheduler: Round Robin [1] mode and Class Scheduler mode.

Round Robin mode means that each job in the long-term processor queue (called PQ2) receives an equal share of the resources of the system. In other words, each job receives the full attention of the system for a short interval called a time slice. When the time slice allotted to a job expires, the job goes to the back of the queue. Then, the full attention of the system turns to the next job in the queue.

Round Robin mode gives good turnaround time to small jobs even though there are large jobs in the system. In addition, it gives each job an equal chance to use the system resources. Each job receives its 'fair share' of the system. Therefore, no job, regardless of its makeup, can take over the system.

Class Scheduler mode means that each job in PQ2 receives a share of the resources of the system. However, unlike Round Robin mode, each of these shares is not necessarily equal. Instead, each job in PQ2 is assigned to a class for which the system administrator sets a quota of system resources. The higher the quota, the more often the class is scanned for scheduling and swapping. In Class Scheduler mode, all jobs in PQ2 are also stored in a set of subqueues by class. As jobs expire their time slices, they go to the back of PQ2 and to the back of the subqueue for their class. This action gives Round Robin operation within the classes. Also, each class is swapped in and scheduled depending on its class quota.

The class quota consists of a primary percentage and a secondary allocation. The primary percentage is the amount of resources allotted to the class. The secondary allocation is the amount of leftover resources allotted to the class. Leftover resources occur when some of the classes do not use all of their primary percentages.

---

[1] Kleinrock, L., "Timeshared Systems: A Theoretical Treatment," Journal of the ACM, Vol. 14, No. 2, 1967, pages 242-261.

The system administrator may define any one of the classes as a background batch class. Background batch jobs do not run unless there are no runnable jobs in any of the other classes. Although normally the background batch class has a zero primary percentage and a zero secondary allocation, this is not a restriction. In fact, the primary percentage and the secondary allocation have the same meaning for the background batch class as for any other class. A nonzero primary percentage forces background batch jobs to run a certain percentage of the time. In addition, a nonzero secondary allocation gives background batch jobs a proportion of the leftover time.

1.2 OBJECTIVES

The overall design objectives of the scheduler are listed in the following.

1. Provide for sharing computer time among jobs with long-term computing needs.
2. Provide fast response time for interactive jobs.
3. Provide very fast response time for real-time jobs.
4. Provide for efficient use of all of the system resources.

Objective 1 above applies to jobs with long-term computing needs. For example,

- Compilation of FORTRAN, COBOL, ALGOL, and BASIC programs
- Execution of mathematical and statistical programs
- Execution of programs for sorting, merging, and/or file storage and retrieval

Objective 2 applies to jobs that require fast response time for interactive jobs. For example,

- A user editing a file
- A user updating a database

In this case, each time the user ends a line sending his input to the system, he expects to receive a response within a matter of seconds (preferably 1 to 4 seconds).

If the scheduler must complete a full cycle through PQ2 before responding, it cannot reliably achieve this optimum 1- to 4-second reponse time. The time required to make a complete cycle through PQ2 can depend on the character of the jobs in the queue, and does depend on the number of jobs in the queue. On a heavily loaded system, the response time can easily exceed 10 seconds. Clearly, this response time is unacceptable to the interactive user. Therefore, the scheduler provides a priority processor queue called PQ1.

Normally, the scheduler selects jobs in PQ1 before it selects any jobs in PQ2. In this way, the scheduler can meet the goal of fast response time without wasting CPU time and without allowing those jobs that do not require interactive response to suffer.

Jobs enter PQ1 at the back of the queue and are assigned a time interval to remain in the fast-response queue. If a job ends before its time slice is exhausted, it leaves the processor queues. If a job does not finish its task before its time slice is exhausted, it goes to the back of PQ2. Thereafter, until the job ends, it receives the same attention as any PQ2 job.

Typically, there are more jobs in PQ1 and PQ2 than can fit in memory at any one time. Therefore, some of the jobs must be stored temporarily on a high-speed swapping device, such as a disk or a drum. As the jobs that are in memory are requeued, they become eligible to be swapped out. Then, as space becomes available in memory, jobs are swapped into memory by scanning the processor queues and swapping in the highest priority job that is not already in memory. Jobs in PQ1 receive priority over jobs in PQ2. This is true both for swap-in and for allocation of resources once they have been swapped in.

Objective 3 applies to jobs that require very fast response time and better performance than PQ1. For these jobs, the scheduler provides a final set of processor queues called high-priority queues. There may be up to 15 high-priority queues, called HPQ1 through HPQ15.

The kinds of jobs that would use the high-priority queues are, for example,

- Card-reader and line-printer spoolers
- Real-time data acquisition

These programs must be swapped to disk when the physical devices are not busy. This action provides more room for other terminal jobs. When there are cards to read or lines to print, these jobs must be swapped in as soon as possible and remain in memory while in service. Also, these jobs must be able to get CPU attention instantaneously to fill and empty buffers of input and output. The scheduler achieves very fast swap-in and instantaneous access to the CPU by swapping in and scheduling resources (such as CPU, and so forth) for all HPQs ahead of PQ1 and PQ2.

Jobs in high-priority queues can require any amount of system resources up to and including 100% of the system. Whatever resources remain unused are then available for jobs in PQ1 and PQ2. In the example of the card-reader-stacker and the line-printer-spooler jobs, a certain amount of memory is dedicated to these jobs when they are active. Therefore, the amount of user memory area available to all other jobs is correspondingly reduced.

As far as CPU time is concerned, jobs in high-priority queues are I/O bound and, therefore, use very little. Because of this, most of the CPU (over 95%) is available for other user jobs.

Objective 4 applies to all jobs. The scheduler runs the system as efficiently as possible within the constraints imposed by the first three objectives, including

- Balancing the percentage of CPU versus I/O jobs in core memory so that multiprogramming is most effective
- Balancing the percentage of PQ1 versus PQ2 jobs in memory so that a good compromise is achieved between throughput and short-term response

CHAPTER 2

OVERVIEW OF SCHEDULER OPERATION

All jobs in the system are maintained in a master set of queues. Each job is in one and only one of the queues. For convenience, the master set is divided into two logical groups: processor queues and long-term wait queues.

2.1 PROCESSOR QUEUES

The processor queues are the high-priority queues (HPQs), PQ1, and PQ2. Each of these is described in the following.

HPQs (Up to 15 levels, called HPQ1 through HPQ15) contain jobs that require real-time response, such as the line-printer-spooler and the card-reader-stacker programs.

PQ1 Contains jobs that require fast response, such as those that conversationally interact with the user.

PQ2 Contains jobs that require long-term computing, such as those that compile FORTRAN, COBOL, ALGOL, and BASIC programs. For the class scheduler, all jobs in PQ2 are also in the class subqueues.

Jobs in the processor queues either are ready to execute on the processor or are in various short-term wait states (such as waiting for disk I/O or I/O from other high-speed devices). These short-term wait states are too small to warrant requeueing the job, because it would then lose its position in the processor queues and be marked for swap-out. A wait-state code indicates which jobs are runnable and which are waiting. Table 2-1 lists the wait-state codes.

Table 2-1
Wait-State Codes

Code	Meaning
IOW	I/O wait for unit record, reader, printer, and so forth.
DIOW	Disk I/O wait (RP02, RP03, RP04, and so forth).
AU	Waiting for system interlock to clear to alter UFD on file structure.
MQ	Waiting for monitor buffer (to read file retrieval pointers, for example).
DA	Waiting for system interlock to clear to access SAT table to get an allocation of disk blocks on the file system.
CB	Waiting for system interlock to clear to access core block allocation routine (to get space for a DDB or file access table from the core block pool, for example).
D1,D2	Waiting for DECtape controller.
DC	Data controller wait.
CA	Core allocation (lock) wait.
PIOW	Paging I/O wait.
PS	Paging I/O satisfied.
EV	Execute virtual-memory wait.
NAP	Short-term sleep.

2.2 LONG-TERM WAIT QUEUES

Table 2-2 lists the long-term wait queues.

Table 2-2
Long-Term Wait Queues

Queue	Meaning
CMQ	Command Wait Queue. You have typed a monitor command that cannot be executed until the job is in memory, and the job is not in memory. This produces a higher priority swap-in, then requeues to PQ1.
TIOWQ	Teletype I/O Wait Queue. Waiting for you to type in or waiting for the device to print output already sent to it. This includes pseudoteletypes.
SLPQ	Sleep Queue. The job has executed the SLEEP monitor call and requested to sleep for some interval, or it has executed the HIBER monitor call and requested to sleep until the WAKE monitor call is executed by another job, or some specified condition has been satisfied.
JDCQ	DAEMON Wait Queue. The job is waiting for service by DAEMON (for example, to record accounting data or to perform error logging).
STOPQ	Stop Queue. The user has typed a CTRL/C, for example, to stop his job.
NULQ	Null Queue. All job slots must be accounted for in the queue structure. This queue contains the numbers of the job slots not currently in use (including jobs that have CORE zeroed).
EWQ	Event Wait Queue. Waiting for a magnetic tape controller, for example.

Within priority wait queues, the jobs are ordered by priority.

The first job in the queue has the highest priority. In the long-term wait queues, the order of the jobs is immaterial.

The master queues (including the subqueues) are each separated into two mutually exclusive lists: one for jobs that have core (JBTADR ≠ 0) and one for jobs that do not have core. This significantly reduces overhead, because various scans use only one set of queues. For example, the scheduling scans do not look at jobs that do not have core.

2.3 SPECIAL QUEUES

A number of special queues are used to improve communication between the scheduler and the swapper, and to properly handle background batch jobs. Jobs in the special queues are also in the master queues.

Table 2-3
Special Queues

Queue	Meaning
JIL	Queue of PQ2 jobs that have just been swapped in, that is, those jobs that have not yet expired 1 time slice since they were swapped in. The queue is divided into two lists: one for timesharing jobs and one for background batch jobs.
OLS	Queue of PQ2 jobs that are eligible to be swapped out. That is, those jobs that have expired at least 1 time slice. The queue is divided into two chains: one for timesharing jobs and one for background batch jobs.

2.4 TIME SLICE

The time slice controls the movement of jobs within the processor queues. The time slice is defined as two separate parameters: quantum runtime and in-core protect time. Quantum runtime is decremented as the job uses the processor. In-core protect time is decremented whether or not the job uses the processor, as long as the job has been scanned by the scheduler.

CPU-bound jobs generally expire quantum runtime. I/O-bound jobs generally expire in-core protect time. When either parameter expires, the job is considered to have ended its time slice.

The time slice is assigned when a job is swapped in or when it initially begins to run. It is reassigned whenever a job is requeued to a new position in the processor queues.

Within its time slice, a job may enter and leave various short-term wait states without being requeued to a new position in the queues. Requeues in and out of short-term wait involve only a change in the wait-state code; no queue transfer takes place.

Jobs that block to any long-term wait state are physically requeued to one of the long-term wait queues. They lose their place in their current processor queues and are not eligible to be swapped in or scheduled until they leave the long-term wait state. However, their positions in the long-term wait queues are immaterial. Jobs become runnable and leave the long-term wait queues according to their individual job characteristics. Most jobs are requeued to the back of PQ1.

Jobs in the processor queues that expire their time slices are requeued to the back of the processor queues. Primarily, the queue that the job is currently in determines its destination and the queue assignment of a new time slice.

2.4.1 PQ2 Time Slice, Round Robin Mode

In Round Robin mode, the PQ2 time slice gives each job in succession an equal opportunity to use the system resources.

PQ2 jobs are kept in two chains. Jobs that are in memory are in the in-core chain. Jobs that have been swapped out are in the out-core chain. Both chains are ordered lists, with the highest priority jobs at the front of the chain.

When the jobs are swapped in, they are assigned a time slice and are linked to the back of the in-core chain. As jobs are scheduled and expire their time slices, they are requeued to the back of the in-core chain. At this point, they become eligible to be swapped out. Jobs that have been swapped out go to the back of the out-core chain. Jobs that have been swapped in come from the front of the out-core chain. This action allows proper Round Robin cycling between the two chains.

Jobs that have not yet expired 1 time slice are kept in a special list in the order in which they were swapped in. They are scheduled to run ahead of jobs waiting to be swapped out. This is consistent with the Round Robin algorithm, and provides the best short-term response time.

Jobs are swapped out in the order in which they expire their first time slices. As they expire, they are placed in the swap-out list and are removed from the just-swapped-in list.

While the jobs are waiting to be swapped out, they cycle around the in-core chain in Round Robin fashion. The jobs are assigned a time slice and, as they expire it, they are requeued to the back of the in-core chain. When there is no demand for swapping, core scheduling around the in-core chain results.

2.4.2 PQ2 Time Slice, Class Scheduler Mode

In the Class Scheduler mode, the jobs in PQ2 are given an opportunity to use system resources in proportion to the size of their class quotas. The time slice allows Round Robin cycling within a class.

All jobs in PQ2 are also stored in a set of subqueues by class. The subqueues are ordered lists, with the jobs of the highest priority at the front of the subqueue. Like PQ2, the subqueues have in-core and out-core chains.

When jobs are swapped in, they are assigned a time slice. As jobs are scheduled and expire their time slices, they are requeued to the back of the PQ2 in-core chain, and to the back of the in-core chain of the subqueue for their class. They are then eligible to be swapped out.

Jobs that have been swapped out go to the back of the PQ2 out-core chain and to the back of the out-core subqueue chain for their class. Jobs that have been swapped in come from the front of the subqueue out-core chain. This allows Round Robin cycling within the subqueues.

The order in which the subqueues are scanned for swap-in depends on the primary percentage and the secondary allocations defined by the system administrator. The swapper operates with a 100-interval swap cycle. At each interval, one of the classes (that is, subqueues) is the first one scanned for swap-in. The number of times a class is scanned first depends on the size of its primary percentage. That is, a class with a primary percentage of 10% will be the first one scanned in 10 out of 100 intervals.

If no jobs are eligible to be swapped in from the primary class, the swapper selects one from a secondary class. The choice of secondary class depends on the size of the class's secondary allocations. The larger a class's secondary allocation, the higher its probability of selection. If the secondary class selected also has no jobs, another selection is made from the remaining secondary classes. If no jobs are found in the primary and secondary classes, the swapper considers a background batch job scan.

Background batch jobs can only be swapped in at a certain rate to prevent thrashing. The system administrator specifies this rate through the SCDSET program. (See Section 6.5.)

If jobs exist in sufficient numbers in all classes, the swapping algorithm fills memory with jobs in proportion to their primary percentages. This allows the scheduler to schedule accurately while still achieving good short-term response times.

Jobs that have not yet expired 1 time slice are kept in a special queue in the order in which they were swapped in. To guarantee a minimum level of short-term response, the list of jobs just swapped in must be scanned for scheduling a certain percentage of the time. The response fairness factor determines the amount of time that the list of jobs just swapped in is scanned. The system administrator sets the response fairness factor with the SCDSET program.

The class scheduling scan is made up of 100 intervals. The microscheduling parameters define the length of these intervals. The system administrator sets the microscheduling parameters with the SCDSET program.

Each time that the microscheduling interval expires, the scheduler moves to the next class in the primary scan table. The table contains 100 entries, each representing the primary class for that interval. The scheduler builds a complete subqueue scan table with all classes by starting with the primary class and selecting the second, third, ..., nth class, depending on the size of the secondary allocations. The scan table determines the order in which the subqueues are scanned throughout the current microscheduling interval.

Jobs are swapped out in the order in which they expire their first time slice. As they expire, they are placed in the swap-in list and are removed from the just-swapped-in list.

While waiting to be swapped out, the jobs cycle around the in-core chain for their subqueue in a Round Robin fashion. They are assigned a time slice and, as they expire it, they are requeued to the back of their subqueue in-core chain. When there is no demand for swapping, this results in class core scheduling.

2.4.3 PQ1 Time Slice

PQ1 jobs that expire their time slices are requeued to the back of PQ2. They are reassigned the normal PQ2 quantum runtime and they retain whatever in-core protect time they have remaining. They are not marked to be swapped out. This allows jobs in PQ1 to have very good response for a short period of time. Thereafter, if they continue to run, they may remain in core at least as long as a PQ2 job.

PQ1 jobs are ahead of PQ2 jobs in the normal swap-in and scheduling scans. To prevent PQ1 jobs from totally taking over the system, there is a set of swapping and scheduling fairness counts. This means that when a PQ1 job has been selected a certain number of times in a row, the fairness counts force PQ2 to be scanned first.

2.4.4 HPQ Time Slice

HPQ jobs that expire their time slices are requeued to the back of the corresponding HPQ. If they expire their quantum runtimes, they are assigned new quantum runtimes and retain whatever in-core protect times they have remaining. If they expire their in-core protect times, they are assigned new quantum runtimes and in-core protect times. They are then eligible to be swapped out.

The HPQ time slice defines how quickly the system can switch from one HPQ job to another. HPQ quantum runtime is, therefore, a very small number of ticks. HPQ in-core protect time is not very meaningful because only another HPQ job can force an HPQ job to be swapped out. It is unlikely that any installation would have more HPQ jobs in execution at once than could fit in memory.

2.5 SCHEDULING SCAN AND ASSIGNMENT OF SHARABLE RESOURCES

The scheduling scan searches the processor queues (in the order of priority) for a job to run. Then, it selects the first runnable job it finds in the scan. Jobs with a zero short-term wait-state code are runnable. So are jobs waiting for sharable resources, if the resources are currently available. To assign a resource, the scheduler clears the job's short-term wait-state code and marks the resource in use. This procedure causes sharable resources to be assigned to the job with the highest priority.

If a high-priority job needs a resource held by a low-priority job, the scheduler will attempt to run the lower priority job until it gives up the resource. This feature is especially important in the class scheduler.

Refer to Chapter 3 for a detailed description of the scheduler.

CHAPTER 3

DETAILED DESCRIPTION OF THE SCHEDULER

This chapter describes the scheduler at the level of the macro code. If you require only general knowledge of the scheduler, it is not necessary that you read this chapter. The labels referenced in this chapter are in the scheduler monitor module, SCHED1.

This chapter discusses the following issues.

- Jobs that perform GETSEGs release their high segments with their low segments still in memory. The swap-in scan must search the in-core chains to link to the new high segments.
- The class scheduler can swap and schedule fixed classes by having a zero secondary allocation. The class scheduler can also perform fixed swapping with nonfixed scheduling.
- Background batch imposes some complexities on the swap-in, swap-out, and scheduling scans.
- The swap-out scan selects jobs in the long-term wait queues ahead of jobs in the processor queues.
- The scheduling scan has a number of fairness counts that control the way in which the master queues and special queues are scanned.

3.1 SCHEDULER ASSEMBLY

The system administrator assembles the scheduler in one of two modes, depending on the value of the assembly switch FTNSCHED. When the system administrator sets FTNSCHED to 0, the Round Robin mode scheduler is assembled.

In Round Robin mode, there are no scheduler classes and there is no SCHED. monitor call. When the system administrator sets FTNSCHED to -1, the Class Scheduler mode scheduler is assembled, which includes the code for the SCHED.

3.2 CALLING THE SCHEDULER

The scheduler is called into action when one of the following occurs:

1. The clock ticks (an interval of 1/60th of a second has elapsed).

2. The current job becomes unrunnable for any reason (for example, long-term wait, short-term wait, or error).

3. The null job is running and some job becomes runnable (for example, finished with disk I/O).

4. An HPQ job of higher priority than the current job becomes runnable.

5. A job that has been chosen to be swapped out has just released all disk-sharable resources.

The entry points for the scheduler are NXTJOB for CPU0, and NXTJB1 for CPU1.

3.3 NXTJOB TO NXTJBX SECTION

This section of code decrements the in-core protect times and requeues jobs when their in-core protect times expire.

In-core protect times are maintained only when there are enough runnable jobs to require some of them to be swapped out. Whenever a specified period of time (SCDCOR) elapses during which no runnable jobs are swapped out, the scheduler assumes that core is not scarce and stops making decisions based on core use.

In-core protect time is stored in the PDB word labeled .PDIPT. (See Figure 3-1.)

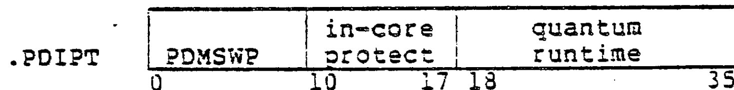


Figure 3-1 In-core Protect Time (.PDIPT)

PDMSWP is the sign bit of the same word, which may be 0 or 1 as defined below.

PDMSWP = 0 The job may not be swapped.

PDMSWP = 1 The job may be swapped.

In-core protect times are decremented every other clock tick unless no runnable jobs are being swapped out. This is done only on the odd ticks to save overhead. When core is not scarce, in-core protect times are not decremented at all (again, to save overhead). The core-is-scarce timer (CORSCP) is decremented at this time. The parameters for assigning in-core protect time and CORSCD are scaled in units of 2 ticks.

Jobs in the processor queues are not decremented unless they have been scanned by the scheduler. This prevents jobs from being swapped in and then swapped out again without having a chance to run. This would occur if a job were swapped in when one or two heavy CPU-bound jobs of higher priority were already in core. In this case, although the newly swapped job would be in core, it would not get a chance to run until these jobs had completed their time slices.

DETAILED DESCRIPTION OF THE SCHEDULER

The table DCSCAN defines the set of queues to be scanned. This table contains an entry for each queue that is allowed to retain its in-core protect time. These queues are listed below.

EWQ SLPQ PQ2 PQ1 HPQs

The queues are scanned from the back so that jobs being requeued cannot be requeued twice. PQ2 is scanned ahead of PQ1 for the same reason.

At NXTJOB, if the clock has not ticked, go to NXTJB1. On the even tick, go to NXTJBX. On the odd tick, set up to scan the queues to be decremented. On every clock tick, decrement CORSCD, which is the in-core protect time.

At NXTJBL, if there are no more queues to be scanned, go to NXTJBX. Otherwise, if there are no jobs in the next queue to be scanned, go to NXTJBG.

At NXTJBA, remember the successor to the job being scanned in case of requeue.

If the job being scanned is in a processor queue but has not already been scanned, go to NXTJBF. Otherwise, clear the scanned-by-scheduler bit (JS.SCN) and decrement the in-core protect time CORSCD.

If the in-core protect time is no longer positive, set the job-is-swappable bit (PDMSWP). If the job is in command wait (CMWB=1) or is waiting for requeue (JRQ=1), go to NXTJBD. Otherwise, assign a new in-core protect time so the job will cycle. Then, if the job is in a processor queue, requeue it with subroutine QXFER using transfer table QTIME. Finally, go to NXTJBF.

At NXTJBD, the in-core protect time is set to a zero. At NXTJBE, deposit a new in-core protect time.

At NXTJBF, pick up the remembered link to the next job. If the link is a job, go to NXTJBA. If the link is a queue header, go to NXTJBL and scan the next queue.

At NXTJBX, in Class Scheduler mode, execute the subroutine SCDQTA to check for the end of the microscheduling interval.

3.4 NXTJB1 TO CKJB0A SECTION

This section of code determines whether or not the current job has become unrunnable and checks for the end of the time slice when the quantum runtime has expired.

If the current job is the null job, exit to CKJB1 to requeue all jobs with JRQ set. If the current job has executed an HPQ monitor call, if it is waiting for DAEMON (JDC=1 or JS.DEP=1), or if it has been requeued out of the processor queues, exit to CKJB0A to requeue the current job.

If the current job is runnable, check the in-core protect and quantum runtime for expiration and requeue the job for time-slice expiration (subroutine QARNDT), if required. Then, go to CKJB1.

If the current job is unrunnable, go to CKJB0 to determine if the current job needs requeuing. Unrunnable is defined as any of the bit settings (in JBTSTS) listed in Table 3-1.

Table 3-1
Unrunnable Bit Settings

Bit Setting	Meaning
RUN = 0	Job does not want to run
JNA = 0	Job number not assigned
JXPN = 1	Job expanding
JERR = 1	Monitor detected error
SHF = 1	Waiting to shuffle or shuffling
SWP = 1	Waiting to swap or swapped out
Wait-state code ≠ 0	Job in wait state
JRQ = 1	Requeue requested

At CKJB0, mask out JXPN, SHF, and SWP. If the job does not need requeuing, exit to CKJB1. Otherwise, if the job does need requeuing, jump to CKJB0A.

3.5 CKJB0A TO CKJB5 SECTION

This section of code requeues the current job and/or all jobs in the system with JRQ equal to 1.

CKJB0A is entered if the current job needs requeuing. If the requeue bit is not set for the current job (JRQ = 0), go to CKJB3.

At CKJB1, if entered by the slave processor, go to CKJB5. Otherwise, requeue all jobs in the requeue chain. The requeue chain is a last-in-first-out linked list (JBTJRQ). The zero word is the header. Each entry contains the job number of the next job in the list. A zero entry indicates the end of the list. Note that the entries are deleted from the list before JRQ is cleared, which prevents entering the same job in the list twice.

At CKJB3, the actual requeue is done by subroutine QREQ. Loop back to CKJB1 to requeue any remaining jobs.

3.6 CKJB5 TO CKJB7 SECTION

This section of code checks to see if exec virtual memory has become available (EVAVAL ≠ -1), and if so, clears the wait-state code for any jobs waiting for it. The EV resource is required for all I/O devices that do not have data channels and, therefore, require that the monitor service routines be able to address user core with the EXEC page map. The resource being allocated is the EXEC page map slots.

At CKJB5, if entered from the slave, go to SCHED.

3.7 CKJB7 TO SCHED SECTION

This section of code determines whether or not SWAP/LOCK is called.

SWAP/LOCK is called only on CPU0, and only if one or more of the following conditions is true:

1. An HPQ job on disk became runnable.
2. The current job is the null job.
3. The clock ticked.

3.8 SCHED TO QREQ SECTION

This section of code selects the next job to run. It assigns sharable resources as required, and unwinds them from other jobs if necessary.

At SCHED, clear the potentially lost time flag and go to the processor-dependent scheduler.

For a single processor, go to SCHEDJ.

For a dual-processor entered by the master, go to MSCHED in CP1SER. For a dual-processor entered by the slave, go to SSCHED in CP1SER. MSCHED selects a job from the slave waiting for a monitor call. Alternatively, if there are no such jobs or if the fairness count (.COUFC) is greater than UFC0, it selects a job by the normal scan at SCHEDJ. SSCHED is a call to SCHEDJ.

At SCHEDJ, determine which scheduling scan table is to be used. If the scheduling scan did not reach the last queue in the scan recently enough (.CPSFC greater than or equal MFC), use the secondary scan table rather than the primary scan table. Table 3-2 contains the primary and secondary scan tables (in-core chains only).

Table 3-2
Primary and Secondary Scan Tables

Primary Scan Table (SSCAN)		Secondary Scan Table (SSCAN1)	
Queue	Routine	Queue	Routine
HPQs	IQFOR	HPQs	IQFOR
PQ1	IQFOR	PQ2	IRRFOR (Round Robin mode)
PQ2	IRRFOR (Round Robin mode)	PQ2	ISSFOR (Class Scheduler mode)
PQ2	ISSFOR (Class Scheduler mode)	PQ1	IQFOR
PQ2	IBBFOR (Class Scheduler mode)	PQ2	IBBFOR (Class Scheduler mode)

For the slave processor, the primary and secondary scan tables are interchanged.

If the swapper has selected a job to force out that has a disk-sharable resource (FORCEF not equal to 0), try to run that job until it gives up all resources, regardless of the job's actual queue position. (The disk-sharable resources are: AU, CB, DA, and MQ.) Otherwise, at SCHDJ1 call QSCAN to scan the processor queues in the order specified by the previously selected scan table. Jobs returned by the scan are processed by the following code from SCHEDB to SCHD1.

At SCHEDB, call DXRUN to see if a job is runnable on the calling CPU. If not, loop to scan for the next job (JRST (T2)).

Set the scan bit (JS.SCN) to 1. If the job has a zero wait-state code (meaning RUN), go to SCHEDC. The code from this point to SCHEDC assigns and unwinds sharable resources. (They are also assigned in CLOCK1 if available the instant a job asks for them.) The sharable resources involved are: AU, MQ, DA, CB, D1, D2, DC, and CA.

Sharable resources are not assigned to jobs that are swapped out (SWP=1), shuffling (SHF=1), expanding (JXPN=1), or need to be requeued (JRQ=1).

If the job needs a resource (identified by a wait-state code) and it is available (AVTBMQ≠0), go to SCHEDA and assign the resource. If the resource is not available, try to unwind it.

The code from UNWND1 to SCHEDA unwinds sharable resources from lower priority jobs so that they are available to higher priority jobs. The unwind process is to look for a job that has the resource that is desired and, if runnable, to run the job until it gives up the resource.

If the job holding the desired resource is not runnable because it also is waiting for a resource then:

1. If that resource is available, assign it and run that job.

2. If that resource is not available, look for the job that has that resource and repeat the unwind process (repeating to a maximum depth of 10, with an expected maximum of 3).

MQ represents a special problem because there is usually more than one monitor buffer. This means that there is more than one path to success. The routine investigates all paths and chooses the shortest. A path to success always ends with a job that is runnable. By running that job until it gives up all resources and by repeating the process for each job in the path, the original objective of freeing a given resource for a higher priority job is eventually achieved.

At SCHEDA, a job being forced out that had a disk-sharable resource (FORCEF=J) will not be given any new sharable resources after it has given up the resources that prevented it from being swapped (assessed by calling FLSDR). Otherwise, if you know that the job is runnable, go to SCHEDE and assign the resource.

At SCHEDE, assign the resource by clearing the wait-state code for the job. Also, do bookkeeping on AVTBMQ and USTBMQ.

At SCHEDC, the job being scanned is checked to be sure it is runnable (normal definition). In addition, if the job scanned is being forced out with a disk-sharable resource, it is considered unrunnable if it has given up the resource. If it has not given up the resource, the JXPN bit is ignored (as far as the job being runnable) because some other job may have expanded a high segment being shared.

If the job is runnable, it is selected to run. The scheduling fairness counts are updated depending on the queue the job is in. The selected job number is in AC J. The scheduler exits to CLOCK1.

If no runnable job is found by the scan, at SCHD1 scan the out-core chains of the processor queues until the lost-time flag is set (.CPPLT ≠ 0) or there are no more jobs to scan. This allows computation of lost time, using a small amount of processor time that otherwise would not be used. Set J to the NULJOB and exit to CLOCK1.

3.9 QREQ TO QCHNG SECTION

This section of code determines if a requeue requires a physical queue transfer, and if so, sets up the right half of AC U to the desired transfer table address (either directly or by indexing the QBITS table with the wait-state code). If no physical queue transfer is required, it performs the necessary bookkeeping for the requeue.

At QREQ, if CMWB, JDC, and JS.DEP are not all zero, go to QREQ1. Otherwise, if the run bit is off at QREQ0, go to QSTOPT. If none of the above special cases apply, call MSQRT to maintain dual-processor monitor call counts. Dispatch to one of eight different transfer routines using the left half of the QBITS table indexed by the wait-state code.

At QREQ1, if the command wait bit (CMWB) is set to 1 and the job is swapped out (SWP=1) or expanding (JXPN=1), set AC U to transfer table QCMW, and go to QXFER.

At QREQ2, if the job is requesting service from DAEMON (JDC=1 or JS.DEP=1) and the job does not have a disk-sharable resource, set AC U to the state code JDCQ and go to QJDCT. Otherwise, go to QREQ0.

The QBITS table has one entry for each wait-state code. (See Table 3-3.) The left half of each entry is the address of the transfer routine. The right half contains either the address of a transfer table (to be used by QXFER) or -1 if no physical queue transfer is required.

The content of QBITS depends on the value of the assembly parameters. Table 3-3 is a typical configuration.

Table 3-3
QBITS TABLE

Code	Left Half	Right Half
RN	QRNT	QRNW
WS	QWST	-1
TS	QTST	QTSW
DS	QDST	-1
PS	QPST	-1
AU	QAUT	-1
MQ	QMQT	-1
DA	QDAT	-1
CB	QCBT	-1
D1	QD1T	-1
D2	QD2T	-1
DC	QDCT	-1
EV	QEVT	-1
IOW	QIOWT	-1
TIOW	QTIOWT	QTIOWW
DIOW	QDIOWT	-1
PIOW	QPIOWT	-1
SLP	QSLPT	QSLPW
EW	QEWT	QEWW
NAP	QNAPT	-1
NUL	QNULT	QNULW
JDC	QJDCT	QJDCW
STOP	QSTOPT	QSTOPW

The following six routines perform bookkeeping and, where required, set up the right half of AC U to the desired transfer table address.

QRNT: Entry for jobs with zero wait state = runtime.

If the job is in PQ2 and is being requeued because it is changing subqueues (JS.CSQ = 1), go to QREQX.

Otherwise, go to QREQ3.

QPST: Entry for paging satisfied.

QWST: Entry for I/O wait satisfied.

QDST: Entry for disk I/O wait satisfied.

This routine checks to see if the job is in a processor queue, and if not, requeues it into PQ1 (QCHNG). It then clears the wait-state code and goes to QREQX.

QTST: Entry for teletype I/O wait satisfied.

This routine clears the wait-state code and goes to QREQ3.

QSLPT: Entry for jobs entering sleep.

QEWT: Entry for jobs entering event wait.

QREQ3: Common entry, various requeue procedures.

This routine sets up AC U from the right half of QBITS (transfer table address) and calls the queue transfer routine (QXFER). It then goes to QREQX.

QSTOPT: Entry for jobs that do not have run bit set.

This routine sets U to STOPQ unless the wait-state code (in U) is NULQ. Go to QREQZ.

QNULT: Entry for jobs going to NULQ.

QJDCT: Entry for jobs going to DAEMON queue.

QTIOWT: Entry for jobs going to TTY wait queue.

QREQZ: Common entry point.

This routine sets PDMSWP, indicating that the job may be swapped.

Go to QREQ3 (to finish requeue).

QREQ6: Common entry point, not labeled as such but includes NAP, all sharable resource waits, and all I/O waits except TTY.

This subroutine checks to see if the job is in a processor queue, and if not requeues it to PQ1 (QCHNG), then goes to QREQX.

QREQX: Exit for all requeue subroutines.

If the job being requeued is changing subqueues (JS.CSQ=1) and is still in PQ2, requeue it to the back of the appropriate subqueue with subroutine TOBACK.

Exit from QREQ.

3.10 SUBROUTINE QCHNG

Requeue a job to the back of PQ1. This subroutine is used to transfer a job into the processor queues if it is in some other queue (STOPQ, for example). It is required because the requeue logic for short-term wait states assumes that the job is already in the processor queues; however, in a few cases it is not. For example, a user types CTRL/C while in I/O wait and then later continues the job.

3.11 SUBROUTINE SETIPT

The SETIPT subroutine sets in-core protect time for the job to the minimum value, which is used after the expiration of the first time slice. (This may be different from values assigned at swap-in because initial quanta reflect the difficulty of swapping in large jobs. In this case, no swap-in has occurred.)

The value for minimum in-core protect time is installation dependent. A reasonable range is from 0.5 second to 5 seconds.

3.12 SUBROUTINES ZERIPT, CLRIPT, CLRIP1

Set PDMSWP to indicate that the job is eligible for swap-out.

3.13 SUBROUTINE ASICPT

Compute and store the in-core protect time based on the size of the job.

No in-core protect time is assigned if CORSCD is less than zero. In this case, it is assumed to be unnecessary because there is probably sufficient core for all running jobs.

3.14 SUBROUTINE TOBACK

Requeue jobs to the back of PQ2 and the subqueue.

3.15 SUBROUTINE QARNDT

Requeue the job because the time slice has expired.

If the job is currently in PQ1, requeue it to the back of PQ2, and then assign a new quantum runtime.

If the job is currently in PQ2, requeue it to the back of PQ2, and then assign a new quantum runtime and in-core protect time. Finally, mark the job eligible for swap-out.

If the job is in HPQ, requeue it to the back of the same HPQ, and then assign a new quantum runtime.

3.16 QXFER TO DICLNK SECTION

This routine performs all of the physical queue transfers. It is called with a job number in AC J and the address of a transfer table in AC U.

Transfer tables occur in two formats (depending on the right half of the first word).

1. Fixed-destination queue.

POSITION OPTION	QFIX
QUANTUM OPTION	NUMBER OF DESTINATION QUEUE

2. Destination queue determined by source queue, quantum runtime determined by job size.

POSITION OPTION	QLNKZ
QUANTUM OPTION	ADDRESS OF TABLE FOR DESTINATION QUEUE

Position Option:

0 = Requeue to beginning of queue

400000 = Requeue to end of queue

Quantum Option:

If negative = Do not assign quantum runtime

If positive = Format 1, address of word containing amount of quantum runtime to assign.
Format 2, quantum runtime is to be computed (within QLNKZ routine).

QFIX/QLNKZ Name of requeue routine.

At QLNKZ, the destination queue is determined by indexing into the specified destination queue table. At present, only one such table exists (QRQTBL in COMMON); it contains one entry for each processor queue.

INDEX	DESTINATION
HPQ	Same HPQ
PQ1	PQ2
PQ2	PQ2

f the transfer table requests computation of quantum runtime, it is alculated in routine CMPQRT as shown in the following.

$$\text{quantum runtime} = \left(\frac{\min(QMX, QAD+K*QML)}{QRANGE} \right)$$

QMX is taken from table QMXTAB by the destination queue; it is the largest quantum runtime permitted for that queue.

QAD is taken from table QADTAB by the destination queue; it is the base quantum runtime for all jobs.

K is the size of the job in K (1024 words).

QML is taken from table QMLTAB by the destination queue; it is a multiplier factor used to modify quantum runtime by job size.

QRANGE is used to scale the multiplier factor.

The tables QMXTAB, QADTAB, QMLTAB, and QRQTAB each have one entry per processor queue. This makes it easier to assign quantum runtimes differently for the processor queues (that is, HPQs, PQ1, and PQ2).

At QFIX, the destination queue is specified by the indicated transfer table. However, if the destination is a processor queue and the job's current HPQ indication (pointed to by HPQPNT) is nonzero, the job will be placed in that HPQ.

In transfer tables, the codes could be set as shown in Table 3-4.

Table 3-4
Example of Codes Set For Transfer Tables

Label	Content	Description
QNULW:	400000,,QFIX -1,,-NULQ	Transfer to back of NULQ. Do not assign quantum runtime.
QSTOP:		
QSTOPW:	400000,,QFIX -1,,-STOPQ	Transfer to back of STOPQ. Do not assign quantum runtime.
QJDCW:	400000,,QFIX -1,,-JDCQ	Transfer to back of JDCQ. Do not assign quantum runtime.
QCMW:	400000,,QFIX -1,,-CMQ	Transfer to back of CMQ. Do not assign quantum runtime.
QTSW:		
QRNW:	400000,,QFIX QADTAB,,-PQ1	Transfer to back of PQ1. Assign quantum runtime by QADTAB.
QRNW1:	400000,,QFIX -1,,-PQ1	Transfer to back of PQ1. Do not assign quantum runtime.
QTIOWW:	400000,,QFIX -1,,-TIOWQ	Transfer to back of TTY I/O wait. Do not assign quantum runtime.
QSLPW:	400000,,QFIX -1,,-SLPQ	Transfer to back of sleep queue. Do not assign quantum runtime.
QTIME:	400000,,QLNKZ 0,,QRQTBL	Transfer to queue specified by QRQTBL. Compute quantum runtime in QLNKZ.
QEWW:	400000,,QFIX -1,,-EWQ	Transfer to back of EWQ. Do not assign quantum runtime.
QRNW2:	400000,,QFIX -1,,-PQ2	Transfer to back of PQ2. Do not assign quantum runtime.

Table JBTCQ contains all the master queues. The table has one entry for each job and two entries for each master queue. Each master queue requires two entries because the master queues are divided between jobs with core and jobs with no core. These entries are referred to as queue headers. The queue headers are defined in the negative direction from JBTCQ.

If there are n master queues, the first n entries above JBTCQ in the negative direction are the in-core headers and the next n entries in the negative direction are the out-core headers. Each queue has an associated queue number. The location of the in-core header for a queue is JBTCQ minus the queue number. The location of the out-core header for a queue is JBTCQ minus the number of master queues minus the queue number. The entry for each job is located at JBTCQ plus the job number. The zero entry of JBTCQ, which would correspond to the null job, is not used.

Each entry in the table contains a pointer to the previous entry in the left half and a pointer to the next entry in the right half. Therefore, the queue headers contain a pointer to the last job in the left half and a pointer to the first job in the right half. The last job in the queue has a pointer back to the queue header (that is, a negative number) in the right half. Similarly, the left half of the first job in the queue points to the header. If a queue is empty, both pointers in the queue header point to itself.

For example, assume queue 2 contains jobs 1,4,2 in core and jobs 5,7,3 not in core. Queue 2 could be represented in JBTCQ as follows:

Location	Left half	Right half	
-MXQUE-2	3	5	queue header for section of queue with no core
-MXQUE-1			
-MXQUE			
⋮			
-3			
-2	2	1	queue header for section of queue with core
-1			
JBTCQ			
1	-2	4	entry for job 1
2	4	-2	entry for job 2
3	7	-MXQUE-2	entry for job 3
4	1	2	entry for job 4
5	-MXQUE-2	7	entry for job 5
6			
7	5	3	entry for job 7

In Class Scheduler mode, all jobs in PQ2 also have an entry in the table JBTCSQ. This table has headers that correspond to scheduler classes, also referred to as subqueues. Each subqueue has one header for jobs with core and one for jobs with no core.

The location of the in-core header for a subqueue is JBTCSQ minus one minus the class number. The location of the out-core header is JBTCSQ minus the number of classes minus one minus the class number.

Entries in JBTCSQ consist of a forward pointer and a backward pointer as in the master queues. Only the entries corresponding to headers or jobs in PQ2 have valid pointers.

For example, suppose class 0 contains jobs 1,4,2 in core and job 5 not in core, and class 1 contains no jobs in core and jobs 7,3 not in core. Subqueues 0 and 1 could be represented in JBTCSQ as follows:

-M.CLSN-2	3	7	queue header for class 1 with no core
-M.CLSN-1	5	5	queue header for class 0 with no core
-M.CLSN			
	:		
-3			
-2	-2	-2	queue header for class 1 with core
-1	2	1	queue header for class 0 with core
JBTCSQ			
1	-1	4	entry for job 1
2	4	-1	entry for job 2
3	7	-M.CLSN-2	entry for job 3
4	1	2	entry for job 4
5	-M.CLSN-1	-M.CLSN-1	entry for job 5
6			
7	-M.CLSN-2	3	entry for job 7

In addition, jobs in PQ2 that have core will also have an entry in either an input list (JBTJIL) or an output list (JBTOLS). The input list gives the order in which jobs with in-core protect time entered PQ2. The output list gives the order in which jobs were requeued to PQ2 after expiring in-core protect time. In Class Scheduler mode, each of these queues is subdivided into normal jobs and background batch jobs.

For example, suppose the just-swapped-in list contains jobs 1,4,2 in the regular chain and job 5 in the background batch chain, and the output list contains jobs 7,3 in the regular chain and no jobs in the background batch chain. The queues would be as follows:

-BBQ	5	5	queue header for background batch just-swapper-in list
-JIQ	2	1	queue header for regular just-swapped-in list

JBTJIL			
1	-JIQ	4	entry for job 1
2	4	-JIQ	entry for job 2
3			
4	1	2	entry for job 4
5	-BBQ	-BBQ	entry for job 5
-OBQ	-OBQ	-OBQ	queue header for background batch output list
-OLQ	3	7	queue header for regular output list
JBTOLS			
1			
2			
3	7	-OLQ	entry for job 3
4			
5			
6			
7	-OLQ	3	entry for job 7

3.17 SUBROUTINE DICLNK

The DICLNK subroutine moves a job from the in-core queues to the corresponding out-core queues. This subroutine is called when a job gives up core.

3.18 SUBROUTINE IICLNK

The IICLNK subroutine moves a job from the out-core queues to the corresponding in-core queues. This subroutine is called when core is assigned to a job.

3.19 SUBROUTINE DCCLNK

The DCCLNK subroutine is used by DICLNK and IICLNK to delete a job from its current master queue and from its subqueue in the class scheduler. The scheduler is locked while the linked lists are being updated.

3.20 SUBROUTINES ICCLNK AND ICSLNK

The ICCLNK and ICSLNK subroutines are used by DICLNK and IICLNK to insert a job into its proper master queue and subqueue.

3.21 SUBROUTINE INOLST

The INOLST subroutine inserts a job in the output list if it has core and is eligible to be swapped out. If the job is already in the output list, it leaves the job in its current position. Also, this subroutine inserts background batch jobs in the background batch chain and other jobs in the regular chain. INOLST is called when a job is requeued to PQ2.

3.22 SUBROUTINE DLOLST

The DLOLST subroutine deletes a job from the output list if it is in one. This subroutine is called when a job is requeued unless it is going from PQ1 to PQ2. It is also called when a job is swapped in or out.

3.23 SUBROUTINE DLJILS

The DLJILS subroutine deletes a job from the just-swapped-in list. This subroutine is called when a job is requeued from PQ2 and when it is swapped out.

3.24 QSCAN THROUGH FSQFOR SECTION

QSCAN scans the queues as specified by a scan table. It returns the job number of the next job in AC J. If the calling routine wishes to reject a job and continue the scan it must JRST (T2). The calling sequence is shown below.

```
     MOVEI     U, address of scan table

     JSP       T1, QSCAN

               Return here when no more jobs.
               Return here with next job.
```

The format of the scan table is shown below.

```
     SCANTAB:  XWD   Q1,CODE1
               .        .
               .        .
               .        .
               XWD   Qn,CODEn

               Z     (Zero terminates table)
```

In this case, CODE specifies the routine used for scanning. Table 3-5 lists the possible codes and their meanings.

Table 3-5
Possible Codes

Code	Meaning
QFOR	Scans whole queue forward. First scans the in-core chain, then the out-core chain.
QBAK	Scans whole queue backward. First scans the out-core chain, then the in-core chain.
IQFOR	Scans in-core queue forward.
IQBAK	Scans in-core queue backward.
IQFOR1	Scans in-core queue for first member.
IQBAK1	Scans in-core queue backward (all but first member).
OQFOR	Scans out-core queue forward.
OQBAK	Scans out-core queue backward.
OQFOR1	Scans out-core queue for first member.
OQBAK1	Scans out-core queue backward (all but first member).
SQFOR	Scans out-core subqueues (PQ2 class swap-in scan).
BQFOR	Scans out-core background batch subqueue (PQ2 class swap-in scan).
ISSFOR	Scans in-core subqueues (PQ2 class scheduling scan).
IBBFOR	Scans in-core background batch subqueue (PQ2 class scheduling scan).
OSSFOR	Scans out-core subqueues (PQ2 class lost-time scan).
IRRFOR	Scans just-swapped-in queue, then PQ2 in-core queue (PQ2 Round Robin scheduling scan).
IGFOR	Scans just-swapped-in queue and jobs waiting for a high segment as a result of a GETSEG UUO a certain percentage of the time (PQ2 swap-in scan).
OLFOR	Scans background batch output queue, then background batch just-swapped-in queue, then regular output queue, then PQ2 in-core queue backward (PQ2 output scan).

3.25 FSQFOR THROUGH BQFOR SECTION

The SQFOR routine is used by the class scheduler for the PQ2 swap-in scan. First, it scans the primary class. Second, it scans any classes with nonzero secondary allocations.

At FSQFOR, set the SWPFAR flag to indicate that the swapper reached fair territory.

If in Round Robin mode (RRFLAG = 0), go to OQFOR to scan the PQ2 out-core queue. Then, subtract one from SQCNT, and if it reaches zero, call SQINI to reinitialize the primary scan pointer. Finally, load the class number of the current primary subqueue into AC J.

At SQFORA, scan the out-core subqueue for that class.

From SQFOR1 to SQFOR2, build the secondary scan table SQSCAN. The primary class and any classes with no jobs in the out-core chain are rejected for efficiency. Any class with the fixed swap-in bit set (bit 0 of CLSSTS = 1) is also rejected, because this class is allowed to swap only when it is the primary class. All other classes with secondary allocations (CLSQTA>0) are stored in the SQSCAN table in the form XWD -CLASS-1,secondary allocation. The sum of the secondary allocations$ of all classes entered into the table is accumulated in SQSUM.

At SQFOR3, select a random integer in the range 0 to SQSUM-1. This integer determines which class will be selected next for the secondary scan. The secondary allocations of each entry in SQSCAN are successively subtracted from the random integer until it goes negative. The class that causes it to go negative is selected as the next class to scan. Therefore, the probability of any given class being selected is equal to its secondary allocation divided by SQSUM.

Eliminate the selected class from the SQSCAN table by moving the top entry down on top of it and subtracting its secondary allocation from SQSUM.

At SQFOR3, scan the out-core subqueue for the selected class. If no job is selected by the scan, decrement the count of classes left in SQSCAN. If any classes remain, go to SQFOR3 to select another class, otherwise go to the SQFOR routine.

3.26 BQFOR THROUGH ISSFOR SECTION

The BQFOR routine is used by the Class Scheduler to scan for background batch swap-in.

If in Round Robin mode (RRFLAG = 0), exit from the BQFOR routine. If no background batch class is defined (BBSUBQ<0) or not enough time has elapsed since the last background batch swap-in (UPTIME<SCNBBS), exit from the BQFOR routine. Otherwise, scan the out-core subqueue for the background batch class in BBFOR2. While scanning background batch, set BBFLAG to -1.

The SQINI routine is used to initialize the swapper's primary scan pointer. The counter SQCNT is initialized to 100 and indicates the number of entries left in this pass through the table. The byte pointer SQPNT is initialized to point to the imaginary byte preceding the first entry in PSQTAB.

The routine SQTEST is used to control the advancing of the primary scan pointer. The SQFOR routine advances the primary scan pointer to the current class. If no job is actually selected to be swapped in, the scan pointer is reset by SQTEST so that the same class is scanned for a time interval that approximates the average swap-in time (SCDSWP).

If in Round Robin mode, exit from the SQTEST routine. Then, add one to the count of how many ticks the current primary class has been scanned (SCNSWP). If this class has been scanned often enough

(SCNSWP>SCDSWP), clear SCNSWP and allow the primary scan pointer to advance to the next class. Otherwise, reset the primary scan pointer so that the same class will be scanned on the next tick. Then, add one to the count of primary classes left (SQCNT), and decrement the byte pointer (SQPNT) so it will point to the current class when incremented.

The routine RAND returns a random integer less than 2(17) in index T2. The algorithm is multiplicative modulo 2(35).

Multiply the current seed by 377775 octal. Store the low-order 35 bits as the new seed. Out of these, extract the leftmost 17 bits as the current random number.

3.27 ISSFOR THROUGH OSSFOR SECTION

The ISSFOR routine is used by the Class Scheduler mode for its PQ2 scheduling scan. The subqueues are scanned in the order specified by the subqueue scheduling scan table (SSSCAN for CPU0 and SSSCN1 for CPU1).

The scan table is built at the beginning of each microscheduling interval by the routine SCDQTA. Each entry in the table is of the form -CLASS-1. The first entry in the table is the primary class. The percentage of time that each class is selected as the primary class is determined by the primary percentage for that class. The remaining entries in the scan table are the secondary classes. SCDQTA uses probability to determine the order of the secondary classes, and uses the secondary allocation of each class to determine its relative priority.

To ensure a minimum level of response and to prevent core from becoming clogged with jobs that come from classes with low primary percentages, a portion of each microscheduling interval is dedicated to running jobs in the order in which they were swapped in. The response fairness factor (SCDJIL) controls the percentage of time that this special scan is in effect.

The code for ISSFOR is as follows.

If in Round Robin mode (RRFLAG = 0), go to IRRFOR to scan PQ2 forward. If response fairness is in effect (UPTIME<SCNJIL), scan the just-swapped-in queue at SJFORA. If response fairness is not in effect or no runnable job is found in the just-swapped-in queue, scan the subqueues in the order specified by the subqueue scheduling scan table. Then, set AC M to the base address of the table. At SSFOR1, if the word pointed to by M is zero, go to SSFOR2 because the end of the table has been reached. Otherwise, scan the in-core subqueue for the class pointed to by M. If no runnable job is found, add one to M and go to SSFOR1 to scan the next class in the scan table.

At SSFOR2, the primary and all secondary subqueues have been scanned. If the just-swapped-in queue was not scanned previously (UPTIME≥SCNJIL), go to ILFOR1 and scan it now.

The only jobs that would be scanned by this final scan of JBTJIQ are jobs that are in a class with no secondary allocation that have not yet expired 1 time slice. Because these jobs will be scanned at the beginning of the next microscheduling interval (when UPTIME<SCNJIL), they can be scheduled now because no other PQ2 timesharing jobs are runnable.

The routine IBBFOR is used by the Class Scheduler mode to schedule background batch jobs.

If in Round Robin mode (RRFLAG = 0), exit from IBBFOR. To indicate that the background batch is being scanned, set BBFLAG to -1. Then, scan the background batch just-swapped-in queue (JBTBBQ) at BBFORA. This action schedules any background batch job that has not yet expired one time slice ahead of those that have expired their time slices. Finally, scan the in-core subqueue for the background batch class at BBFORB. Zero BBFLAG and exit IBBFOR if no runnable job has been found.

3.28 OSSFOR THROUGH ILFOR SECTION

The routine OSSFOR is used for the Class-Scheduler-mode PQ2 lost-time scan.

Set M to the base address of the scan table. For each entry in the table, scan the out-core subqueue for that class at SSFORB. When all classes in the scan table have been processed, scan the out-core subqueue for the background batch class at OBBFOR.

The Round Robin scheduler uses the routine IRRFOR for its PQ2 scheduling scan. For best response, all PQ2 jobs that have not yet expired 1 time slice are scheduled ahead of those that have expired at least 1 time slice.

Scan the just-swapped-in queue (JBTJIQ) at RJFORA. Then, go to IQFOR to scan the full PQ2 in-core queue.

3.29 ILFOR THROUGH SAVSUM SECTION

The swapper uses routine ILFOR to scan for PQ2 jobs that have done GETSEGs and need to be linked up to their high segments. The in-core fairness factor (SCDIOF) controls how often these jobs are scanned ahead of regular PQ2 jobs.

ILFOR generates a random number in the range 0 to 99. If this number is greater than or equal to SCDIOF, exit the ILFOR routine. Otherwise, scan the just-swapped-in queue at ILFORA (ignore jobs with JS.HNG = 1).

The swapper uses routine OLFOR to scan PQ2 for output.

The swapper also saves SUMCOR in the temporary variable SAVSUM and performs the following tasks:

1. Scans the background batch output queue at OLFORA.
2. Scans the background batch just-swapped-in queue at OLFORB.
3. Scans the regular output queue at OLFORC.
4. Resets SUMCOR to SAVSUM
5. Scans the PQ2 in-core queue backward at IQBAK.

It is necessary to reset SUMCOR because some jobs will be scanned twice, once by OLFOR and once by IQBAK.

Note that all background batch jobs are swapped out ahead of any PQ2 timesharing jobs. If the system administrator wishes to give background batch jobs that have not expired 1 time slice a higher priority for remaining in core than timesharing jobs that have expired their time slices, the OLFORB code should be moved below the OLFORC code.

CHAPTER 4

DETAILED DESCRIPTION OF SWAPPER

The swapper code is dependent on a number of assembly switches. This discussion assumes a KI10 processor (FTKI10 = -1), with the virtual-memory option (FTVM = -1) and the high-availability option (FTDHIA = -1), which does not swap PDBs (FTPDBS = 0).

The swapper is entered at the label SWAP. It determines whether or not any jobs require swap-in or swap-out, and if so sets up the required swap control information in tables for VMSER and SWPSER. The actual swap (and any virtual-memory paging) is performed at interrupt level in VMSER and SWPSER.

Most operations started by the swapper require several clock ticks to run to completion. A number of flags are used to remember previous starts. Table 4-1 is a list of the important flags and data items used by the swapper.

Table 4-1
Important Flags and Data Items

Code	Meaning
FIT	Job chosen by QSCAN to be swapped in.
FORCE	Job chosen to be swapped out.
FORCEF	Job being forced out but waiting to give up disk resource.
SWPIN	Job number associated with high segment being swapped in.
SWPOUT	Job number associated with high segment swapped out last.
LASIN	Last segment swapped in.
LASOUT	Last segment swapped out.
SWPERC	LH=number swap errors. RH=number pages lost -- bits 18-23=err flags.
MAXJBN	Job number of job to swap out.
SUMCOR	Total amount core found so far of eligible jobs to swap out.

Table 4-1 (Cont.)
Important Flags and Data Items

Code	Meaning
INFLG	NOFIT flag -- same job waiting to be swapped in for 360 ticks.
INFLGJ	Frustrated job waiting to be swapped in.
INFLGC	Time frustrated job started waiting.
SPRCNT	Number swap operations in progress.
SWPCNT	Number jobs finished with data transmission, waiting for cleanup.

The overall operation of the swapper is described in the following.

The next job to be swapped in is selected by the input scan and stored in the item FIT. If the job will fit in available free core (unused), it is immediately swapped in. If the job will not fit, but would fit if the idle and dormant high segments were deleted from core, enough are deleted until the job will fit. The job is then swapped in.

If the job would not fit even if all idle and dormant high segments were deleted, the swapper checks to see if the job would fit if all jobs eligible to swap were swapped out. If no, the swapper exits and checks again each clock tick. If yes, jobs are sequentially selected by the output scan and put in the item FORCE to be forced out.

When all I/O has stopped and all sharable disk resources have been given up, the job is swapped out. While waiting for I/O to stop or resources to be given up, the swapper exits and rechecks on the occurrence of each clock tick.

After each job is swapped out, control returns to checking whether the job will fit after all idle and dormant segments have been deleted. When the job will fit, the high segments are deleted and the job is swapped in.

The following sections provide a description of the swapper at the level of the macro code. The labels referenced are in the last half of the module SCHED1.

4.1 SWAP TO SWAP1 SECTION

This section provides improved swapping response for HPQ jobs.

If FIT is zero, go to SWAP0A (equivalent to SWAP1); otherwise, check (SKIPG J,.CPRTF(P4)) to see if an HPQ job wants to be swapped in. If no, go to SWAP0A. If yes, and this job is of higher priority than the job in FIT, reset the job currently in FIT unless the job being fit has a high segment that is already in core, in which case, go to SWAP0A.

4.2 SWAP1 TO FININO SECTION

This section determines if swapping input or output has just finished, and if so, branches to the appropriate wrap-up routine.

At SWAP1, if no swapping requests have just finished (SWPCNT = 0), go to SWP2 and bypass swapping wrap-up. Otherwise, at FININN test whether the swap just completed was swap-in (IO = 0) or swap-out (IO = 1). For swap-out, go to FINOUT. For swap-in, check for swap read error (sign bit of S = 1) and if yes, go to INERR. Otherwise, go to FININO.

4.3 FININO TO INERR SECTION

This section does the wrap-up after a segment has been swapped in.

At FININO, if the segment just swapped in is a high segment (job number greater than JOBMAX), go to FININH. Otherwise, for low segment return swapper space and delete SWPLST entry (at GIVBAK), then go to FININ5 (which jumps to FININ1).

At FININH, get the job number of low segment associated with this high segment (from SWPIN). If this job is migrating to a new device (SWPIN = MIGRAT), clear the high-segment swapping space (at ZERSWP) so that it will be swapped out to a different unit and fall through at FININ1.

At FININ1, using FININ subroutine at SEGCON:

1. If a low segment was just swapped in and the associated high segment is in transit, check to see if there is another swap list entry completed (subroutine NXTSLE). If yes, go to FININN to process it. If no, exit from the swapper (POPJ).

2. If a low segment was just swapped in and it has an associated high segment that is not in core, set AC J to the associated high-segment number, and go to FININ2 to initiate swap-in.

3. If there is no high segment, or it is already swapped in, or we just swapped it in, then go to FININ3 to wrap-up job swap-in with J = low-segment job number. (Note that in virtual-memory systems nonsharable high segments are treated as part of the low segment.)

At FININ2, go to FIT1 to initiate swap-in for high segment (job slot indicated by AC J).

At FININ3, both segments are now in memory. Do a wrap-up for the job just swapped in. At this point, J = low-segment number, regardless of the order in which the jobs segments are actually swapped in.

Use IMGIN and IMGOUT to determine if the job size has decreased. If so, add the amount of decrease to the counter for the amount of virtual memory available (VIRTAL).

Call subroutine UNSWAP for housekeeping on various swapper flags, to give back disk space, and to mark the job as swapped in (SWP = 0, SHF = 0).

Clear the job scanned by the scheduler (JS.SCN), the job could not be forced out flag (JS.HNG), and the job forced out by timer (JS.TFO) for the just-swapped-in job.

If the just-swapped-in job is migrating (MIGRAT = JOB #) or is not in a processor queue or command wait queue, flag the job as eligible to swap-out (PDMSWP = 1) and go to FININ7. This procedure is needed for jobs that are requeued out of the queue they were in when they were selected for swap-in (usually by command decoder). Otherwise, the job can be marked as not eligible for swap-out (PDMSWP = 0) and can be in a queue that is not decremented for in-core protect time.

If the swap-in was caused by a GETSEG (JS.NNQ = 1), go to FININ7. This avoids reassigning time slices, which would allow jobs doing GETSEGs to take over the system.

Assign in-core protect time (subroutine ASICPT) for just-swapped-in job (bits 1 to 17 in .PDIPT). Then, mark the job not to be swapped (PDMSWP = 0). If the job is in a processor queue, assign a quantum runtime depending on which queue the job is in. If the job is on the swap-out list, delete it from the list (subroutine DLOLST).

If the just-swapped-in job was not background batch, go to FININ6. Otherwise, set the background batch bit (JS.BBJ) to 1, put the job in the just-swapped-in background batch queue, and go to FININ7.

At FININ6, clear the background batch bit (JS.BBJ = 0). If the job is in PQ2, put it in the just-swapped-in timesharing queue.

At FININ7, clear the no-new-quanta bit (JS.NNQ = 0). Clear the background batch fit flag (BBFIT). Clear the frustration indicators INFLG, INFLGJ, and INFLGC. Clear the flag that FIT was zeroed by an HPQ job (.PDHZF), and go to SWP1.

4.4 INERR TO FINOUT SECTION

This section processes input swap read errors.

At INERR, if the segment is a high segment, go to INERR2. Otherwise, call SWPREC to record errors, clear JACCT, call ZAPUSR to clear all DDBs and I/O channels, and call CLRJOB to clear the protected part of the job data area. Then, fall through to INERR2.

At INERR2, save the segment number (PUSH P,J), and call SEGERR.

For low segments, SEGERR returns immediately (POPJ P,). For high segments, SEGERR sets the high-segment error flag (SERR) to 1, clears JBTNAM, and returns the virtual swapping space for the high segment if this is the first time the high segment had the error (SERR was equal to 0). SEGERR returns with J = associated low segment.

Restore segment number (POP P,J). If the segment was not user page map, go to FININ0. Otherwise, call GVPAGS and give back core for the page map and segment, delete the SWPLST entry, clear JBTADR, JBTUPM, and JBTSWP. If the job has a nonsharable high segment, clear JBTSGN. Call KILHGH to remove the high segment from the address space, and go to UNSWAP.

4.5 FINOUT TO SWP1 SECTION

This section does the wrap-up after a segment has been swapped out.

If any errors have occurred (RH of AC S ≠ 0), go to OUTERR. Otherwise, delete the SWPLST entry (at DLTSLE).

If no jobs are migrating or the segment just swapped was a high segment, go to FINOU2. Otherwise, check to see if the job has completely migrated (at PGOFF), and if so, mark the job as completed (JS.MIG = 1).

At FINOU2, set R to the base address for segment and call KCORE1 to return core.

At FINOU0, call FINOT and:

Return + 1	with J set to low-segment number if the segment just swapped was a high segment and there is a low segment yet to swap. Go to FORCEL to swap out low segment.
Return + 2	if just swapped a low segment and swapping is all finished for this user. Fall through to SWP1.

4.6 SWP1 TO FIT1 SECTION

This section contains much of the overall control code of the swapper, plus the code for the swapping input scan. (See Table 4-2.)

At SWP1 clear the FINISH flag (largely meaningless for virtual-memory systems). At SWP2, if there is a job to be forced out (FORCE = job number ≠ 0), go to FORCE1 and try to swap it out. Otherwise, go to FIT0.

At FIT0, if there are swaps in progress (SPRCNT≠0), go to CHKXPN. Otherwise, if a swap has completed (SWPCNT≠0), go to SWAP1. If there is a job waiting to swap in (FIT = job number ≠ 0), go to FIT1. Otherwise, input scan by performing the following tasks.

1. Zero BBFLAG to indicate that background batch is not currently being scanned.
2. Zero SWPFAR to indicate that the swap-in scan did not yet reach fair territory (PQ2).
3. Set U to the proper scan table depending on the swapper fairness count.

Table 4-2
Primary and Secondary Scan Swapping Tables

Primary Scan		Secondary Scan	
ISCAN		ISCAN1	
Queue	Routine	Queue	Routine
HPQs	QFOR	HPQs	QFOR
CMQ	QFOR	CMQ	QFOR
PQ1	IQFOR	PQ2	OQFOR (Round Robin mode)
PQ2	ILFOR	PQ2	SQFOR (Class Scheduler mode)
PQ1	OQFOR	PQ1	OQFOR
PQ2	OQFOR (Round Robin mode)	PQ2	ILFOR
PQ2	SQFOR (Class Scheduler mode)	PQ1	IQFOR
PQ2	IQFOR	PQ2	IQFOR
PQ2	BQFOR (Class Scheduler mode)	PQ2	BQFOR (Class Scheduler mode)

If the number of jobs in PQ1 swapped in a row is less than the maximum allowed (SWPIFC less than MAXIFC), use the primary scan. Otherwise, use the secondary scan.

At FITPRM, call QSCAN to do an input scan. At Return + 1, if all queues have been searched, go to ZCKXPN (no jobs waiting to swap in were found). At Return + 2 and following, process jobs returned from scan (code below).

If the job is expanding (JXPN = 1 or JS.XPN = 1), or if all of the job's segments are in core (SWP = 0) for low segment, or if the high segment is expanding (determined by CKXPN), reject the job and request the next job from QSCAN (JRST (T2)).

Otherwise, select job as next job to swap in (set FIT = job number).

If the job in FIT was selected in the background batch scan, remember the job number in BBFIT.

Otherwise, clear the SCNSWP counter (used to control the advance of the swapper primary scan pointer when there are no jobs to be swapped in).

If the swap-in scan reached fair territory (that is, the job selected was in PQ2), clear the counter for the number of unfair scans (SWPIFC). Otherwise, add one to the number of unfair scans (SWPIFC), and go to FIT1A.

Only one job at a time is entered into the swapping tables for the interrupt level routines to process. This is because jobs become unrunnable (SWP = 1) as soon as they are entered in the tables and this would result in too much core memory being tied up with jobs that are not runnable. The only exception is jobs that are expanding, which may be stacked into the swapping tables because they are not runnable in any case.

4.7 FIT1 TO OUTERR SECTION

This section controls the swap-in process so that it proceeds in an optimum manner.

If the job in FIT will fit in available free core, swap it in (at SWAP1).

Otherwise, if the job will fit when a number of idle or dormant high segments have been deleted, then delete that number of high segments and swap the job in at SWAP1.

Otherwise, go to SCNOUT to see if enough space is available to swap the job in. If there is not enough space, SCNOUT will exit and FIT1 will be reentered next tick for a reevaluation. If enough space exists, SCNOUT will put the next job to swap out into FORCE to force it out.

In detail, the logic of this section is as follows:

At FIT1, save the job number in FIT because it may just have been selected for swap-in (if entered from FININ2).

Check (at CKXPN) to see if this is a low segment for which the associated high segment is expanding (perhaps by some other job that is sharing it). If yes, go to NOFITZ to deselect this job (set FIT to 0) and exit the swapper. (The input scan will not select such a job. The output scan will swap out and expand the high segment. Only then may any of the jobs swap in again.) If no, fall through to FIT1A.

At FIT1A, put the size of the low segment plus the size of the page map (UPMPSZ) in P1 in preparation for calling FITSIZ. In special cases where low segment is already in core (JBTADR(J)=0), set P1 to zero.

Call the routine FITSIZ is to determine if the job will fit in free core plus the space occupied by idle and dormant segments. (Idle segments are high segments that are linked to low segments on the swapper but not to any low segments in core. Dormant high segments are not connected to any low segments, either on the swapper or in core.) FITSIZ determines if the job will fit by testing for the existence of a high segment, and if required, by adding its size to the total job size in P1. The total job size (P1) is then compared with free + dormant + idle space (CORTAL). Returns from FITSIZ are as follows:

Return + 1	Job will not fit, go to SCNOUT (to try to swap jobs out).
Return + 2	Job will fit, go to FIT1B (swap into free core, or delete high segments as required; then swap in).

At FIT1B, the job being swapped is known to fit in free + dormant + idle core. If the job will fit in free core (job size less than or equal to BIGHOL), go to SWAP1 and swap the job in. (On KA10s, BIGHOL is the largest contiguous block of available memory, not all of free core. To get a fit, KA10s may need to shuffle to increase BIGHOL.) If the job will not fit, use subroutine FRECR1 to delete idle or dormant segments. (For KA10s, this implies a preference to delete idle or dormant segments before shuffling.)

FRECR1 searches for dormant and then idle high segments to delete. It will not delete a high segment that is needed by the job being swapped in. Returns are as follows:

Return + 1 High segment selected has no copy on swapper, so it must be swapped out. Go to FORIDL (high segment was just associated by get segment, or for non-virtual-memory systems high segment was nonsharable).

Return + 2 One idle or dormant segment has been deleted, go to FIT1B and see if job now fits.

Return + 3 All idle and dormant segments have been deleted. Job still did not fit; execute code starting at SHFPAT.

At SHFPAT, check to see if there are any holes in memory that the shuffler could eliminate (HOLEF ≠ 0). If no (always no for KI10 and KL10), go to SCNOUT to try to swap some jobs out. If yes, call the shuffler and successively move jobs (only for KA10s).

4.3 OUTERR TO SWPREC SECTION

This section processes swap-out errors.

At OUTERR, if the error was caused by the disk system, go to OUTER1. Otherwise, fall through to the code described below.

For memory parity errors (read from memory by the swapping channel) record the error flags and the number of swap errors (SWPRC1). Then, call HGHSWE to take action if the segment being swapped out was a high segment.

Returns from HGHSWE are as follows:

Return + 1 Job was a high segment, the swap-out error message has been sent to all low segments attached to this high segment. Name of high segment is cleared so no others can connect to it (CLRNAM). Go to OUTER0.

Return + 2 Job was a low segment. Fall through to code described below.

For low segments, if the error was in a protected part of the job data area, return the swap space (CHGSWP), and clear all user DDBs and I/O channels (ZAPUSR).

Print error message and stop job (SWOMES), and clear the swap error (SL.ERR and SL.CHN) in swap tables (SWPLST (P1)).

Reenter the swapper to start a new operation (at SWAP1).

At OUTER1, the swap-out error is known to be a device error. Call SWPREC to record the errors, reset the map at MAPBAK, and try to swap out again in a different place (at SWAP0). The old copy is left on disk so that the bad area will not be used again.

4.9 SWPREC SUBROUTINE

This subroutine records the amount of swapping space lost because of swap errors (in and out). It falls through to the SWPRC1 subroutine that counts the number of swap errors.

Add amount of virtual core lost to error-count register (RH SWPERC).

If the segment lost is not a high segment, add the page map size (UPMPSZ) to the amount of space lost (T1), and decrease the amount of virtual core (VIRTAL) by amount lost (T1). Fall through to SWPRC1.

4.10 SWPRC1 SUBROUTINE

This subroutine stores the error flags in SWPERC from bits 18 through 23 in S and adds 1 to the number of swap errors (LH SWPERC).

4.11 ZCKZPN TO SCNJOB SECTION

This section is entered at ZCKXPN if the swap input scan found no jobs to swap in. It checks for and forces out expanding jobs. It is also entered at SCNOUT if jobs must be swapped out to make room for the next job coming in.

At ZCKXPN, clear the swapper fairness count. In the Class Scheduler mode, call SQTEST to maintain the primary scan pointer. At CHKXPN, if there are no expanding jobs (XJOB = 0), go to CHKMIG (check for migrating jobs, then exit swapper). Otherwise, clear the control flag (SCNJBS), and go to SCHOU0.

At SCNOUT set the control flag (SCNJBS) to -1. If a job has already been selected for swap-out (FORCE ≠ 0), go to SCNOU1.

If there are no jobs waiting to expand (XJOB = 0), go to SCNJOB. Otherwise, fall through to code below. (Note, entry from ZCKXPN implies existence of expanding job to be swapped. Such entry never causes nonexpanding jobs to be swapped.)

Loop through bit map (XPNMAP) looking for expanding jobs. If an expanding job is found with JS.HNG = 0 or no longer has active devices (ANYDEV), exit to SCNOK with J = job number. If no expanding jobs are found, execute STOPCD XTH, because there should have been at least one expanding job (because XJOB ≠ 0). If all expanding jobs have JS.HNG set, go to CHKMIG if SCNJBS = 0, or to SCNJOB if SCNJBS = 0.

At SCNOK, if the expanding job has core assigned (JBTADR (J) ≠ 0), go to FORCE0 to force the job out. Otherwise, decrement the count of expanding jobs (XJOB), clear the expand bit for the job (JXPN), and go to FORCE0 to try to swap the job out. (The purpose of this code is to keep the expand bit set until the expanding job has been placed in the interrupt level swap tables, when the swap bit is set.)

At CHKMIG, if there are swaps in progress (SPRCNT ≠ 0) or a swap was just completed (SWPCNT ≠ 0), go to FLGNUL and exit the swapper.

If there are no migrating jobs (MIGRAT = 0), go to FLGNUL and exit the swapper.

If we have checked all jobs for migration (J greater HIGHJB) at CHKMI1, go to MIGDON. Otherwise, for all jobs that are not swapped

(SWP = 0), check to see if the job has any pages on the unit going down (subroutine PGOFF), and if so, put the job number in MIGRAT and go to FORCE0 to try to swap the job out. If the job is swapped and has not already migrated (JS.MIG = 0) and is not currently being swapped, set MIGRAT = J, and go to FIT1.

Clear the migration flag (set MIGRAT to 0) at MIGDON, and exit the swapper.

If SCNOU1 is reached, FORCE was already set on a previous clock tick. If the job being forced had a sharable resource assigned the last time the swapper tried to swap it out (FORCEF = J = job we want to swap out), go to FORCE1 to see if job has given up all resources. Otherwise, go to FORIDL and make sure swap indicator is set for job (SWP = 1).

The reason for not setting SWP for a job with a sharable resource is that the job must be run until it gives up all resources before it can be swapped out and the SWP bit prevents jobs from being selected to run by the scheduling scan.

4.12 SCNJOB TO FORCE0 SECTION

This section is entered if the job being swapping in will not fit in free and dormant and idle core, and all expanding jobs have already been swapped out. Some jobs that are not expanding must be swapped out to create more space. Swap-out begins when a sufficient number of jobs are eligible to be swapped (PDMSWP = 1), so that enough space will be available for the job coming in. No jobs are swapped before this time, so that runnable jobs will be kept in core as long as possible.

The routine is entered with AC P1 set to the amount of space (in pages) needed to swap the next job in (so that both segments are in).

At SCNJOB, set SUMCOR to amount of free and idle and dormant space (in pages). Clear indicator for first job found to swap out (MAXJBN).

Set J to segment number being swapped (from FIT) and call FITHPQ to set J = associated low segment if the segment being swapped was a high segment.

Save low-segment number in FITLOW.

Set AC T4 to the job's high-priority queue number. (If not zero, this will be used later to give preference to HPQ jobs that you want to swap in.) If the job in FIT was forced out by the timer (JS.TFO = 1) set T4 to 0 (this prevents swapper thrashing when an HPQ job and another job both want to run and will not fit in core simultaneously).

Set SCNSTP so output scan will stop at queue of job being swapped in (when not forced by timer).

Set AC U to output scan table (OSCAN) and call the QSCAN subroutine to select the next job to swap out. The order specified by OSCAN is shown in the following:

DETAILED DESCRIPTION OF SWAPPER

OSCAN

QUEUE	ROUTINE
STOPQ	IQFOR
SLPQ	IQFOR
EWQ	IQFOR
JDCQ	IQBAK1
TIOWQ	IQFOR
JDCQ	IQFOR1
PQ2	OLFOR
PQ1	IQBAK
CMQ	IQBAK
HPQs	IQBAK

The returns from QSCAN are:

Return + 1 — All queues have been scanned, job still will not fit. Go to NOFIT to service timer and exit from the swapper.

Return + 2 — Returns next job in AC J, to be processed by code described below.

If the job being scanned for swap-out is from a queue beyond the end of the scan limit (U greater than SCNSTP), the routine has scanned all jobs of lower priority than the job trying to swap in. If the timer has expired (6 seconds have elapsed since the low segment in FITLOW was selected), then allow output scan to search all queues to completion (JRST .+2) so that the job being swapped in can replace jobs of higher priority if it has been waiting too long. Otherwise, go to NOFIT to service timer and exit from the swapper.

Reject the job being scanned for output (JRST (T2)) if it is the job being swapped in (that is, the low segment associated with a high segment being swapped in), or if the job does not have core assigned (JBTADR (J)) = 0).

Set AC W to the address of PDB. If there is no PDB, go to SCNJB1 and ignore the in-core protect time check. Also, go to SCNJB1 if the job has the swap bit (SWP) set to 1, or if the job going out is in background batch and the job coming in is not.

At SCNJB0, if the job's in-core protect time has expired (PDMSWP = 1) and the job may be swapped (NSWP = 0), fall through to code described in the next paragraph. Otherwise, if the in-core protect time has not expired, reject the job if the job coming in is not HPQ (JUMPE T4, (T2)). If the job coming in is in HPQ, test to see if the job can be swapped (NSWP = 0). Reject the job if it cannot be swapped (JRST (T2)).

Reject the job if it is in a processor queue (other than background batch) and the job coming in is background batch.

At SCNJB2, if the job has JS.ENG equal to 0, go to SCNJB3. Otherwise, call ANYDEV to see if the job still has active I/O. If yes, reject job for swap-out. If no, fall through to SCNJB3.

At SCNJB3, execute special code to prevent system hang in rare circumstances.

Reject the selected job (JUMPN F, (T2)) if it is in the process of swap-in and status indicators have not yet been properly set up (avoids instant swap-out).

Set F to the size of job (IMGIN) plus the size of the page map (UPMPSZ). Call subroutine FORSIZ in SEGCON to estimate the high-segment size.

If the selected job has a high segment, and it is in core, the subroutine FORSIZ adds to AC F an estimate of the high-segment size according to the following formula:

Estimated Size = (High-Segment Size/In-Core Count)+1

If a job has been selected for swap-out, go to FORCE2. Otherwise, if a job is running on the slave, set SW0JOB = job number and reject the job (the slave will stop running the job at next opportunity). If a job has a real high segment with a SAVE in progress (ANYSAV), reject the job for swap-out. Otherwise, set MAXJBN to job number of the first job found in the scan that is eligible for swap-out.

At FORCE2, add the size of this job to the total found so far (SUMCOR). If the job being swapped in still will not fit (P1>SUMCOR), go back and see if there are more jobs eligible to swap out (JRST (T2)). Otherwise, fall through to the code described below.

In the Class Scheduler mode, if a background batch job is being fit (BBFIT ≠ 0), clear SCNSWP and allow the primary scan pointer to advance on the next swap-in scan. Also, calculate the time at which the next background batch job is allowed to swap in (UPTIME + SCDBBS).

Set J to MAXJBN, the first job found in output scan, and therefore the lowest priority job. If the timer has expired (INFLG ≠ 0) and the job is in a processor queue, then set the job forced out by timer flag (JS.TFO).

Fall through to FORC00 with J = job to be forced out.

4.13 FORCE0 TO SWAP0 SECTION

This section determines whether a job can be swapped out, or if it must wait for I/O to finish or for sharable resources to be given up.

At FORC00, if the job selected for swap out (J = job number) is not runnable with respect to CACHE, exit to FLGNUL. If it has a SAVE in progress, exit the swapper (JRST (T2)) without setting FORCE so that another job will be selected next tick. This is meaningful if it is entered from SCNOUT for expanding jobs. This has already been checked if it was entered from SCNJOB above.

If the segment being swapped is a high segment (J>JOBMAX), go to FORCEA.

If the segment being swapped is a low segment, and the job was hung with I/O active (JS.HNG), go to SWAPO. (The remaining checks were already made before the hung indicator was set). If the job is not marked, hung check for disk-sharable resources at FLSDR. From FLSDR, returns are:

Return + 1	No disk-sharable resources. Go to FORCEA.
Return + 2	Job currently assigned one or more disk-sharable resources. Execute code described below.

Save the job number of the segment being swapped (set FORCE equal to J). Also, store an indicator that job is being forced with sharable resources (FORCEF = job number). Go to FLGNUL to exit the swapper without swapping a job this tick. The scheduler will then run the job at highest priority until it gives up all sharable disk resources so it can be swapped.

At FORCEA, check (FORHGH) to see if there is a high segment that can be swapped before this segment. True if this is a low segment that has a high segment in core (SWP = 0) that is not expanding, has a core count of one (this low segment), and is not associated with the job being swapped in. If yes, return the high-segment number in J. If no, return low-segment number in J, and set the shuffle bit (SHF) to 1 so that I/O will stop after the buffer full.

At FORIDL, set swap bit (SWP) to 1.

At FORCEL, save the job number of the segment to be swapped out into the force-out indicator (FORCE).

At FORCE1 (entered from SWP2 at the clock tick), see if the job can now be swapped, as well as from above. If not forcing job with a disk-sharable resource (FORCEF = 0), go to FORCEB. Otherwise, check to see if the job still has resources (FLSDR). If yes, exit the swapper at FLGNUL and check again next tick. If no, clear FORCEF and go back to FORCEA to complete steps that were delayed while waiting for job to give up resources.

At FORCEB, if the job has no core assigned (JBTADR (J) = 0), go to SWAPO and swap out the next job (it cannot have any active devices). Otherwise, check to see if the job is the current job (J = .C0JOB). If yes, exit the swapper at NOFORC and wait for scheduler to context switch out of the job. If no, check for active devices or for the current job on the slave processor (ANYDEV). If yes, exit the swapper at NOFORC. If no, fall through to SWAPO and swap out the job.

4.14 SWAPO TO NOFIT SECTION

This section puts the swap-out information in the SWPLST tables and calls the interrupt-level routines.

At SWAPO, clear the output timer. If the segment being swapped is a low segment, delete the job from the output list and the just-swapped-in list. Clear JS.XPN, JS.HNG, and JS.NNQ. Save the job number of the last job swapped out (set LASOUT equal to J), and clear the force flag (set FORCE equal to 0). Clear the SW0JOB flag. If the job has zero core (JBTADR(J) = 0), go to SWP1 to start a new operation, because there is no need to swap out the job. Otherwise, continue below.

Set AC U to job input size (IMGIN). If the segment is a low segment (J less than or equal to JOBMAX), set the shuffle bit (SHF) to 1 to indicate that a swap-out is in progress.

Set the segment output size (IMGOUT) from the input size (IMGIN, as stored in U above), unless the job expanded (IMGOUT ≠ 0), then leave it as set by the expand routine and set U to new segment (IMGOUT).

If the segment is a low segment, add the user page map size (UPMPSZ) to AC U (to be used in call to SWPSPC).

Save IMGOUT (in AC F), set to zero for call to SWPSPC. Call SWPSPC. If there is no space, go to SWAP03 (to exit swapper and try again next tick). Otherwise, get device storage space and restore IMGOUT to the saved value (AC F).

Save J, build SWPLST entry (at BOSLST), add one to the number of swap operations in progress (SPRCNT), start I/O if it is not already going (at SQOUT), and restore J.

If the job that was just swapped was a low segment that expanded, decrement the count of expanding jobs (XJOB), clear the entry in the bit table (XPNCLR), clear the table expand bit (JXPN), and go to CHKXPN to swap out the expanding jobs, and when there are no more, exit from the swapper.

At SWAP03, set IMGOUT to 0 unless it is different from IMGIN, set FORCE to the job number, and go to FLGNUL.

4.15 NOFIT TO NOFIT3 SECTION

This section is entered every clock tick that the job in FIT cannot be swapped in, because there is not enough space even if all idle and dormant segments are deleted and all jobs that are eligible to be swapped out are swapped out. Recall that eligible to be swapped out implies that the jobs have expired their in-core protect time and are of lower priority in the swap-out scan than the job being swapped in.

A timer keeps track of how many ticks the job has waited to swap in. After 6 seconds, the timer expires and sets a flag to indicate that the swap-out scan routine (SCNJOB) may now ignore the queue position and swap jobs out with expired in-core protect, even if they are of higher priority.

This timer is needed only for very special cases. For example, if an HPQ job and a very large job both want to run and cannot fit in core simultaneously, then the large job will not displace the HPQ job until the timer expires, because the HPQ job is always higher in the queue. No known special cases exist for PQ1 and PQ2, because of the orderly operation of the Round Robin algorithm.

At NOFIT, if the job selected for swap-in was a background batch job, deselect it (set FIT and BBFIT to 0), reset the scan pointer as though no job were swapped in (at subroutine SQTEST), and go to FLGNUL.

At NOFIT1, if the job being swapped in was preempted by an HPQ job, restore the timer to the value it held when the job was preempted and go to NOFIT7.

At NOFIT3, if the frustrated job is the same as last time (FITLOW - INFLGJ), go to NOFIT7. Otherwise, start the frustration timer for this job and go to FLGNUL.

At NOFIT7, if the job being timed has been waiting 6 seconds, set the frustration flag (INFLG) to -1 and go to FLGNUL.

4.16 NOFITZ TO ZERFIT SECTION

This section clears the FIT and BBFIT indicator if a job were selected to FIT and then the high segment it was connected to was expanded by some other job that is sharing it. (See Section 4.7.)

4.17 ZERFIT TO NOFORC SECTION

This section clears the FIT and BBFIT indicator if an HPQ wants to swap in and certain conditions have been met. (See Section 4.1.) It also stores the frustration time for the job being preempted in the PDB for that job (.PDHZF).

4.18 NOFORC TO SWAP1 SECTION

This section is entered every clock tick that the job in FORCE cannot be swapped out because it has active I/O or is the current job on some CPU. A timer keeps track of how many ticks the job has been selected for swap-out. After 3 seconds, the timer expires. The job is deselected for swap-out, and is marked hung as far as swap-out is concerned.

At NOFORC, if the job being swapped out is a high segment, exit to FLGNUL. If this job is the same as the previous job being timed (J = OUFLGJ), go to NOFOR1. Otherwise, start the timer for this job and go to FLGNUL.

At NOFOR1, if the job being timed has been waiting for 3 seconds, set JS.HNG (so that the swapper will not select this job for swap-out again until I/O is no longer active). Clear FORCE, FORCEF, and OUFLGJ.

On non-virtual-memory systems, if the selected job was expanding (JS.XPN = 1), set JXPN to 1 and reenter the job in the table of expanding jobs (this is done because non-virtual-memory systems clear JXPN as soon as the job is selected for swap-out).

4.19 CHGSWP TO CHG1 SECTION

This section changes disk-swapping space allocations (VIRTAL).

At CHGSWP, save the present input size (IMGIN) in T2. If the new core assignment is zero (T1 = 0), go to CHG1; otherwise, continue below.

Convert the new core assignment pages, store them in IMGIN, and save AC J.

Compute the change to the system's virtual address space and update the indicator (VIRTAL).

Restore AC J and exit.

4.20 CHG1 TO UNSWAP SECTION

This section calls the subroutine (GIVBKH) to give back physical disk space.

At CHG1, if the segment has no space on disk (T2 = 0), go to ZERSWP. Otherwise, increment VIRTAL by T2 (plus size of page map if it is a low segment).

At ZERSWP, save AC U, and if the disk output size is 0 (IMGOUT = 0), go to CHG10. Otherwise, set up T1 and call ZERSWH.

From ZERSWH, the returns are:

Return + 1	Call GIVBKH, low segment or no error in high segment (gives back disk space).
Return + 2	Restore U, error in high segment or fall through from above. Fall through to UNSWAP.

4.21 UNSWAP TO RTNDSP SECTION

This section housekeeps job and swapper flags after a segment has been swapped in.

At UNSWAP, clear the swap and shuffle bits (SWP and SHF).

If the job just swapped was being forced (J = FORCE), clear FORCE and FORCEF.

At UNSWP1, set the disk output size to 0 (IMGOUT). For low segments, clear LH JBTSWP(J).

Exit (POPJ P,).

4.22 RTNDSP TO GIVBKH SECTION

This section returns disk space.

4.23 GIVBKH TO XPAND SECTION

This section clears the SWPLST entry and calls RTNDSP to return physical disk space.

4.24 XPAND TO XPANDH SECTION

This section gets more core for a job by swapping it out and then swapping it back in again.

4.25 XPANDH TO SCHED. SECTION

This section stops a job and swaps it out if it has just been connected to a sharable high segment that is on disk or is being swapped in or out. The job remains stopped until the high segment is in core.

CHAPTER 5

SCHEDULING PARAMETERS

The scheduler contains a variety of control parameters that may be set by an installation to suit its particular needs. The non-class scheduler provides a basic set of parameters. The class scheduler provides a number of additional parameters.

This chapter describes the location of the parameters and the default values assigned at start up. The default values may be modified by an installation as desired. Also, in the class scheduler, any parameter may be modified dynamically with a SCHED. monitor call (using the SCDSET program).

5.1 PROCESSOR QUEUE TIME SLICES

The processor queue time slices are made up of two parts: in-core protect time and quantum runtime.

One of the following formulas determines the in-core project time (in ticks) for all processor queues.

1. At swap in, in-core protect is

   min(PROTM,JOBSIZ*PROT+PROT0+8333)/16667

2. When requeued to back of PQ2 because of time-slice expiration, in-core protect time is

   PROT1

The indicated ONCMOD tables indexed by the primary swapping device determine the default values for the in-core protect-time parameters. However, the indicated SCHED. monitor call may dynamically modify them.

Scheduling Parameter	ONCMOD Table	SCHED. Monitor Call
PROT	PROTTB	PROT
PROT0	PRT0TB	PROT0
PROTM	PRTMTB	PROTM
PROT1	PRT0TB	PROT1

SCHEDULING PARAMETERS

To compute quantum runtimes, use the following formula:

quantum run = min {[QMX, QAD + (size of job in K)*QML]/QRANGE}

where QMX, QAD, and QML come from tables QMXTAB, QADTAB, and QMLTAB indexed by processor queue, with index 0 for PQ1, 1 for PQ2, and 2 and following for HPQs.

For PQ1, the default values are set in COMMON and modified by SCHED. monitor calls as indicated.

Scheduling Parameter	COMMON Parameter	SCHED. Monitor Call
QADTAB (0)	QQRUN1	TIME BASE
QMLTAB (0)	0	TIME MULTIPLIER
QMXTAB (0)	QQRUN1	TIME MAXIMUM

For PQ2, the default values are set from ONCMOD tables indexed by the primary swapping device and modified by SCHED. monitor calls as indicated.

Scheduling Parameter	ONCMOD Table	SCHED. Monitor Call
QADTAB (1)	ADDTAB	TIME BASE
QMLTAB (1)	MULTAB	TIME MULTIPLIER
QMXTAB (1)	MAXTAB	TIME MAXIMUM

For HPQs, the quantum runtimes are defined by macros at the location of QADTAB, QMLTAB, and QMXTAB in COMMON. The values generated depend on the number of HPQs. SCHED. monitor calls cannot change HPQ quantum runtimes. All processor queues use QRANGE. It is set to the default value of 45K directly in COMMON, and may be modified by a Time Multiplier subfunction of the SCHED. monitor call.

In-core protect and quantum runtimes have a different meaning for each of the processor queues.

5.1.1 PQ1 Time Slice

For PQ1 jobs, quantum runtime is a measure of the amount of time that the job receives exceptional (PQ1 level) attention for scheduling after it is swapped in. When this time expires, the job is requeued to the back of PQ2 (without being marked for swap-out) and is assigned the PQ2 quantum runtime. A PQ1 job is assigned the same in-core protect time as PQ2 jobs when it is swapped in. On requeue to PQ2, it retains any leftover in-core protect time.

This procedure gives fast scheduling response to PQ1 jobs that require very little CPU time, and reduces swapping for PQ1 jobs that continue to run after expiring the PQ1 quantum runtime. (Once a job is swapped, it is allowed to run at least as long as the PQ2 time slice, if it does not go into long-term wait.)

5.1.2 PQ2 Time Slice

For PQ2 jobs, the parameters for in-core protect and quantum runtime control the bias of the scheduler for throughput versus response and for I/O versus CPU.

Throughput versus response is controlled by increasing or decreasing the magnitude of both parameters. As the parameters are increased, jobs expire their time slices more slowly, swapping rate decreases, and throughput is improved (less core is tied up in swapping). Response is correspondingly degraded because jobs wait longer to swap in. When you decrease both parameters the effect is reversed.

I/O versus CPU response is controlled by changing the ratio of in-core protect to quantum runtime. Increasing only quantum runtime favors CPU jobs. Increasing only in-core protect favors I/O jobs, while reducing it tends to favor CPU-bound jobs.

5.1.3 HPQ Time Slice

For HPQ jobs, quantum runtime is set to a very small value so that if more than one HPQ job wants to run, the scheduler will context switch between jobs frequently.

The value of in-core protect time for HPQ jobs is the same as for PQ2 jobs. Normally, this is not significant because HPQ jobs can only be swapped out by other HPQ jobs. (It would be significant if an installation wanted to allow two HPQ jobs that did not fit to be in memory simultaneously.)

PQ1 and PQ2 jobs do not normally replace HPQ jobs, because the swapping output scan does not swap out a job of higher priority than the job coming in (even if the job in core has expired its in-core protect time). The only exception is if the 6-second fairness timer expires.

5.2 SWAPPING AND SCHEDULING FAIRNESS COUNTS

The PQ1 versus PQ2 swapping and scheduling fairness counts are defined in COMMON with the default values listed in Table 5-1. They may be modified with the indicated functions to the SCHED. monitor call.

Table 5-1
Default Values of Swapping and Scheduling Fairness Counts

Parameter	Description	Default	SCHED. Monitor Call
IFCO	Swapping Threshold	5	Swapper Fairness
SFCO	Scheduling Threshold (CPU0)	20	Scheduler Fairness
SFC1	Scheduling Threshold (CPU1)	20	Scheduler Fairness

Note that the SCHED. monitor call sets both CPUs to the same scheduling fairness threshold in a dual-processor system.

The scheduling and swapping fairness counts are a measure of the number of consecutive times the scheduling/swapping scan has selected a PQ1 job. After a specified threshold has been reached, a PQ2 job is selected, if available, by scanning with an alternate scan table that has PQ2 ahead of PQ1. Small threshold values favor PQ2. Large values favor PQ1.

5.3 IN-CORE FAIRNESS FACTOR

The in-core fairness factor, SCDIOF, is set to an initial value of 50% in SYSINI. It may be modified with the Incore Fairness subfunction of the SCHED. monitor call.

The in-core fairness factor determines the percentage of time that PQ2 jobs that have done a GETSEG and have not yet expired 1 time slice are scanned for swap-in ahead of regular PQ2 jobs.

This is the last of the scheduling parameters for the non-class scheduler. The following parameters apply to the class scheduler only.

5.4 CLASS QUOTAS AND MICROSCHEDULING INTERVAL

The class quotas are made up of the following three sets of parameters:

1. Primary percentages.

2. Secondary allocations.

3. Fixed swapping indicators.

The table CLSSTS stores the primary percentages as well as the fixed swapping indicators. The packed table PSQTAB, however, only stores the primary percentages. The initial values of both of these tables are zero at start up. The primary percentages and fixed swapping indicators are modified with the Primary Percentage subfunction of the SCHED. monitor call.

The table CLSQTA stores the secondary allocations. The initial value of this table is zero. The secondary allocations are modified with the Secondary Allocation subfunction of the SCHED. monitor call.

Item SCDINIT stores the microscheduling interval. The initial value is zero. It is modified by the Micro Scheduling Interval subfunction of the SCHED. monitor call.

The default values of zero for the above parameters cause the system to start up in Round Robin mode. To enter Class Scheduler mode, the parameters must be set with the SCDSET program.

The system enters Class Scheduler mode whenever the following conditions are met:

1. The primary percentages add to 100%.

2. The microscheduling interval is nonzero.

Conversely, the system enters Round Robin mode if either of the above conditions is not met.

The primary percentages define the amount of system resources granted to each class. The secondary allocations define the proportion of leftover resources allocated to each class. Leftover resources occur when some of the classes do not use all of their primary percentages.

If a class has a zero primary percentage, it is not guaranteed any portion of the machine. If it has a nonzero secondary allocation, it will get a share of leftover resources; if not, it will not be swapped or scheduled at all.

If a class has a nonzero primary percentage and a zero secondary allocation, it will be swapped and scheduled only a fixed amount of time. In other words, it will get exactly its primary percentage and no more.

The fixed swapping indicator causes a class to be swapped at a fixed rate, but scheduled as though it were nonfixed. This assumes that the class has a nonzero primary percentage and a nonzero secondary allocation. The class is swapped using only the primary percentage, ignoring the secondary allocation as though it were zero. Scheduling uses both the primary percentage and the secondary allocation.

This feature defines classes that will be treated as fixed classes as long as there are other classes swapping in and out. When there are no other classes to force them out, the fixed swapping class will remain in memory and be scheduled ahead of background batch.

5.5 BACKGROUND BATCH PARAMETERS

Background batch is controlled by two parameters: background batch class and background batch swap time.

Background batch class is stored in parameter BBSUBQ. The initial value of -1 is set in SYSINI. It may be modified with the Background Batch Class subfunction of the SCHED. monitor call.

Background batch swap time is stored in parameter SCDBBS. The initial value of zero is defined in COMMON. It may be modified with the Background Batch Swap Time subfunction of the SCHED. monitor call.

Any class may be designated as the background batch class. In general, it has a zero primary percentage and a zero secondary allocation, but this is not a restriction. If background batch has a primary percentage, it is guaranteed a certain level of response. If it has a secondary allocation, it is allowed a share of leftover resources. In any event, it is also scanned whenever there are no other classes to run. The negative initial value implies there is no defined background batch class.

The background batch swap time defines the rate in ticks at which background batch jobs can be swapped. In situations where the timesharing load fluctuates between existence and nonexistence of timesharing jobs, it can be used to prevent thrashing.

5.6 RESPONSE FAIRNESS FACTOR

The response fairness factor is stored in parameter SCDJIL. The initial value of 10% is set in SYSINI. It may be modified with the Response Fairness subfunction of the SCHED. monitor call.

The response fairness factor defines the percentage of time that jobs are scheduled in the order in which they were swapped in versus scheduling by the class scheduling scan. (A list of jobs just swapped in is maintained for all jobs that have not yet expired 1 time slice.)

A value of 100% gives the best possible short-term response with reduced accuracy when jobs do not exist on the swapper in sufficient numbers to satisfy the desired primary percentages.

A value of 1% gives the best possible accuracy with reduced response when many jobs in memory are in classes that are rarely scheduled.

The range of acceptable values for response fairness factor are from 1% to 100%. Values of 10% and above are recommended for acceptable short-term response. A zero value is not allowed.

5.7 AVERAGE SWAP TIME

The average swap time is stored in variable SCDSWP. It may be modified with the Average Swap Time subfunction of the SCHED. monitor call. The initial value is calculated in ONCMOD by multiplying the time it takes to swap one page by the specified average job size, PAVJSP, and adding in the swapper latency time. The time required to swap one page depends on the speed of the installation's swapping device.

The default value for PAVJSP is 20 pages, or 10K.

The average swap time is used to calculate when the swapper should advance to the next class in the primary table when there are no jobs in the system to swap. This parameter is required to achieve correct swap-in rates for fixed classes when there are no jobs in any other classes. Fixed classes have no secondary allocations. In fact, they can only swap in when the primary percentage pointer has been advanced to an entry for their class.

5.8 JOB CLASS

The class to which each job belongs is stored by job number in bits 14 through 17 of the table JBTSCD. The initial value at system start up is all zeros.

The job's class is set by LOGIN using the Job Class subfunction of the SCHED. monitor call. It can also be set by the SCDSET program.

5.9 CLASS RUNTIME

The class runtimes are set by the monitor and are read by the Runtime subfunction of the SCHED. monitor call through the SCDSET program. The values are reset to zero whenever the primary percentages are changed.

CHAPTER 6

DETERMINATION OF PARAMETERS FOR SCHEDULER

This chapter uses the scheduler as an example of how the default scheduling parameters are determined. This chapter also discusses the rationale behind a choice of parameters for a specific system.

The Western Michigan University (WMU) computer system is a KI10 with 160K of memory, six RP02 disk drives, and two RP03 disk drives (on one channel), and two RD10 swapping disks (on a second channel), and two TU20 tape drives (on the I/O BUS). The system is configured for 74 jobs. The monitor is 6.02A with virtual-memory option (the swapper and scheduler are modified to be equivalent to the WMU class scheduler in 6.03).

The job mix is made up of a wide variety of programs. Compilations are primarily BASIC and FORTRAN with a fair amount of COBOL, MACRO, and ALGOL. User programs and system library programs cover many areas including simulation, mathematics, statistics, engineering, chemistry, physics, management, and so forth.

Most activity is terminal oriented. Of the 74 job slots, 3 are allocated for BATCH. The maximum user core is 35K during prime time. The average job size is 10K. The majority of jobs are relatively small and conversationally oriented (that is, TECO, LINED, and small student programs). There are a fair number of large jobs that make heavy use of the CPU and/or disk I/O (large compilations, STATPACK, and virtual-memory jobs).

The overall performance objectives are:

1. To provide good response to conversational jobs (PQ1).
2. To maintain a reasonable level of system throughput for system and I/O users.
3. To provide a good balance of CPU versus I/O jobs in core so that multiprogramming is effective over a wide range of job mixes.

The discussion of specific parameter values in this chapter parallels the general discussion in Chapter 5. For each section in Chapter 5, there is a corresponding section in this chapter describing how the parameters are determined for the scheduler.

6.1 PROCESSOR QUEUE TIME SLICES

For in-core protect time and quantum runtime, the ONCMOD tables are indexed by an RD10 as a primary swapping device. The values referenced in ONCMOD tables and the values transferred to scheduling tables are indicated in the following.

Table 6-1 lists the parameter values for in-core protect time.

Table 6-1
In-Core Protect-Time Parameter Values

Scheduling Parameter			ONCMOD Parameter		
Name	Value	Units	Name	Value	Units
PROT	0	microseconds	PROTTB	0	microseconds
PROT0	3000000	microseconds	PRT0TB	3000000	microseconds
PROTM	3000000	microseconds	PRTMTB	3000000	microseconds
PROT1	180	ticks	PRT0TB	3000000	microseconds

These values imply a fixed 3-second in-core protect time for all jobs, regardless of job size, both at swap-in and when requested for time-slice expiration.

If desired, PROTTB could be set nonzero to vary the assignment at swap-in by job size. PRTMTB would need to be modified also to define the maximum allowed value.

Table 6-2 lists the quantum runtime parameter values for PQ1, which are generated directly in the scheduling tables in COMMON.

Table 6-2
PQ1 Quantum Runtime Parameter Values

Scheduling Parameter			COMMON Parameter		
Name	Value	Units	Name	Value	Units
QADTAB(0)	8	ticks	QQRUN1	8	ticks
QMLTAB(0)	0	ticks		0	ticks
QMXTAB(0)	8	ticks	QQRUN1	8	ticks

These values imply a fixed 8 ticks for all PQ1 jobs, regardless of job size.

These values may be changed by inserting the new values directly in COMMON, or by inserting code in ONCMOD to set up the values.

Table 6-3 lists the quantum runtime parameter values for PQ2.

Table 6-3
PQ2 Quantum Runtime Parameter Values

Scheduling Parameter			ONCMOD Parameter		
Name	Value	Units	Name	Value	Units
QADTAB(1)	45	ticks	ADDTAB	750000	microseconds
QMLTAB(1)	45	ticks	MULTAB	750000	microseconds
QMXTAB(1)	90	ticks	MAXTAB	1500000	microseconds

The value of QRANGE in COMMON is 45K.

The values imply a base quantum runtime of 0.75 second for a 1K job. This grows one tick per K of job core size to a maximum of 1.5 seconds for a 45K job. Thereafter, it is a fixed 1.5 seconds. Because the average job size at WMU is about 10K, the average PQ2 quantum runtime is approximately 1 second.

Table 6-4 lists the quantum runtime parameter values for HPQs, which are generated directly in the scheduling tables in COMMON.

Table 6-4
HPQ Quantum Runtime Parameter Values

Name	Value	Units
QADTAB(2)	2	ticks
QMLTAB(2)	0	ticks
QMXTAB(2)	2	ticks

This implies a fixed quantum runtime of 2 ticks for all HPQ jobs, regardless of size.

The rationale for each of the processor queue time slices is as follows.

6.1.1 PQ1 Time Slice

In PQ1, the quantum runtime of 8 ticks allows very fast response for a very short period of time. In-core protect time is a constant 3 seconds.

At WMU, most PQ1 jobs finish processing and return to long-term wait within the 8 ticks allowed by the PQ1 quantum runtime. Table 6-5 lists the number of PQ1 jobs blocking to long-term wait as a function of time.

Table 6-5
Percent of PQ1 Jobs Blocking to Long-Term Wait as Function of Time

Percent Blocking	CPU Ticks Used
50%	Less than 6 ticks (1/10 second)
80%	Less than 20 ticks (1/3 second)
95%	Less than 50 ticks (5/6 second)

PQ1 jobs that do run long enough to expire their time slices are requeued to PQ2, assigned a PQ2 amount of quantum runtime, and retain their remaining in-core protect time. This reduces their response priority to the level of PQ2, but allows the job to compute at least as long as a PQ2 time slice.

As Table 6-5 shows, less than 5% of the PQ1 jobs compute long enough to use the additional PQ2 time slice. For those that do, a small reduction in swapping rate is achieved with little impact on the other jobs in PQ2.

6.1.2 PQ2 Time Slice

For PQ2 jobs, the quantum runtime is 0.75 second to 1.50 seconds, depending on job size. In-core protect is a fixed 3 seconds. These values give good response, low overhead, and optimum balance between CPU and I/O-bound jobs.

Good response is achieved when the PQ2 time slice is small enough so that jobs swapping in can find sufficient space in memory to come in (free space or jobs with expired time slices). One measure of good response is that the swapper can achieve full speed during periods of heavy demand for short-term response. Another measure is the average swap time required to swap in a PQ1 job, that is, the time from when the job enters PQ1 to the time it is swapped in.

The scheduler overhead increases as time-slice parameters are made smaller. Also, the PQ2 swapping rate goes up, making less swapper capacity available to PQ1 jobs.

The goal is to make the PQ2 time slice small enough to allow good response, and large enough to achieve low overhead and low PQ2 swapping rate.

A second goal is to make the ratio of in-core protect to quantum runtime such that an optimum balance is achieved between CPU and I/O jobs. This can be measured by looking at percent CPU utilization versus utilization of the disk system. Disk rates are measured in terms of the number of disk blocks transferred.

CPU utilization, disk rates, swapper rate, swap times, overhead, PQ1 swap rate, and PQ2 swap rate can be monitored with the system performance analysis package. This has been done to ensure that optimum values are in use. A reevaluation is done periodically, because the system load characteristics change over time.

To illustrate the importance of a proper ratio of in-core protect time versus quantum runtime, a test was run with simulated jobs to show the

effect of incorrect parameters. The job mix contains an equal number of CPU and I/O-bound jobs. The same job mix was run with two different monitors, one with WMU standard parameters (approximately 3 to 1), and one with incorrect parameters (approximately 1.5 to 1).

The test results are tabled below:

Table 6-6
Example of Effect of Incorrect Parameters

Monitor	CPU	Disk Blocks/Minute
Standard Parameters	94%	3184
Incorrect Parameters	92%	2219

The standard parameters produced better I/O rate with no decrease in CPU utilization.

Note that the PQ2 time slice is sufficient to slow the PQ2 swapping rate, but is not sufficient to bring the PQ2 swapping rate up to a minimum level. To accomplish this, the swapping and scheduling fairness counts are necessary. This is discussed in Section 6.1.3.

6.1.3 HPQ Time Slice

HPQ jobs are assigned quantum runtimes of 2 ticks and in-core protect times of 3 seconds. The extremely small quantum runtimes allow very fast alternation between HPQ jobs. The in-core protect times are immaterial, because WMU never has more HPQ jobs than can fit in core at once.

6.2 SWAPPING AND SCHEDULING FAIRNESS COUNTS

Table 6-7 lists the default values for swapping and scheduling fairness counts that are used at WMU.

Table 6-7
Default Values for Swapping and Scheduling Fairness Counts

Parameter	Description	Value
IFC0	Swapping fairness	5
SFC0	Scheduling fairness (CPU0)	20
SFC1	Scheduling fairness (CPU1)	20

The swapping and scheduling fairness counts prevent PQ1 jobs from taking over the system. PQ1 jobs are swapped in and scheduled ahead of PQ2 jobs. If they exist in sufficient numbers, they can fill memory and take over the system. The fairness counts allow PQ1 jobs to have the highest priority up to a limit. After that, PQ2 jobs have priority.

DETERMINATION OF PARAMETERS FOR SCHEDULER

For good response, PQ1 should get the majority of swapper capacity. Assuming the swapper is operating at 100% capacity, a good goal is 80% for PQ1 jobs and 20% for PQ2 jobs. This allows good response for PQ1 jobs and provides good system throughput for PQ2 jobs.

If PQ1 jobs are not restricted by fairness counts, system throughput will be severely degraded during periods of heavy demand for short-term response. This is because PQ1 typically blocks to long-term wait very quickly after swap-in. Without restraint, memory becomes filled with jobs that are not runnable. The swapper cannot swap them out as fast as they expire. In this case, CPU utilization goes down and lost time goes up.

There are two direct measures of fairness. First, PQ2 jobs should get at least a certain minimum of swapping capacity whenever there are sufficient numbers of PQ2 jobs in the system. Second, the machine should not be filled with unrunnable jobs (that is, jobs in long-term wait, which are generally expired PQ1 jobs). Both of these variables can be measured with the system performance analysis package.

At WMU, the PQ2 swapping rate is approximately 20% when sufficient jobs exist and the swapper is operating at capacity. The average amount of core occupied by unrunnable jobs is approximately 20 pages (10K) out of a total user area of 91K.

To illustrate the effect of fairness counts on the WMU system, a set of simulated jobs was created containing a mix of PQ1 and PQ2 jobs similar to that seen on the real system. Performance was measured for a wide range of swapping fairness counts. (See Table 6-8.)

Table 6-8
Example of Effect of Fairness Counts

Swapping Fairness	CPU Utilization	Number of PQ1 Jobs Swapped in per Minute	Number of PQ2 Jobs Swapped in per Minute
30	29.9	101.2	3.4
16	37.7	98.9	6.2
9	48.7	96.2	10.7
5	61.7	96.1	19.1
3	72.2	88.5	29.6
1	91.1	64.9	64.3

The data shows that as more PQ2 jobs are swapped in (fairness threshold is lowered), the CPU utilization is increased. At the same time, the PQ1 swapping rate is decreased, showing a corresponding impact on short-term response.

The value of 5 for swapping fairness was chosen at WMU because it produces good PQ2 throughput with very little impact on short-term response (PQ1 swapping rate). Scheduling fairness was arbitrarily set to 20. In most cases this has little effect, because PQ1 jobs typically expire so fast that PQ2 jobs run without the need for scheduling fairness.

6.3 IN-CORE FAIRNESS FACTOR

WMU uses the default value of 50% for the in-core fairness factor. This gives good response to jobs that do GETSEGs without allowing them to take over the swapping.

Values below 50% are not recommended because too many low segments would exist in memory in an unrunnable state.

6.4 CLASS QUOTAS AND MICROSCHEDULING INTERVAL

The WMU class scheduler comes up to Round Robin mode. The SCDSET program is run shortly after start up with an OPSER.ATO file to place the scheduler in Class Scheduler mode. The file defines primary percentages of 95% for class 0, and 5% for classes 1 and 2. Secondary allocations and fixed swapping bits are set in a variety of permutations to test the response and accuracy of the class scheduler.

The microscheduling interval is set to 30 ticks or 0.5 second.

WMU tested values for the microscheduling interval in the range 1 to 60 ticks. The smaller values gave the best accuracy and smoothest response. In this range, there was no measurable difference in scheduler overhead.

6.5 BACKGROUND BATCH PARAMETERS

The WMU system starts up with the default value -1 for background batch class and 0 for background batch swap time. The SCDSET program, which runs at start up, defines class 15 as the background batch class with a background batch swap time of 120 ticks, or 2 seconds. Primary percentage and secondary allocation are both 0.

Values of 60 through 180 ticks were tried in live operation under various system loads. The value of 120 ticks appears to adequately prevent thrashing.

6.6 RESPONSE FAIRNESS FACTOR

The WMU system is assembled with the default value of 10% as the response fairness factor. This is overridden at start up by the SCDSET program to a value of 100%, which produces the best possible response at all times.

WMU has tried a range of values from 1% to 100% on the live system under a wide variety of loads. A value of 10% gives good response with very good accuracy. Values below 10% produce poor response, and are not recommended. Values above 10% did not noticeably improve response, but did reduce accuracy.

The present value of 100% is arbitrary. WMU is presently more concerned with response than accuracy.

6.7 AVERAGE SWAP TIME

The WMU system is assembled with the average job size, PAVJSP, set to the default value of 20 pages, or 10K. With the WMU primary swapping device, an RD10, this yields an average swap time of 9 ticks.

This value of the parameter gives very accurate allocation of time to fixed classes during periods when no other classes are present.

6.8 JOB CLASS

The WMU system begins operation with all job classes set to default values of 0. Jobs are placed in the appropriate class as they log in. WMU uses classes 0, 1, 2, and 15.

6.9 CLASS RUNTIME

WMU class runtimes are set to the default values of 0 at start up.

CHAPTER 7

DETAILED DESCRIPTION OF SCHED. MONITOR CALL

This section describes the macro code for the SCHED. monitor calls. This code is included in the class scheduler only.

7.1 SCHED. TO SCDQTA SECTION

The scheduling parameters are defined by the system administrator through the SCDSET program, which uses the SCHED. monitor calls to store the parameters in the monitor data base. The SCHED. monitor calls store most parameters and retrieve all parameters. A description of the detailed code for each of the SCHED. monitor calls follows.

At SCHED. the argument block for the SCHED. monitor call is interpreted and checked for legality. A dispatch is made to the appropriate read or write routine based on the function code. Functions 1, 4, and 8 are not used in the WMU class scheduler.

7.1.1 Function 0

Routine SCHRSI reads the microscheduling interval (SCDINT).

Routine SCHWSI writes SCDINT. It also forces a new scheduling interval to begin. If the microscheduling interval goes to zero, the scheduler is placed in Round Robin mode by clearing RRFLAG.

7.1.2 Function 2

Routine SCHRQT reads the primary percentages for each class up to the class specified in the argument block. First, check the class number for legality. Then, for each class up to that number, load the primary percentage and status bits from table CLSSTS, and store them in the user-specified area.

Routine SCHWQT stores the primary percentages for any number of classes. The first argument specifies the number of classes to be stored. Each following argument contains the class number and status bits in the left half, and the primary percentage in the right half. First, check the class number for legality. Then, store the primary percentage and status bits in table CLSSTS. After all of the arguments have been processed, zero the table of runtimes by class (CLSRTM).

At SCHWQ2, build a table of all classes with positive primary percentages in SQSCAN. Each entry in the table is of the form XWD 0, class number. Store the total number of classes with primary percentages in CNTSTS. If no classes have a primary percentage or the percentages do not add to 100%, place the scheduler in Round Robin mode by clearing RRFLAG, and leave SCHWQT.

At SCHWQ6, pick the next class to be entered into the primary scan table PSQTAB. If there is only one class, go to SCHWQ9 to store that class. Otherwise, determine which class is most overdue to be picked at SCHWQ7. For each class in SQSCAN, add its primary percentage to the relative priority, which is stored in the left half of the SQSCAN table. Weight the relative priority by multiplying by the class primary percentage, and if the product is the largest seen so far, set AC P1 to point to this class. Repeat from SCHWQ7 until all classes have been tested.

At SCHWQ9, store the selected class as the next entry in PSQTAB. Also, subtract 100% from its relative priority to reflect the fact that it is no longer overdue to be selected. Repeat from SCHWQ6 until all 100 entries have been stored in PSQTAB. Set entry 101 to entry 1 for use by CPU1.

This algorithm guarantees that each class will be selected the number of times specified by its primary percentage. Also, this algorithm spaces the entries optimally if each percentage is a multiple of ten, and does a very good job on most other cases.

7.1.3 Function 3

Routine SCHRTS reads the base quantum runtimes for either PQ1 or PQ2.

Routine SCHWTS stores the base quantum runtimes for PQ1 or PQ2 in the QADTAB table. The first word in the argument block specifies the number of arguments to follow. A code 1 in the left half of the argument specifies PQ1, and a code of 2 in the left half of the argument specifies PQ2. The right half of the argument contains the new value for the base quantum runtime.

7.1.4 Function 5

Routine SCHRJC reads the class numbers for all jobs in the system up to the job specified in the argument block. First, check the job number for legality. Then, for all jobs up to that job number, load the job's class from table JBTSCD and store it in the user-specified area.

Routine SCHWJC places any number of jobs into their proper scheduler classes. The first argument specifies the number of jobs to be reclassified. Each following argument contains the job number in the left half and the new class number in the right half. First, make sure that the job number is valid and that the job is logged in. Then, check the class number for legality and store the new class number in table JBTSCD. If the job is in PQ2, set the changing subqueue bit (JS.CSQ) and requeue the job.

7.1.5 Function 6

Routine SCHRMC reads the constant added to in-core protect time (PROT0).

Routine SCHWMC writes PROT0.

7.1.6 Function 7

Routine SCHRCT reads the runtime used by each class up to the class specified in the argument block. First, check the class number for legality. Then, for each class up to that number, load the runtime used by that class from table CLSRTM and store it in the user-specified area. Runtimes are stored in ticks and represent the CPU time used in PQ2 since the primary percentages were last changed. The write option is illegal for function 7.

7.1.7 Function 9

Routine SCHRPF reads the multiplier used to calculate in-core protect time (PROT).

Routine SCHWPF writes PROT.

7.1.8 Function 10

Routine SCHRCD reads the default class for a new job (DEFCLS).

Routine SCHWCD sets DEFCLS.

7.1.9 Function 11

Routine SCHRRC reads the constant used for assigning in-core protect time on requeue because of time-slice expiration (PROT1).

Routine SCHWRC writes PROT1.

7.1.10 Function 12

Routine SCHRPM reads the maximum value of in-core protect time (PROTM).

Routine SCHWPM writes PROTM.

7.1.11 Function 13

Routine SCHRPM reads the in-core protect time constant (PROT0).

Routine SCHWRC writes PROT0.

7.1.12 Function 14

Routine SCHRML reads the quantum multipliers for either PQ1 or PQ2, or the scale factor used in calculating quantum runtime (QRANGE).

Routine SCHWML stores the quantum multipliers for PQ1 or PQ2 into the QMLTAB table. As in function 3, a code of 1 specifies PQ1 and a code of 2 specifies PQ2. A code of 3 specifies a new value for QRANGE.

7.1.13 Function 15

Routine SCHRMX reads the maximum quantum runtimes for either PQ1 or PQ2.

Routine SCHWMX stores the maximum quantum runtimes for PQ1 or PQ2 into the QMXTAB table. A code of 1 specifies PQ1 and a code of 2 specifies PQ2.

7.1.14 Function 16

Routine SCHRSQ reads the secondary allocations for each class up to the class specified in the argument block. First, check the class number for legality. Then, for each class up to that number, load the secondary allocation from table CLSQTA and store it in the user-specified area.

Routine SCHWSQ stores the secondary allocations for any number of classes. The first argument specifies the number of classes to be stored. Each following argument contains the class number in the left half and contains the secondary allocation in the right half. First, check the class number for legality. Then, store the secondary allocation of the class in the table CLSQTA. After all arguments have been processed, store the number of the highest class with a positive secondary allocation in MAXQTA.

7.1.15 Function 17

Routine SCHRIQ reads the response fairness factor (SCDJIL).

Routine SCHWIQ writes SCDJIL. The value is a percentage and must be positive.

7.1.16 Function 18

Routine SCHRSS reads the average swap-time estimate (SCDSWP).

Routine SCHWSS writes SCDSWP. The value is specified in ticks.

7.1.17 Function 19

Routine SCHRBB reads the background batch class (BBSUBQ).

Routine SCHWBB writes BBSUBQ. The value must be a legal class number or -1 if no background batch is desired.

7.1.18 Function 20

Routine SCHRBS reads the background batch swap-time interval (SCDBBS).

Routine SCHWBS writes SCDBBS. The value is specified in ticks.

7.1.19 Function 21

Routine SCHRSF reads the scheduler fairness factor for CPU0.

Routine SCHWSF writes the scheduler fairness factor for CPU0. The same value is stored for CPU1 if it exists. The value must be positive.

7.1.20 Function 22

Routine SCHRSW reads the swapper fairness factor (MAXIFC).

Routine SCHWSW writes MAXIFC. The value must be positive.

7.1.21 Function 23

Routine SCHRIO reads the in-core fairness factor (SCDIOF).

Routine SCHWIO writes SCDIOF. The value is a percentage and must be positive.

7.1.22 Function 24

Routine SCHRCS reads the core scheduling interval (SCDCOR). The value is converted to seconds before being returned to the user. SCDCOR is used to determine whether in-core protect times are used in scheduling.

Routine SCHWCS converts the user argument from seconds to tick-pairs, and stores the result in SCDCOR.

7.2 SCDQTA TO SCDQT7 SECTION

This section checks for the end of the microscheduling interval and performs all necessary functions when the interval expires. Routine SCDQTA is called once every tick.

If no microscheduling interval is defined (SCDINT=0), or no primary classes are defined (CNTSTS=0), return immediately because the scheduler is operating in Round Robin mode. Otherwise, set RRFLAG nonzero to cause the scheduler to operate in Class Scheduling mode.

If the current microscheduling interval is not yet over (UPTIME<SCDTIM), return. Otherwise, store the end of the new microscheduling interval in SCDTIM. Store the time at which response fairness is no longer in effect in SCNJIL.

Advance the primary scan pointers to the next class for both CPUs. For each CPU, load the primary class into AC T1 and the address of the subqueue scheduling scan table into T4, and call SCDSST to build the scan table.

At SCDSST, set the first entry in the scan table to the primary class. Build the secondary scan table in the remaining locations of the scan table. All classes with secondary allocations except the primary class are entered into the table in the form: XWD class, secondary allocation. The sum of the secondary allocations of all classes entered into the secondary scan table is accumulated in SSSUM.

At SCDQT4 select a random integer in the range 0 to 1 SSSUM-1. This integer determines which class will be selected next for insertion into the subqueue scheduling scan table for the microscheduling interval. The secondary allocations of each entry in the secondary scan table are successively subtracted from the random integer until it goes negative. The class that causes it to go negative is selected as the next class to insert into the subqueue scan table. Thus, the probability of any given class being selected is equal to its secondary allocation divided by the total of all remaining secondary classes (SSSUM).

Eliminate the selected class from further consideration by moving the bottom entry up on top of it, and by subtracting its secondary allocation from SSSUM. Store the selected class as the next entry in the scan table. Repeat from SCDQT4 until all entries in the secondary scan table have been incorporated into the subqueue scheduling scan table. A zero terminates the table.

PART 5

DISK I/O PROCESSING

Disk I/O Processing

File: DISKIO.RNO
Date: April 1978
Edition: 1

The information in this document is subject to change without notice and should not be construed as a commitment by Digital Equipment Corporation. Digital Equipment Corporation assumes no responsibility for any errors that may appear in this document.

The software described in this document is furnished under a license and may be used or copied only in accordance with the terms of such license.

Digital Equipment Corporation assumes no responsibility for the use or reliability of its software on equipment that is not supplied by DIGITAL.

Copyright © 1978 by Digital Equipment Corporation

CONTENTS

		Page
1.0	GENERAL DISK I/O FLOW	2
2.0	DUAL PORT HANDLING	4
3.0	EXCEPTION CONDITIONS	4
3.1	Data Transfer Errors	4
3.1.1	ECC Correctable Error	4
3.1.2	Non-data Error	5
3.1.3	Data Error	5
3.2	Seek and Status Errors	6
3.2.1	Medium-on-line = 0	6
3.2.2	Drive Powered Up	6
3.2.3	Seek Incomplete	6
3.2.4	Hung Device	6
3.2.5	Rib Errors	6
3.3	RAE Errors	6
4.0	BAT BLOCKS	7
5.0	DSKRAT	7
INDEX		Index-1

1.0 GENERAL DISK I/O FLOW

This discussion assumes that the disk request has been processed to the point that an I/O list has been built and the initial sector address is known. The disk is not necessarily on the correct cylinder. The following flow describes the general processing; subsequent text will describe it more fully.

```
1-  Calculate required cylinder.
2-  If seek required then
3-      If data transfer in progress
        (non-massbuss device or massbuss
        device with active transfer on this unit)
4-          Queue request to this unit
5-          Exit
6-      Else
7-          Start seek
8-          Exit
9-  Else
10-     If data transfer in progress then
11-         Queue request for channel
12-         Exit
13-     Else
14-         Disable attention interrupts
15-         Start transfer
16-         Exit
17- End


18- On interrupt
19-     Read drive number and register # from RH
20-     Read attention summary register
21-     For each on-bit in summary register do
22-         If corresponding drive was not transferring
23-             If seek complete then queue request for channel
24-             Else process status (eg. drive coming on line)
25-     If data transfer complete then
26-         If hardware detected error then
27-             Perform error recovery
28-         Compare channel termination with predicted termination
29-         If software detected error
30-             Perform error recovery
31-     For each unit with queued requests do
32-         Select next seek and start it
33-     If channel queue (already positioned drives) is not empty then
34-         Select best transfer and start it
35-     Restore register # and drive # to RH
36- Dismiss interrupt
```

The correct cylinder is determined by dividing the sector number by the number of sectors per cylinder. To determine if a seek is needed, (2) the cylinder number is compared with the current cylinder number, which is remembered from the last transfer. (There are some limited conditions under which the drive will not be on the cylinder which is recorded in the software. In these cases, the implied seek of the drive will be used). The system can only start a seek if the drive is idle (for non-MASSBUS drives, both the drive and the controller must be idle). Therefore, if there is a data transfer in progress, the request is queued for the unit and will be started at a later time at interrupt level (4). If the drive is free, a seek will be started. If the drive is already on the correct cylinder, the seek logic is bypassed. If the drive and channel are not already busy, then th-

transfer is started; otherwise, the request is added to a queue for the required channel to be started at interrupt level at a later time. A transfer may range in size from a single word (128 words) to a whole cylinder; TOPS10 attempts to perform the longest possible transfer in order to maximize I/O throughtput. The system never attempts an implied seek in the middle of a transfer. Such a user request would be broken into two or more transfers with explicit intermediate seeks. Also, in order to simplify the code considerably, attention interrupts are disabled while doing a data transfer.

When an interrupt occurs, the system may or may not have just completed a data transfer. Since attention interrupts were disabled during the data transfer, there may be a number of outstanding attention conditions when the interrupt is actually honored. First, the system reads the attention summary register. Each drive (except the one which was completing a transfer) is checked for an attention bit on. If there is an attention bit on, and if there is a seek complete, the transfer request is added to the channel queue to be started for I/O. If there was no seek in progress, then the drive has just come on line or powered up (see later discussion for these conditions).

Once all outstanding seeks are processed, the data transfer completion is handled. If there was no error or after error correction (see error recovery later), the channel termination word is compared to the predicted channel termination word. If the check fails, then error recovery is started. After completing the processing for the interrupt, any outstanding seeks are started. For each drive, the closest (shortest) seek is the one selected for startup (a fairness count will cause the system to select the oldest transfer every 'n'th time). After seeks are started, the channel queue is checked for positioned drives and the transfer with the shortest latency is started (again unless the fairness count says otherwise). SWAPPER requests receive preference over file I/O (unless fairness count expires).

There is some special processing for interrupts on a MASSBUS device caused by the fact that the system may be attempting some operation using the device registers at UUO level at the time of the interrupt. When the interrupt occurs, the system reads RHxx and saves the drive and register number to which the RH was pointing. Before dismissing the interrupt, a DATAO is done to restore the drive number and register number. The need for reading the register from the RH at interrupt and restoring them before dismissing the interrupt is made worse by the fact that the system must wait 3 microseconds after the DATAO specifying what data is wanted before the DATAI can read the data.

There are other special considerations with the front end disk unit. In general, both the front end and TOPS10 may attempt to use the disk at the same time. The most frequent conflict occurs at system startup when the front end is using the disk at the same time that TOPS10 is running INITIA on all lines (there is a count of the times that TOPS10 tried to get the disk and found it busy; this normally rises quickly at system startup to about 40 and seldom changes thereafter. It is possible to do an assembly on the front end while timesharing continues on the -10 which might generate considerable conflict).

When the -10 attempts to get the disk and finds that it is in use by the front end, the requests is delayed (with considerable trickery to upper level code) and restarted when the drive can be gotten. Since TOPS10 may complete a seek for the front end drive and have the front end grab the disk and move it before the data transfer is started, it is possible that the drive will not be on the correct cylinder when TOPS10 tries to start the transfer. In this case, implied seek will be used since TOPS10 will not realize that the disk has been moved. This would also happen if TOPS10 got two different requests for the same cylinder and would decide that no seek is necessary when in fact, the front end had moved the heads.

2.0 DUAL PORT HANDLING

The dual port handling is very simple. It occurs only when the system attempts a data transfer on one channel and finds it busy. It then tries the other port. At no other time is the dual port facility used.

At system startup, the system reads the drive type and serial number from each drive on all channels. When the same serial number, drive type is found on two different channels, the disk is determined to be dual ported. One path (the first one found) is then the primary path and the second is the alternate path. When starting a transfer, the system will attempt to use the primary path. If that path is busy, it will then check for the alternate path; if that is available, the transfer is started. Otherwise, the request is placed in the channel queue for the primary channel.

3.0 EXCEPTION CONDITIONS

There are a number of possible error conditions that can occur while TOPS10 attempts to operate the disks. This section will attempt to list the error conditions, the circumstances under which they occur, and the action taken by the system. It will not attempt to show the 'flow diagram' of the error handling in the normal code. In general, the error processing is called as soon as possible after the error is detected.

3.1 Data Transfer Errors

These errors are detected on the completion interrupt for a data transfer (either read or write). These do not include the software detected error of the channel termination word not agreeing with the predicted channel termination word.

3.1.1 ECC Correctable Error - When a transfer terminates with an ECC correctable error the transfer stops after the sector in error. The software will reconstruct the data and restart the transfer at the sector following the sector in error.

3.1.2 Non-data Error - When a transfer completes that is not a data error (for example, a header error) the software will attempt to retry the transfer a number of times before recording the error as a hard error. The retry sequence is:

Retry 10 times
Recalibrate
Seek
Retry 10 times
Recalibrate
Seek
Retry 10 times

If after 30 tries the transfer still fails, the error is considered hard and an error is returned to the user. The data is recorded in SYSERR and a count of hard errors for this device is incremented.

If the count of hard errors reaches a system default (not 25), a message is given to the operator saying that there has been an excessive number of hard errors and the count is zeroed. The expectation is that the operator may want to set some hardware offline or call his field service rep to run a few diagnostics.

3.1.3 Data Error - If a data error (including header compare error) occurs which is not ECC correctable, then the system will retry the transfer and will use the offset register to vary the head position on each side of the track centerline. The retry sequence is:

Retry 16 times on centerline
Offset head to +200 microinches
Retry 2 times
Offset head to -200 microinches
Retry 2 times
Offset head to +400 microinches
Retry 2 times
Offset head to -400 microinches
Retry 2 times
Offset head to +600 microinches
Retry 2 times
Offset head to -600 microinches
Retry 2 times
Return to centerline
Retry disabling stop on error
Retry (every retry except the next to last is done
with stop on error enabled, this enables the recording
of the maximum of information in SYSERR).

For an RP04 the offset distances are twice the above. If the transfer is recovered at the offset position, the drive is left positioned at offset. If the next transfer on that drive is for that cylinder, it is first to be attempted at the same offset. If that fails, the head is returned to centerline and the entire above sequence is tried. If any seek is done, the heads will be on centerline (including transfers which cause the head to return to the cylinder on which an error was recovered at offset). If the device is not an RP04 or RP06 (MASSBUS drive), the error recovery is 10 retries of the sequence: read/write 10 times, recalibrate, seek.

On a hard non-recoverable error, an error is returned to the user and the system remembers the block number in error. When the file is subsequently closed, the system checks for a remembered block number. It starts reading from the bad block number+1 until it finds a good

sector (or 1000 sectors whichever is smaller). This gives the extent of the error region, which is then recorded in both the RIB of the file and the BAT block. If the program does not close the file after the error, but continues processing and hits a second error, the second error is lost.

3.2 Seek and Status Errors

In the attention summary register, on an interrupt, there may be attention interrupts for drives that were not transferring or seeking. In this case, the drive is going through some sort of status change, such as coming on line or going down.

3.2.1 Medium-on-line = 0 - If medium-on-line is 0, the drive has just powered down. It is marked as such in the monitor tables.

3.2.2 Drive Powered Up - If medium-on-line (MOL)=1 and volume valid (VV) =0 then the drive has just powered up. The monitor will read the home blocks and check that the pack is the expected pack on that drive.

3.2.3 Seek Incomplete - On all seek errors, the error is counted and ignored. This will cause the data transfer to use the implied seek facility to perform the actual seek. If that implied seek fails, the data transfer will return an error and the appropriate retry sequence will be started(4.1.2).

3.2.4 Hung Device - Any time a seek or data transfer is started, the monitor starts an independent 'hung timer' that will fire in 7 seconds if the device has not responded with a completion interrupt for the operation.

If the failing request was a seek, then it is retried. If it was a data transfer, the monitor does a CONO to clear BUSY and set DONE. After this, the appropriate retry sequence for a data error is started. If 8 hung retries in a row fail, then the monitor will set the drive offline and tell the operator that it is offline (message is Inconsistent Status for Drive x).

3.2.5 Rib Errors - Every RIB error detected (along with every 'n' hard errors) is reported to the operator.

3.3 RAE Errors

On an RH10, Register Access Error(RAE) is ignored. The hardware will set the Selected Drive RAE at which point error recovery is started.

On the RH20, after every DATAO, a CONSZ on RAE is done, if there was an RAE, then it is cleared and the DATAO is retried. There is also a system count of RAE's per controller for the RH20's.

4.0 BAT BLOCKS

The BAT blocks provide a record of up to 63 errors on the disk. After each detected error (actually when the file is closed), the monitor will update the BAT blocks with the blocks in error and the type of error. It is possible that the BAT blocks will be filled to overflowing and there will be no room for additional entries. The system will leave bad blocks marked as allocated in the SAT table and thus avoid reallocating them. SYSERR will also complain when there are less than 5 entries remaining in the BAT block.

In general, there are some pathological cases where the total damage to a disk is unknown, but a reasonable PM of disks which includes checking SYSERR and DSKRAT and saving, refreshing (or replacing), and restoring packs with many bad spots will avoid difficulties.

5.0 DSKRAT

DSKRAT is a program which can be run to check for RIB errors and the disk space allocated as reported in the SAT table with the allocation as reported by the RIB's of the files on the pack. In general, it will find four kinds of errors:

1. RIB errors - A RIB is not consistent in format with a valid RIB. Lost blocks - These are blocks which are marked as allocated in the SAT but are not a part of any file.

2. Free blocks - These are blocks which are owned by some file on the system but are not marked as allocated in the SAT table. If one of these files is deleted, the system will get a BAZ STOPCD.

3. Multiply Defined blocks - These are blocks which are marked as owned in two or more files.

The safest procedure when a disk has significant problems in terms of free or multiply defined blocks is to BACKUP the pack, refresh it, and restore it.

PART 6

LABS

DECsystem-10 MONITOR INTERNALS COURSE
LAB 1

The goal of the student in this lab is to object patch the command written during week 1 and successfully load and run the monitor.

PROCEDURE

1. Copy LIB:SYSTEM.EXE to your own disk area as SYSTEM.EXE

2. Run FILDDT (.R FILDDT), in patch mode(/P), to insert your patch into SYSTEM.EXE.

3. Enter your routine in the PAT area as described in the CRASH ANALYSIS handbook chapter 9. Be sure to redefine PAT as described.

4. Overlay the COMTAB and DISP entries for the CORE command with the sixbit command value, dispatch bits and dispatch address. The dispatch address will be the first location in your routine in the patch area.

5. Update CONFIG and SYSDAT to reflect a new version of the monitor.

6. Terminate the patching session by entering a control Z to FILDDT.

7. If you did not define any new symbols or delete any existing symbols do a filcom on the original SYSTEM.EXE verses the patched SYSTEM.EXE and justify each word that is different.

8. Load and run the monitor verifying that the command works as expected.

DECsystem-10 MONITOR INTERNALS COURSE

FILDDT Lab session # 2 : examining the running monitor

This lab session requires you to examine portions of the data base of the currently running monitor, particularly the JOB TABLES data base.

To begin, simply type .R FDSYS, and when FILDDT asks "File:", you should respond with /M (crlf). Thereafter, regular DDT commands apply.

Use FILDDT and the Monitor Table Descriptions to answer all the following questions, (note: answers should consist of the table name and word label plus the data)

I. Job tables

A. There is a job running under [75,3]. Learn the following:

1. What program is it running? What ppn is its high segment from?

2. How much core is it running in? Where in core is it?

3. What is its wait state code?

B. Find the PDB for this job.

1. How much run time has it accumulated? Kilo-core ticks?

2. What is its MCU? Is it swappable?

Decsystem-10 Monitor Internals Course
Lab3

FILDDT Session To Begin Crash Analysis

The purpose of this lab description is to guide you through the preliminary steps in crash analysis. The crash analysis worksheet will be the basis for all further labs. Prior to completing the worksheet for this lab do the activities listed below.

After completing the worksheet you should be able to describe what part of the monitor data base was sabotaged to produce this crash and list the correct data base value.

1. Before you start FILDDT, it will be very helpful if you get a SYSTAT of the crash. Type the Monitor command(s) needed to cause SYSTAT to examine the status of LIB:SER001.EXE and write the output in your disk area as file SYSTAT.TXT.

2. Print SYSTAT.TXT on the hardcopy terminal in the lab and retrieve the resulting listing. Keep this listing next to your terminal for further reference while you are using FILDDT.

3. Run SYS:FILDDT.EXE. Make a monitor specific FILDDT by typing the FILDDT command(s) needed to load the symbols from LIB:SYSTEM.EXE. Type the monitor command(s) needed to save the resulting monitor specific FILDDT in your disk area as FD.EXE.

4. Run the FD.EXE you just created. Type the FILDDT command(s) needed to examine SER001.EXE.

5. Complete the crash analysis worksheet for this crash particular crash. note the crash dump worksheet supplement which explains how to obtain the data necessary to complete the worksheet.

INITIAL CRASH DUMP WORKSHEET

1. CRASH FILE__________________SERIAL #___________PROCESSOR______

2. CRASH TIME AND DATE____

3. WHAT INTERRUPT LEVELS WERE IN PROGRESS: PISTS _______________

(1)_____(2)_____(3)_____(4)_____(5)_____(6)_____(7)_____

4. HARDWARE STATUS AT TIME OF CRASH

UPTSTS_________ EPTSTS_________ APR STATUS_______

UPMP _________ EPMP__________ CURRENT AC BLOCK ________

5. WHAT CAUSED THE CRASH DUMP?
STOP CODE_____ NON-ZERO IN 30_____ 407 RESTART_____ OTHER_____

6. IF THE STOP CODE WAS YOUR ANSWER TO 5, ANSWER THE FOLLOWING:

STOPCODE NAME_______ STOPCODE MODULE_______

STOPCODE DESCRIPTION ____

____

STOPCODE TYPE?
HALT_____ STOP_____ JOB_____ DEBUG_____ OTHER_____

DATA ITEM TESTED AND TEST CONDITIONS____________________________

____

EXPECTED VALUE_____________________ ACTUAL VALUE_______________

7. CURRENT JOB_______ PPN_______ PROGRAM_______

8. WHAT CYCLE DETECTED OR EXPERIENCED THE ERROR?

UUO_____ MONITOR_____ DEVICE INTERRUPT_____ OTHER_____

9. WHICH MAJOR PROCESS WITHIN THE CYCLE?
   IF UUO:
      PRE-DISPATCH____ COMMON I/O CODE____ SPECIFIC CODE____ POST DISPATC

   IF MONITOR:
      TIME ACCOUNTING___ TIMING REQUESTS___ HUNG CHECK___ REQUEUE___
      SWAPPING___ SCHEDULING___

   IF DEVICE INTERRPUT:
      DEVICE______ STATUS_______ RETRY______ BUFFER CHECK_____
      DEVICE START/STOP________ DISMISS_____

10. LAST 10 STACK ENTRIES:

	VALUE	ROUTINE
TOP OF STACK	______________	________________________
	______________	________________________
	______________	________________________
	______________	________________________
	______________	________________________
	______________	________________________
	______________	________________________
	______________	________________________
	______________	________________________
	______________	________________________

11. ANALYSIS OF THE CAUSE OF THE CRASH.

____

____

____

____

____

____

____

INITIAL CRASH DUMP WORKSHEET SUPPLEMENT

THE WORKSHEET WAS DESIGNED FOR INSTRUCTIONAL USE IN ELEMENTARY CRASH ANALYSIS THE PURPOSE OF THE INITIAL CRASH DUMP WORKSHEET IS TO STRUCTURE THE DATA COLLECTION PROCEDURE NECESSARY TO ANALYZE THE CAUSE OF A SPECIFIC MONITOR CRASH. THE INFORMATION TO BE RECORDED ON THIS WORKSHEET IS JUST A SMALL SUBSET OF ALL THE INFORMATION AVAILABLE IN A CRASH DUMP.

THE PURPOSE OF THIS SUPPLEMENT TO THE DUMP WORKSHEET IS TO EXPLAIN WHERE THE ITEMS IN THE WORKSHEET MAY BE FOUND IN A CRASH DUMP AS WELL AS HOW TO INTERPERT THEIR CONTENTS.

1). THE CRASH FILE NAME IS JUST THE NAME OF THE CRASH DUMP, IE SER001.EXE

THE PROCESSOR SERIAL NUMBER MAY BE FOUND IN .COASN FOR CPU0 AND .C1ASN FOR CPU1.

THE PROCESSOR TYPE MAY BE DETERMINED FROM THE SERIAL NUMBER ACCORDING TO THE FOLLOWING SERIAL NUMBER (IN DECIMAL) ASSIGNMENTS:

```
              KA < 513
     512 <    KI < 1025
    1024 <    KL < 4097
    4096 <    KS
```

DEPENDING ON PROCESSOR TYPE FILDDT SHOULD BE SET UP TO MAP ADDRESSES.

2) THE CRASH DATE AND TIME MAY BE FOUND IN LOCATIONS: LOCYER, LOCMON, LOCDAY, LOCHOR, LOCMIN, LOCSEC IN DECIMAL. THIS IS USEFUL FOR CORRELATION WITH OTHER EVENTS THAT OCCURED AT THE TIME OF THE CRASH, IE HARDWARE FAILURES ETC.

3). FOR STOP CODE CRASHES PISTS WILL CONTAIN THE RESULTS OF A CONI PI. BITS 21-27 DESCRIBE THE INTERRUPT IN PROGRESS AS DESCRIBED IN THE HARDWARE REFERENCE MANUAL SECTION 3.2.

4). THE HARDWARE STATUS MAY BE FOUND AS FOLLOWS:

```
    UPMP- UPTSTS  KL  BITS 23-35
          EUBSTS  KI  BITS 5-17
    EPMP- EPTSTS  KL  BITS 23-35
          EUBSTS  KI  BITS 23-35

CURRENT AC BLOCK- UPTSTS  KL  BITS 6-8
                  EUBSTS  KI  BITS 1-2

APR STATUS  - APRSTS
```

INTERPERTATION OF THE BITS IN APRSTS INDICATE VARIOUS PROCESSOR ERRORS AS DESCRIBED IN THE HARDWARE REFERENCE MANUAL.

5). THE CAUSE OF THE DUMP CAN BE FOUND BY EXAMINING THE CONSOLE OR OPERATOR LOGS.

6). THE STOP CODE ITSELF CAN BE FOUND IN CRSWHY. THE MODULE CONTAINING THE STOP CODE MAY BE FOUND BY TYPING 'S..XXX?' WHERE XXX IS THE STOP CODE. LOOK AT STOPCD.MEM IN THE SOFTWARE NOTEBOOKS OR THE CODE IN THE SOURCE LISTINGS FOR THE DESCRIPTION OF THE STOP CODE INCLUDING THE STOP CODE TYPE.

DESCRIBE THE CONDITION THAT CAUSED THE STOP CODE IE THE SPECIFIC CONDTIONAL TEST MADE INCLUDING THE DATA EXPECTED AND ACTUALLY FOUND.

EXAMINE THE DATA BASE USED TO MAKE THE DECISION TO CRASH THE MONITOR. DETERMINE WHETHER ITS VALUE IN CORE OR IN AN AC IS CORRECT VIA EXAMINING AN UNRUN MONITOR OR MONITOR LISTINGS.

7). THE CURRRENT JOB NUMBER IS STORED IN CURJOB AND .COJOB. THIS IS USEFUL FOR SETTING UP PAGING FOR THE PROPER UPMP.

8). SYMBOLIC INTERPERTATION OF THE CONTENTS OF P YIELDS INFORMATION ABOUT WHAT THE MONITOR WAS DOING WHEN THE ERROR WAS DETECTED.

P	PROCESS
NULPDL	USED BY THE MONITOR CYCLE
370510	UUO LEVEL PUSH DOWN STACK. THIS RESIDES IN THE CURRENT JOBS UPMP SO SET UP PAGING PRIOR TO REFERENCING THE STACK ITSELF.
C'N'PD1	CHANNEL 'N' PUSHDOWN STACK
ERRPDL	USED BY THE DIE ROUTINE

9). CAN BE DETERMINED BY EXAMINING THE CODE AND CORRELATING THE PC TO THE FLOW CHARTS USED IN THE MONITOR INTERNALS COURSE.

10). NOTE THE CONTENTS OF THE STACK TO TRACE THE HISTORY OF THE EVENTS LEADING TO THE CRASH.

11). NOTE THE ACTUAL CAUSE OF THE CRASH AFTER ANALYZING ALL THE INFORMATION COLLECTED UP TO THIS POINT. THIS ANALYSIS MIGHT DETERMINE THE EXACT CAUSE AND BUG FIX OR JUST SPECULATION AS TO WHAT ADDTIONAL INFORMATION NEED BE KNOWN TO COME TO A FINAL CONCLUSION.

Decsystem-10 Monitor Internals Course
Lab 4

Using the crash analysis worksheet as a guide analyze SER002.EXE as to why it crashed. You should be able to find the offending instruction. This is a 701 1091 crash. Use the same monitor specific FILDDT that was made for Lab3. remember that this crash dump was obtained by poking the monitor therefore once you find the word in error your analysis is complete.

INITIAL CRASH DUMP WORKSHEET

1. CRASH FILE_________________SERIAL #__________PROCESSOR_____

2. CRASH TIME AND DATE____

3. WHAT INTERRUPT LEVELS WERE IN PROGRESS: PISTS _______________

   (1)_____(2)_____(3)_____(4)_____(5)_____(6)_____(7)_____

4. HARDWARE STATUS AT TIME OF CRASH

   UPTSTS_________ EPTSTS_________ APR STATUS_______

   UPMP _________ EPMP__________ CURRENT AC BLOCK ________

5. WHAT CAUSED THE CRASH DUMP?
   STOP CODE_____ NON-ZERO IN 30_____ 407 RESTART_____ OTHER_____

6. IF THE STOP CODE WAS YOUR ANSWER TO 5, ANSWER THE FOLLOWING:

   STOPCODE NAME_______ STOPCODE MODULE_______

   STOPCODE DESCRIPTION ____

   ____

   STOPCODE TYPE?
   HALT_____ STOP_____ JOB_____ DEBUG_____ OTHER_____

   DATA ITEM TESTED AND TEST CONDITIONS__________________________

   ____

   EXPECTED VALUE____________________ ACTUAL VALUE_______________

7. CURRENT JOB_______ PPN_______ PROGRAM_______

8. WHAT CYCLE DETECTED OR EXPERIENCED THE ERROR?

   UUO_____ MONITOR_____ DEVICE INTERRUPT_____ OTHER_____

9. WHICH MAJOR PROCESS WITHIN THE CYCLE?

IF UUO:

PRE-DISPATCH____ COMMON I/O CODE____ SPECIFIC CODE____ POST DISPATC

IF MONITOR:

TIME ACCOUNTING___ TIMING REQUESTS___ HUNG CHECK___ REQUEUE___
SWAPPING___ SCHEDULING___

IF DEVICE INTERRPUT:

DEVICE_______ STATUS_______ RETRY______ BUFFER CHECK_____
DEVICE START/STOP________ DISMISS_____

10. LAST 10 STACK ENTRIES:

	VALUE	ROUTINE
TOP OF STACK	______________	________________________
	______________	________________________
	______________	________________________
	______________	________________________
	______________	________________________
	______________	________________________
	______________	________________________
	______________	________________________
	______________	________________________
	______________	________________________

11. ANALYSIS OF THE CAUSE OF THE CRASH.

____

____

____

____

____

____

____

INITIAL CRASH DUMP WORKSHEET SUPPLEMENT

THE WORKSHEET WAS DESIGNED FOR INSTRUCTIONAL USE IN ELEMENTARY CRASH ANALYSIS
THE PURPOSE OF THE INITIAL CRASH DUMP WORKSHEET IS TO STRUCTURE THE DATA COLLECTION PROCEDURE NECESSARY TO ANALYZE THE CAUSE OF A SPECIFIC MONITOR CRASH. THE INFORMATION TO BE RECORDED ON THIS WORKSHEET IS JUST A SMALL SUBSET OF ALL THE INFORMATION AVAILABLE IN A CRASH DUMP.

THE PURPOSE OF THIS SUPPLEMENT TO THE DUMP WORKSHEET IS TO EXPLAIN WHERE THE ITEMS IN THE WORKSHEET MAY BE FOUND IN A CRASH DUMP AS WELL AS HOW TO INTERPERT THEIR CONTENTS.

1). THE CRASH FILE NAME IS JUST THE NAME OF THE CRASH DUMP, IE SER001.EXE

THE PROCESSOR SERIAL NUMBER MAY BE FOUND IN .COASN FOR CPU0 AND .C1ASN FOR CPU1.

THE PROCESSOR TYPE MAY BE DETERMINED FROM THE SERIAL NUMBER ACCORDING TO THE FOLLOWING SERIAL NUMBER (IN DECIMAL) ASSIGNMENTS:

```
              KA < 513
     512 <    KI < 1025
    1024 <    KL < 4097
    4096 <    KS
```

DEPENDING ON PROCESSOR TYPE FILDDT SHOULD BE SET UP TO MAP ADDRESSES.

2) THE CRASH DATE AND TIME MAY BE FOUND IN LOCATIONS: LOCYER, LOCMON, LOCDAY, LOCHOR, LOCMIN, LOCSEC IN DECIMAL. THIS IS USEFUL FOR CORRELATION WITH OTHER EVENTS THAT OCCURED AT THE TIME OF THE CRASH, IE HARDWARE FAILURES ETC.

3). FOR STOP CODE CRASHES PISTS WILL CONTAIN THE RESULTS OF A CONI PI. BITS 21-27 DESCRIBE THE INTERRUPT IN PROGRESS AS DESCRIBED IN THE HARDWARE REFERENCE MANUAL SECTION 3.2.

4). THE HARDWARE STATUS MAY BE FOUND AS FOLLOWS:

```
    UPMP- UPTSTS  KL  BITS 23-35
          EUBSTS  KI  BITS 5-17
    EPMP- EPTSTS  KL  BITS 23-35
          EUBSTS  KI  BITS 23-35

CURRENT AC BLOCK- UPTSTS  KL  BITS 6-8
                  EUBSTS  KI  BITS 1-2

APR STATUS   - APRSTS
               INTERPERTATION OF THE BITS IN APRSTS INDICATE VARIOUS
               PROCESSOR ERRORS AS DESCRIBED IN THE HARDWARE REFERENCE
               MANUAL.
```

5). THE CAUSE OF THE DUMP CAN BE FOUND BY EXAMINING THE CONSOLE OR OPERATOR LOGS.

6). THE STOP CODE ITSELF CAN BE FOUND IN CRSWHY. THE MODULE CONTAINING THE STOP CODE MAY BE FOUND BY TYPING 'S..XXX?' WHERE XXX IS THE STOP CODE. LOOK AT STOPCD.MEM IN THE SOFTWARE NOTEBOOKS OR THE CODE IN THE SOURCE LISTINGS FOR THE DESCRIPTION OF THE STOP CODE INCLUDING THE STOP CODE TYPE.

DESCRIBE THE CONDITION THAT CAUSED THE STOP CODE IE THE SPECIFIC CONDTIONAL TEST MADE INCLUDING THE DATA EXPECTED AND ACTUALLY FOUND.

EXAMINE THE DATA BASE USED TO MAKE THE DECISION TO CRASH THE MONITOR. DETERMINE WHETHER ITS VALUE IN CORE OR IN AN AC IS CORRECT VIA EXAMINING AN UNRUN MONITOR OR MONITOR LISTINGS.

7). THE CURRRENT JOB NUMBER IS STORED IN CURJOB AND .COJOB. THIS IS USEFUL FOR SETTING UP PAGING FOR THE PROPER UPMP.

8). SYMBOLIC INTERPERTATION OF THE CONTENTS OF P YIELDS INFORMATION ABOUT WHAT THE MONITOR WAS DOING WHEN THE ERROR WAS DETECTED.

P	PROCESS
NULPDL	USED BY THE MONITOR CYCLE
370510	UUO LEVEL PUSH DOWN STACK. THIS RESIDES IN THE CURRENT JOBS UPMP SO SET UP PAGING PRIOR TO REFERENCING THE STACK ITSELF.
C'N'PD1	CHANNEL 'N' PUSHDOWN STACK
ERRPDL	USED BY THE DIE ROUTINE

9). CAN BE DETERMINED BY EXAMINING THE CODE AND CORRELATING THE PC TO THE FLOW CHARTS USED IN THE MONITOR INTERNALS COURSE.

10). NOTE THE CONTENTS OF THE STACK TO TRACE THE HISTORY OF THE EVENTS LEADING TO THE CRASH.

11). NOTE THE ACTUAL CAUSE OF THE CRASH AFTER ANALYZING ALL THE INFORMATION COLLECTED UP TO THIS POINT. THIS ANALYSIS MIGHT DETERMINE THE EXACT CAUSE AND BUG FIX OR JUST SPECULATION AS TO WHAT ADDTIONAL INFORMATION NEED BE KNOWN TO COME TO A FINAL CONCLUSION.

Decsystem-10 Monitor Internals Course
Lab 5

Use the crash analysis worksheet to help in analyzing SER003.EXE. This crash was obtained by exercising a bug. You should be able to determine why the machine crashed and after studying what function was being performed by the monitor you should be able to outline a general cure for the problem.

INITIAL CRASH DUMP WORKSHEET

1. CRASH FILE________________SERIAL #__________PROCESSOR_____

2. CRASH TIME AND DATE____

3. WHAT INTERRUPT LEVELS WERE IN PROGRESS: PISTS _______________

   (1)_____(2)_____(3)_____(4)_____(5)_____(6)_____(7)_____

4. HARDWARE STATUS AT TIME OF CRASH

   UPTSTS_________ EPTSTS_________ APR STATUS_______

   UPMP _________ EPMP_________ CURRENT AC BLOCK ________

5. WHAT CAUSED THE CRASH DUMP?
   STOP CODE_____ NON-ZERO IN 30_____ 407 RESTART_____ OTHER_____

6. IF THE STOP CODE WAS YOUR ANSWER TO 5, ANSWER THE FOLLOWING:

   STOPCODE NAME_______ STOPCODE MODULE_______

   STOPCODE DESCRIPTION ______________________________________

   ____

   STOPCODE TYPE?
   HALT_____ STOP_____ JOB_____ DEBUG_____ OTHER_____

   DATA ITEM TESTED AND TEST CONDITIONS_______________________

   ____

   EXPECTED VALUE___________________ ACTUAL VALUE______________

7. CURRENT JOB_______ PPN_______ PROGRAM_______

8. WHAT CYCLE DETECTED OR EXPERIENCED THE ERROR?

   UUO_____ MONITOR_____ DEVICE INTERRUPT_____ OTHER_____

9. WHICH MAJOR PROCESS WITHIN THE CYCLE?
IF UUO:
PRE-DISPATCH____ COMMON I/O CODE____ SPECIFIC CODE____ POST DISPATC

IF MONITOR:
TIME ACCOUNTING___ TIMING REQUESTS___ HUNG CHECK___ REQUEUE___
SWAPPING___ SCHEDULING___

IF DEVICE INTERRPUT:
DEVICE______ STATUS_______ RETRY______ BUFFER CHECK_____
DEVICE START/STOP________ DISMISS_____

10. LAST 10 STACK ENTRIES:

	VALUE	ROUTINE
TOP OF STACK	______________	________________________
	______________	________________________
	______________	________________________
	______________	________________________
	______________	________________________
	______________	________________________
	______________	________________________
	______________	________________________
	______________	________________________
	______________	________________________

11. ANALYSIS OF THE CAUSE OF THE CRASH.

____

____

____

____

____

____

____

INITIAL CRASH DUMP WORKSHEET SUPPLEMENT

THE WORKSHEET WAS DESIGNED FOR INSTRUCTIONAL USE IN ELEMENTARY CRASH ANALYSIS
THE PURPOSE OF THE INITIAL CRASH DUMP WORKSHEET IS TO STRUCTURE THE
DATA COLLECTION PROCEDURE NECESSARY TO ANALYZE THE CAUSE OF A SPECIFIC
MONITOR CRASH. THE INFORMATION TO BE RECORDED ON THIS WORKSHEET IS
JUST A SMALL SUBSET OF ALL THE INFORMATION AVAILABLE IN A CRASH DUMP.

THE PURPOSE OF THIS SUPPLEMENT TO THE DUMP WORKSHEET IS TO EXPLAIN
WHERE THE ITEMS IN THE WORKSHEET MAY BE FOUND IN A CRASH DUMP AS
WELL AS HOW TO INTERPERT THEIR CONTENTS.

1). THE CRASH FILE NAME IS JUST THE NAME OF THE CRASH DUMP, IE
SER001.EXE

THE PROCESSOR SERIAL NUMBER MAY BE FOUND IN .C0ASN
FOR CPU0 AND .C1ASN FOR CPU1.

THE PROCESSOR TYPE MAY BE DETERMINED FROM THE SERIAL NUMBER ACCORDING
TO THE FOLLOWING SERIAL NUMBER (IN DECIMAL) ASSIGNMENTS:

```
           KA < 513
  512 <    KI < 1025
 1024 <    KL < 4097
 4096 <    KS
```

DEPENDING ON PROCESSOR TYPE FILDDT SHOULD BE SET UP TO
MAP ADDRESSES.

2) THE CRASH DATE AND TIME MAY BE FOUND IN LOCATIONS:
LOCYER, LOCMON, LOCDAY, LOCHOR, LOCMIN, LOCSEC IN DECIMAL.
THIS IS USEFUL FOR CORRELATION WITH OTHER EVENTS THAT OCCURED
AT THE TIME OF THE CRASH, IE HARDWARE FAILURES ETC.

3). FOR STOP CODE CRASHES PISTS WILL CONTAIN THE RESULTS OF A
CONI PI. BITS 21-27 DESCRIBE THE INTERRUPT IN PROGRESS AS
DESCRIBED IN THE HARDWARE REFERENCE MANUAL SECTION 3.2.

4). THE HARDWARE STATUS MAY BE FOUND AS FOLLOWS:

```
     UPMP- UPTSTS  KL  BITS 23-35
           EUBSTS  KI  BITS 5-17
     EPMP- EPTSTS  KL  BITS 23-35
           EUBSTS  KI  BITS 23-35

CURRENT AC BLOCK- UPTSTS  KL  BITS 6-8
                  EUBSTS  KI  BITS 1-2
```

APR STATUS - APRSTS
INTERPERTATION OF THE BITS IN APRSTS INDICATE VARIOUS
PROCESSOR ERRORS AS DESCRIBED IN THE HARDWARE REFERENCE
MANUAL.

5). THE CAUSE OF THE DUMP CAN BE FOUND BY EXAMINING THE CONSOLE OR OPERATOR
LOGS.

6). THE STOP CODE ITSELF CAN BE FOUND IN CRSWHY. THE MODULE CONTAINING THE STOP CODE MAY BE FOUND BY TYPING 'S..XXX?' WHERE XXX IS THE STOP CODE. LOOK AT STOPCD.MEM IN THE SOFTWARE NOTEBOOKS OR THE CODE IN THE SOURCE LISTINGS FOR THE DESCRIPTION OF THE STOP CODE INCLUDING THE STOP CODE TYPE.

DESCRIBE THE CONDITION THAT CAUSED THE STOP CODE IE THE SPECIFIC CONDTIONAL TEST MADE INCLUDING THE DATA EXPECTED AND ACTUALLY FOUND.

EXAMINE THE DATA BASE USED TO MAKE THE DECISION TO CRASH THE MONITOR. DETERMINE WHETHER ITS VALUE IN CORE OR IN AN AC IS CORRECT VIA EXAMINING AN UNRUN MONITOR OR MONITOR LISTINGS.

7). THE CURRRENT JOB NUMBER IS STORED IN CURJOB AND .COJOB. THIS IS USEFUL FOR SETTING UP PAGING FOR THE PROPER UPMP.

8). SYMBOLIC INTERPERTATION OF THE CONTENTS OF P YIELDS INFORMATION ABOUT WHAT THE MONITOR WAS DOING WHEN THE ERROR WAS DETECTED.

P	PROCESS
NULPDL	USED BY THE MONITOR CYCLE
370510	UUO LEVEL PUSH DOWN STACK. THIS RESIDES IN THE CURRENT JOBS UPMP SO SET UP PAGING PRIOR TO REFERENCING THE STACK ITSELF.
C'N'PD1	CHANNEL 'N' PUSHDOWN STACK
ERRPDL	USED BY THE DIE ROUTINE

9). CAN BE DETERMINED BY EXAMINING THE CODE AND CORRELATING THE PC TO THE FLOW CHARTS USED IN THE MONITOR INTERNALS COURSE.

10). NOTE THE CONTENTS OF THE STACK TO TRACE THE HISTORY OF THE EVENTS LEADING TO THE CRASH.

11). NOTE THE ACTUAL CAUSE OF THE CRASH AFTER ANALYZING ALL THE INFORMATION COLLECTED UP TO THIS POINT. THIS ANALYSIS MIGHT DETERMINE THE EXACT CAUSE AND BUG FIX OR JUST SPECULATION AS TO WHAT ADDTIONAL INFORMATION NEED BE KNOWN TO COME TO A FINAL CONCLUSION.

DECsystem-10 MONITOR INTERNALS COURSE

FILDDT Lab session # 6 : examining the running monitor

This lab session requires you to examine portions of the data base of the currently running monitor, particularly the FILSER data base.

To begin, simply type .R FDSYS, and when FILDDT asks "File:", the student should respond with /M. Thereafter, regular DDT commands apply.

Use FILDDT and the Monitor Table Descriptions to answer all the following questions, (note: answers should consist of the table name and word label plus the data)

I. FILSER data base

A. The [75,3] job is reading or writing a file.

1. Find the PPB and follow the NMB and UFB linkages from it.

2. Find the UFB. What is the disk address of the UFD?

3. Find the NMB and from it find the access table for an active file. (Note: You may encounter many NMB'S but only one will be active. Inactive NMB'S usually dont point to ACC blocks, they point back on themselves.)

4. What does the access table think is being done to the file?

B. FIND THE DDB. (That can be tough if the JDA is not in core.) HINT: see PDB,(.PDDVL word) program is using a logical name for Disk ...

1. What mode is being used to read or write the file?

2. What is this DDB's logical device name ?

3. What relative block number is being accessed ?

4. Describe the disk allocation of the file from its group pointer(s).

II. Disk file structures, storage allocation, etc.

A. How many file structures are on the system?

1. Their names?

2. Number of units in each structure and physical unit name of each?

3. Describe the active swapping list. How much swap space on each unit? How much is free on each unit?

B. SAT blocks --

1. How many total SAT blocks for dskb:? How many in core?

2. How much space left in each SAT block?