

Monitor TOPS-20 Internals

DIGITAL

Copyright (c) 1980 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 that may appear in this document.

The software described in this document is furnished under a license and may only be used or copied 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 or its affiliated companies.

The following are trademarks of Digital Equipment Corporation, Maynard, Massachusetts:

COMPUTER LABS	COMTEX	DBMS-1C
DBMS-11	DBMS-20	DDT
DEC	DECCOMM	DECsystem-10
DECSYSTEM-20	DECTape	DECUS
DIROL	DIGITAL	EDUSYSTEM
FLIPCHIP	FOCAL	INDAC
LAB-8	MASSBUS	OMNIBUS
OS/8	PDP	PHA
RSTS	RSX	TYPESET-8
TYPESET-10	TYPESET-11	TYPESET-20
UNIBUS	DECSYSTEM-2020	

TABLE OF CONTENTS

Monitor Program Logic Manual

Chapter 1	Extended Addressing in TOPS-20
2	Bias Control
3	Class Scheduling
4	Execute-Only
5	Monitor Address Space
6	Monitor Modules
7	Watch
8	Working Set Swapping
9	System Debugging and Crash Analysis
Index	
10	1-23 DDT41
11	TOPS-20 Coding Conventions

Monitor Program Logic Manual

This reflects version 3 of TOPS-20.

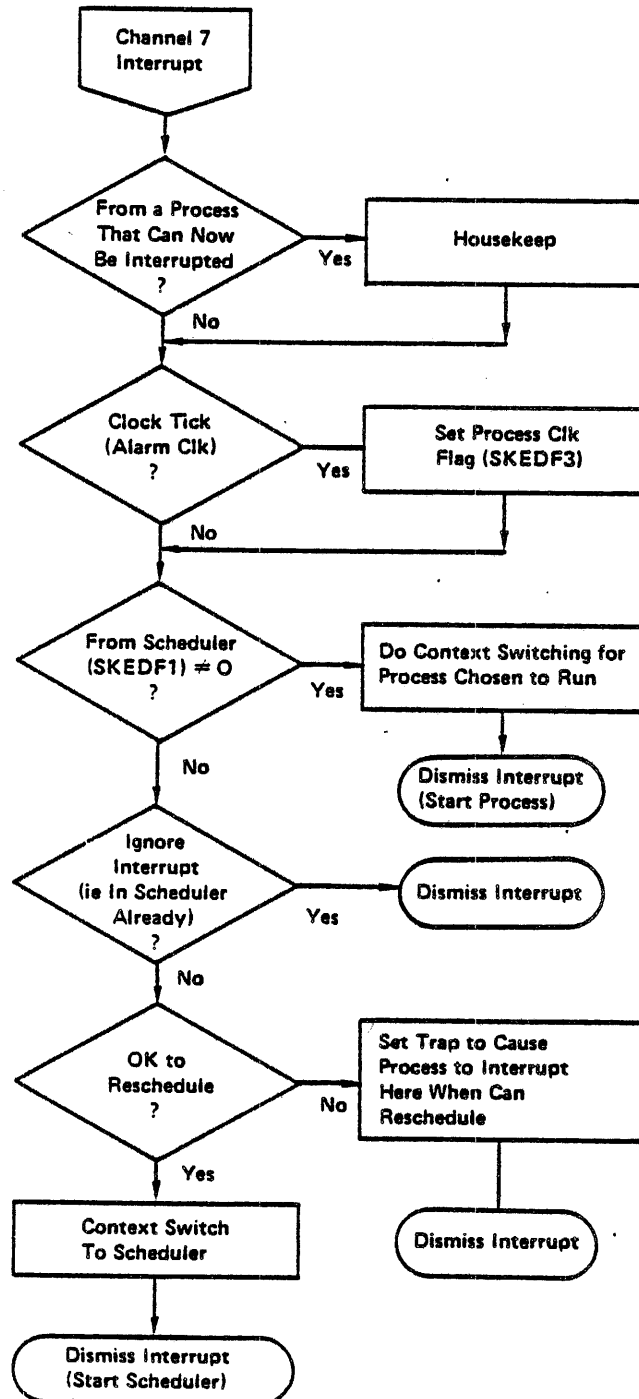
DECSYSTEM-20 MONITOR FLOWCHARTS

- I. Scheduler
- II. Page Fault Handling
- III. JSYS Calls - Device Independent Level
- IV. JSYS Calls - Disk Dependent Level
- V. JSYS Calls - Magtape Dependent Level
- VI. Requesting DSK/MTA I/O & Interrupt Handling
- VII. JSYS Calls - TTY Dependent Level
- VIII. Requesting TTY I/O & Interrupt Handling

SCHEDULER FLOWCHARTS

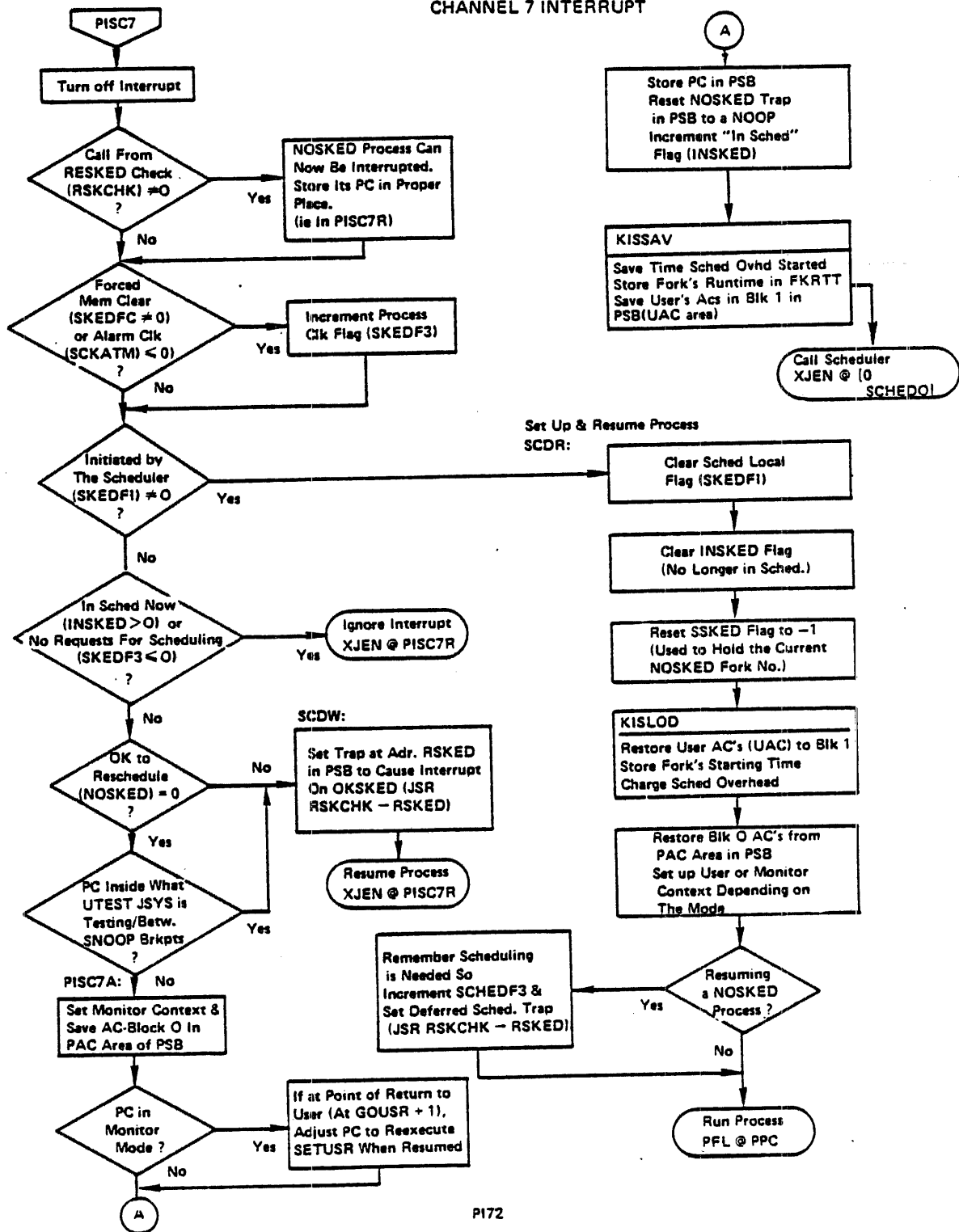
Channel 7 Interrupt - Context Switching Overview	PI71
PISC7 - Detailed Context Switching	PI72
SCHEDO - Process Controller	SCH1
UCLOCK - Process and System Accounting	SCH2
SKCLK - Update Clocks	SCH2
TCLKS - Test Clocks & Perform Action on Timeout	SCH2
SCDRQ1 - Process Requests in Scheduler's Queue	SCH9
JOBSRT - Job Startup	SCH9
SKDJOB - Select Process to Run	SCH3
GCCOR - Global Garbage Collect	SCH7
TSTBAL - Check if Balance Set Needs Adjustment	SCH5
AJBALS - Adjust Balance Set	SCH5

CHANNEL 7 INTERRUPT AN OVERVIEW

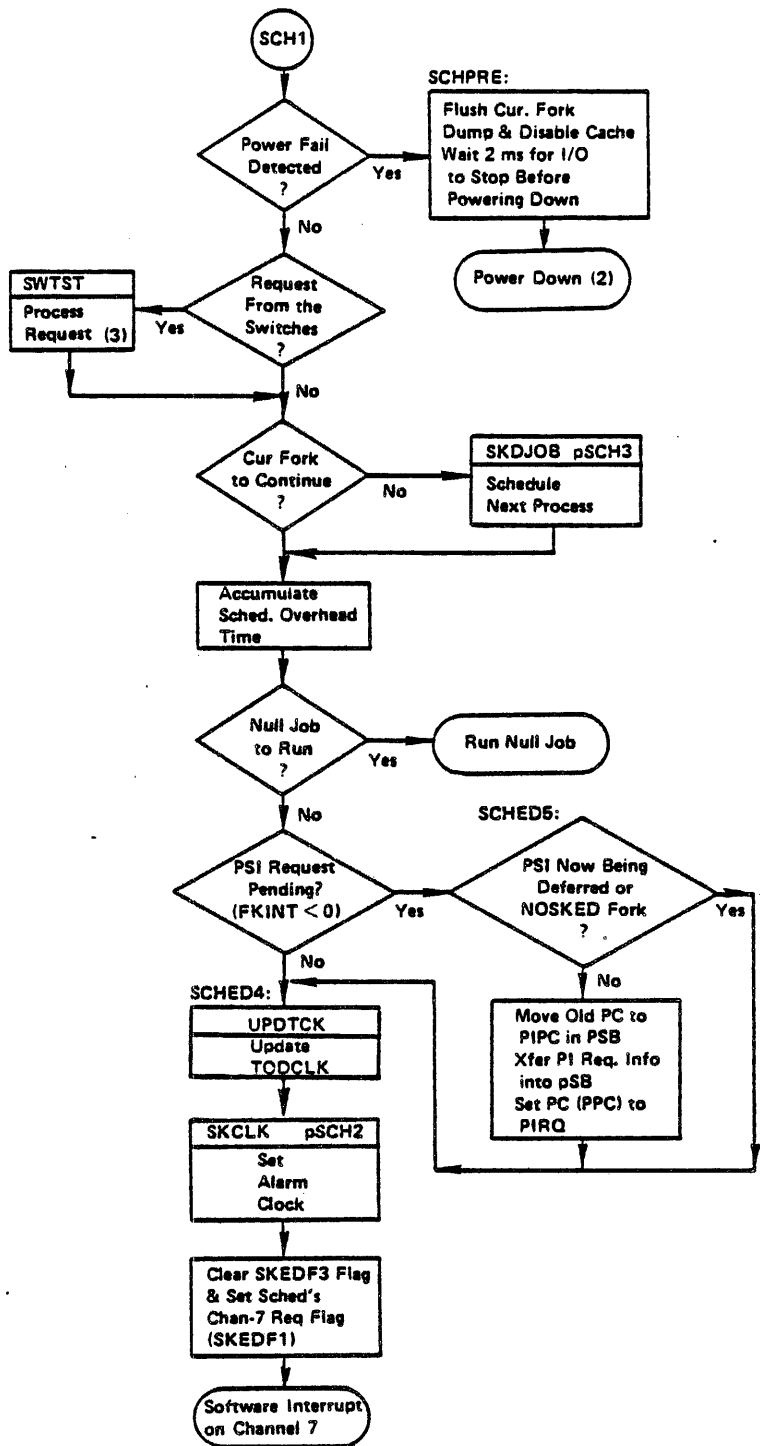
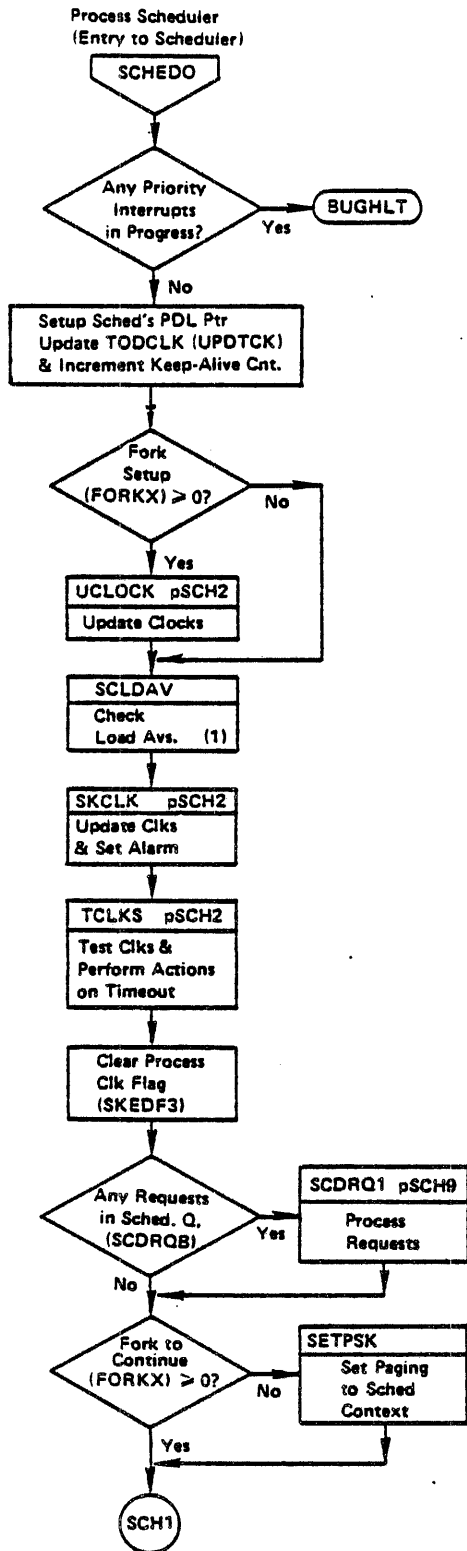


PI71

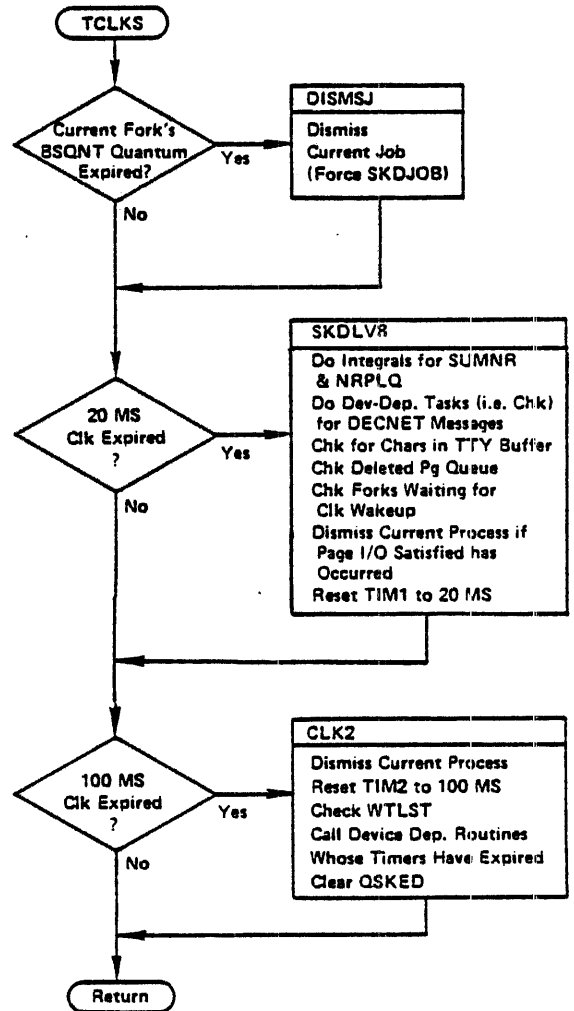
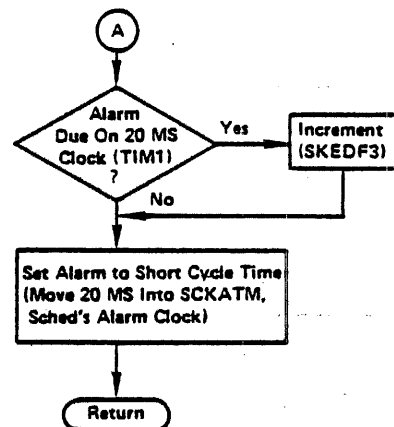
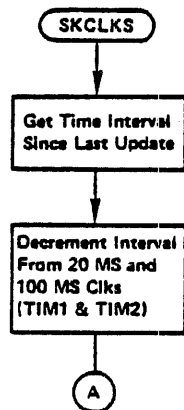
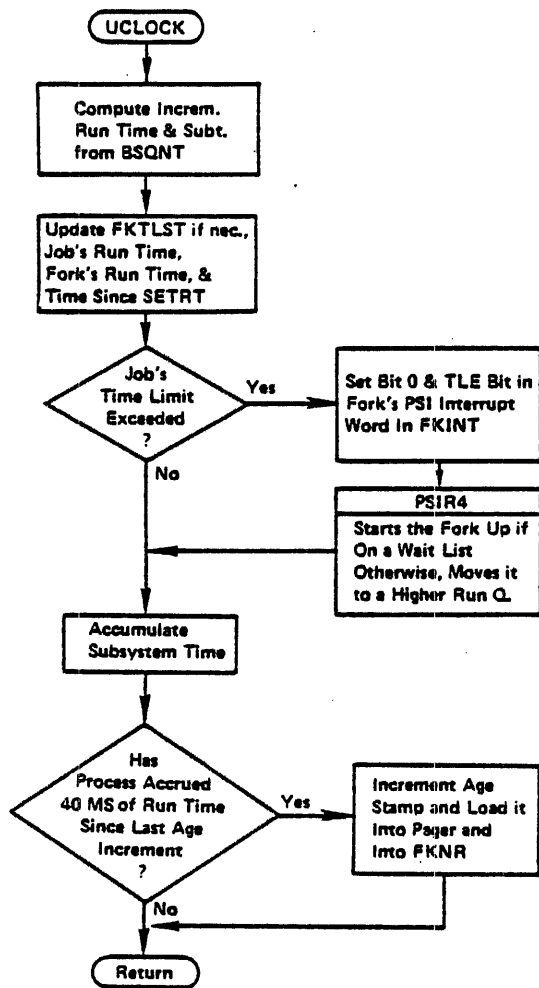
CHANNEL 7 INTERRUPT



P172

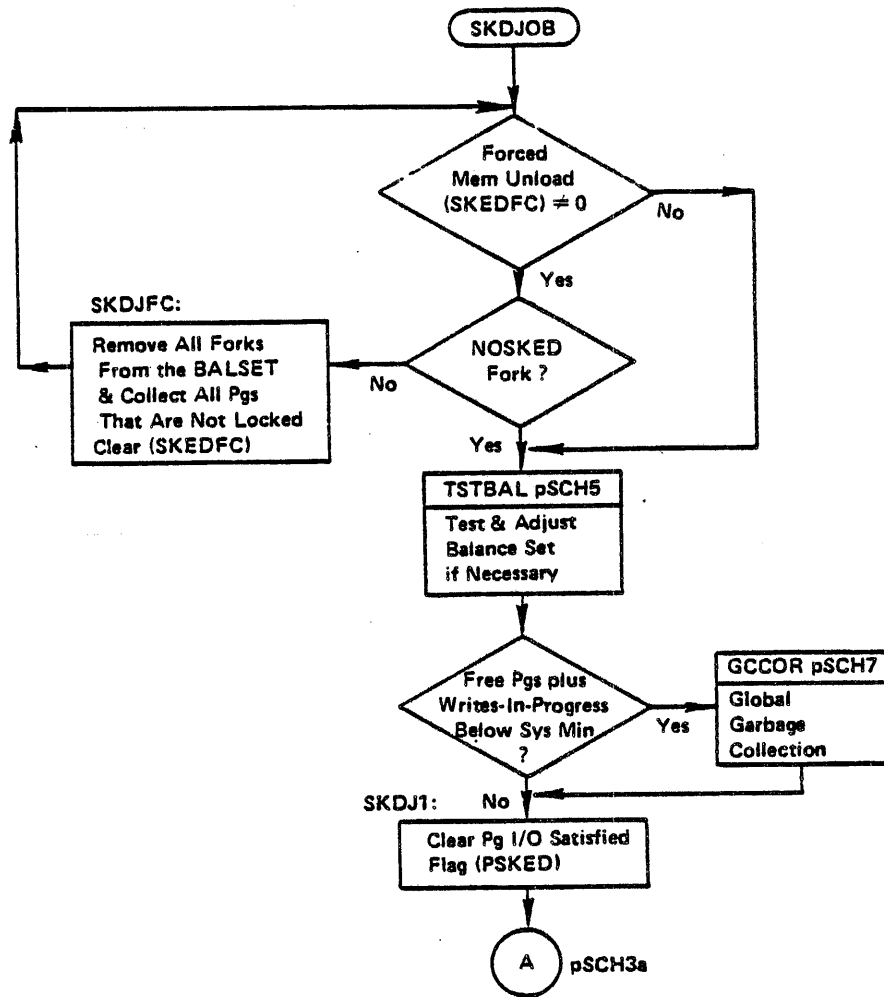


SCH1

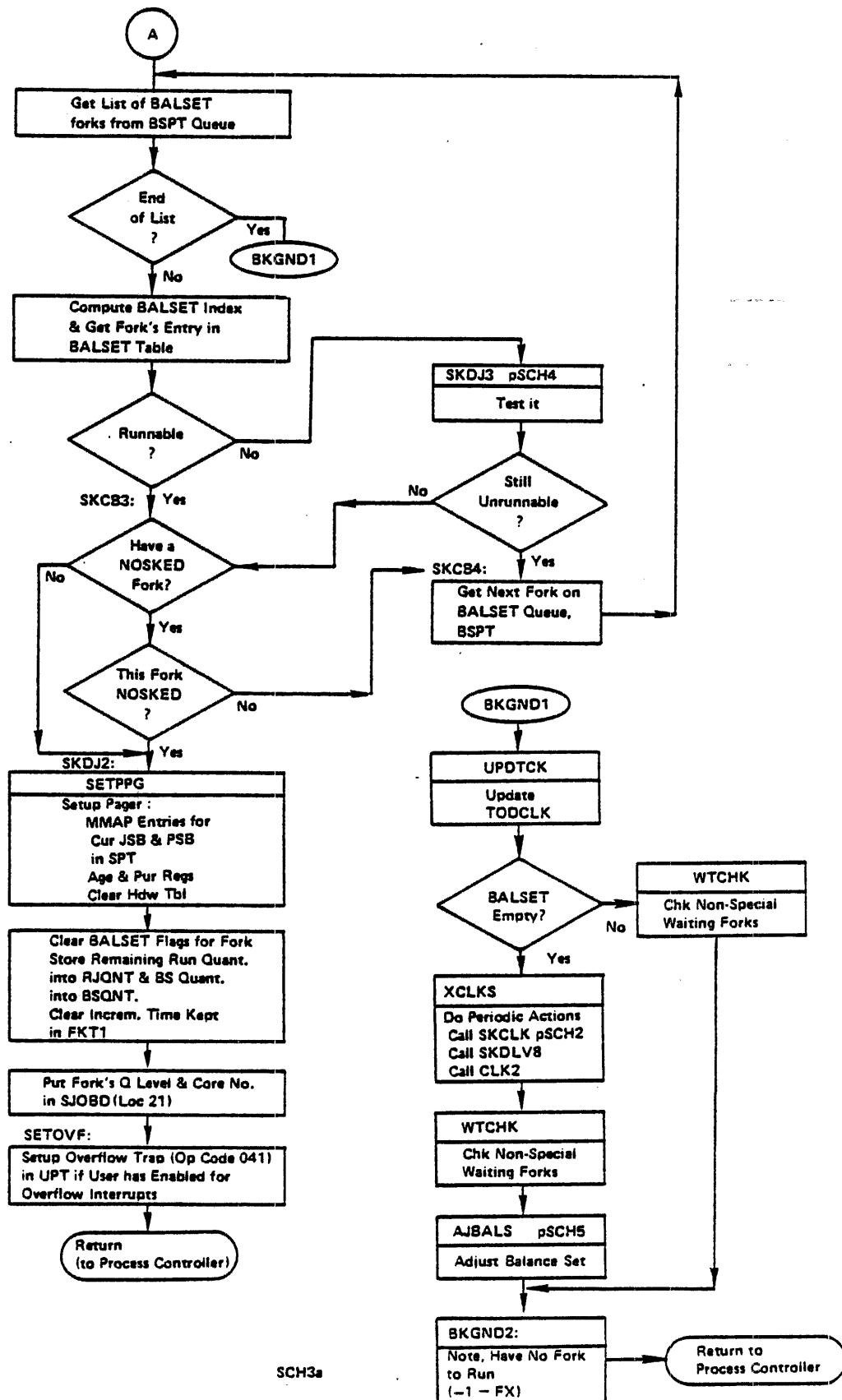


SCH2

Balance Set Scheduler
Called to Select Process to Run

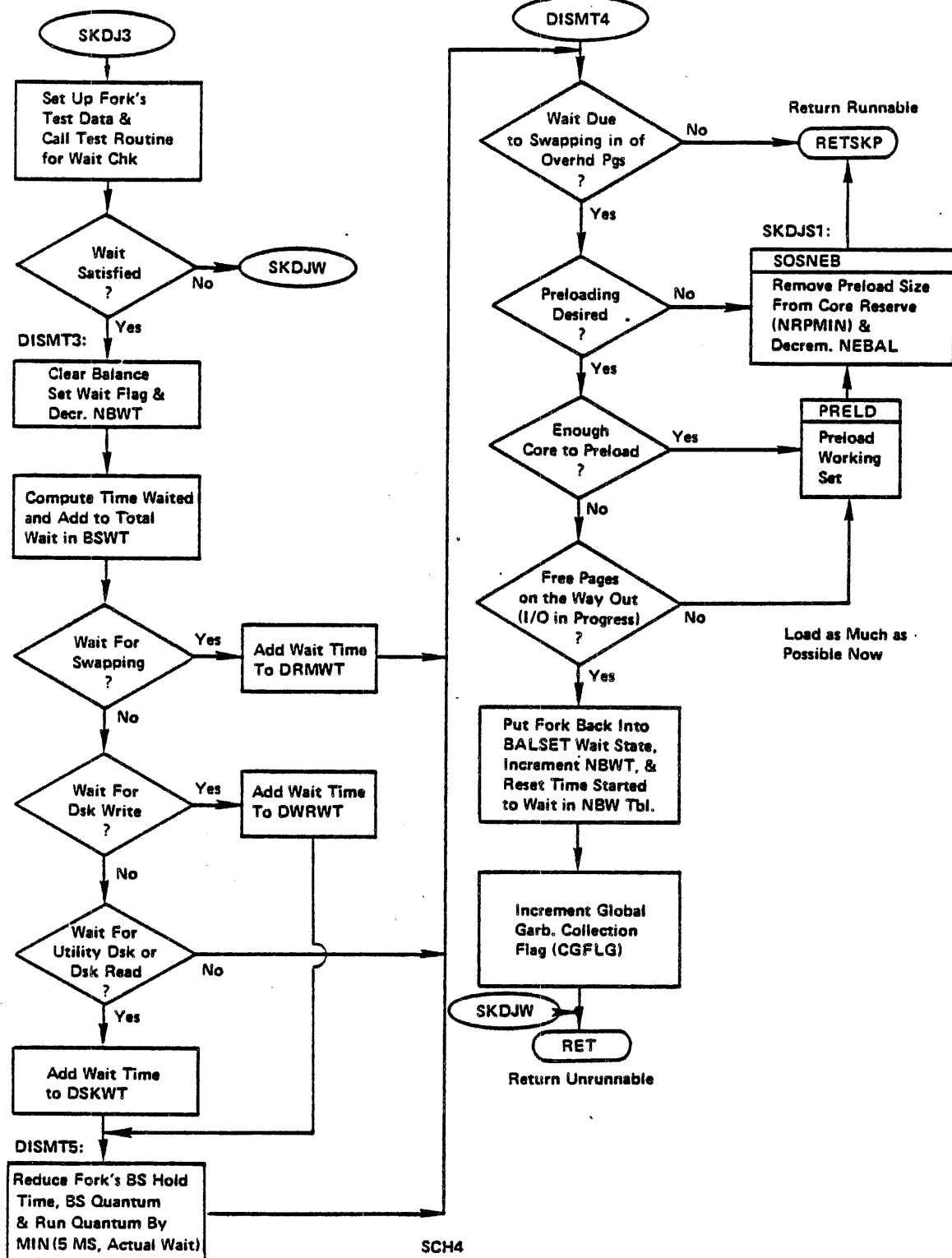


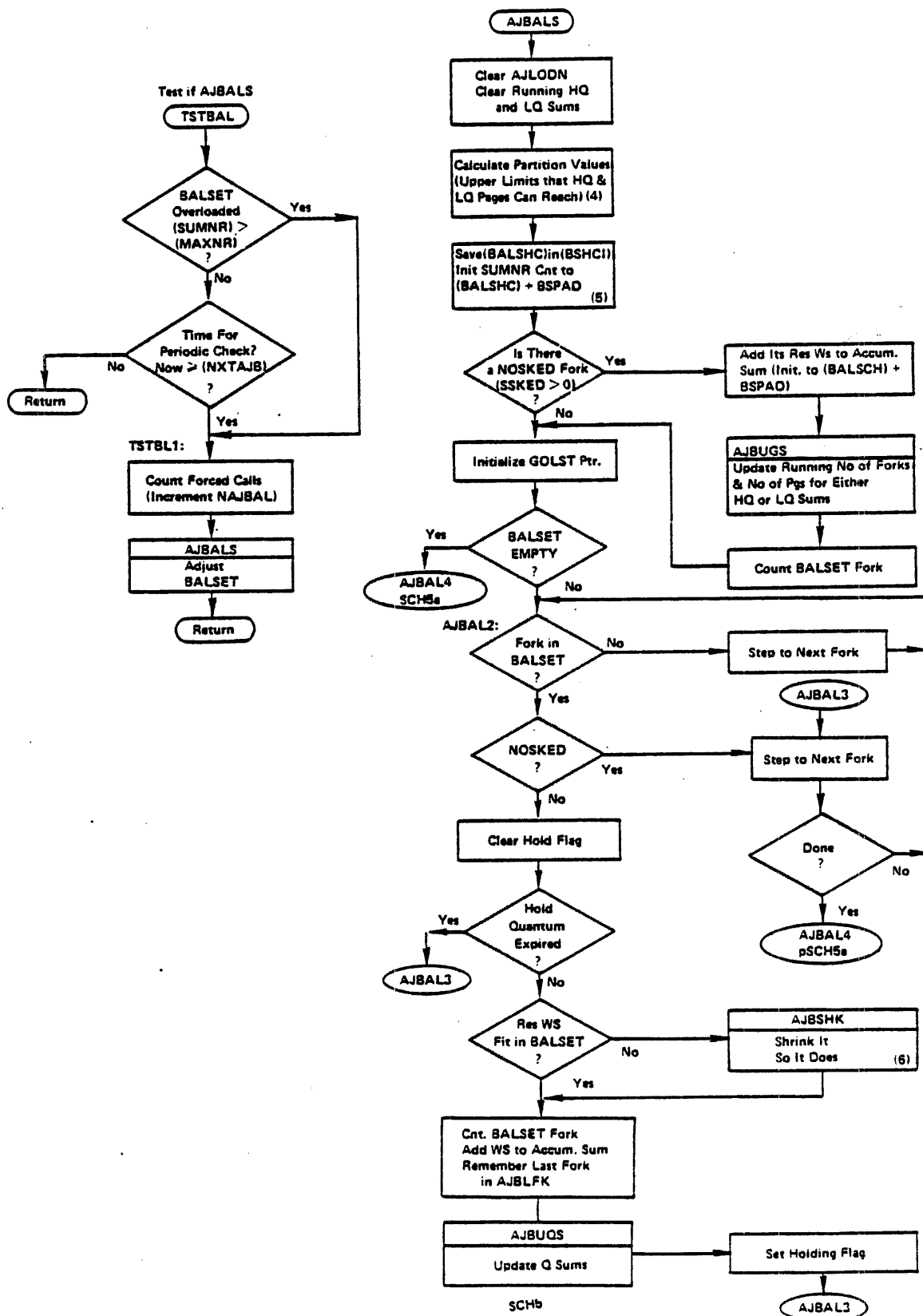
SCH3

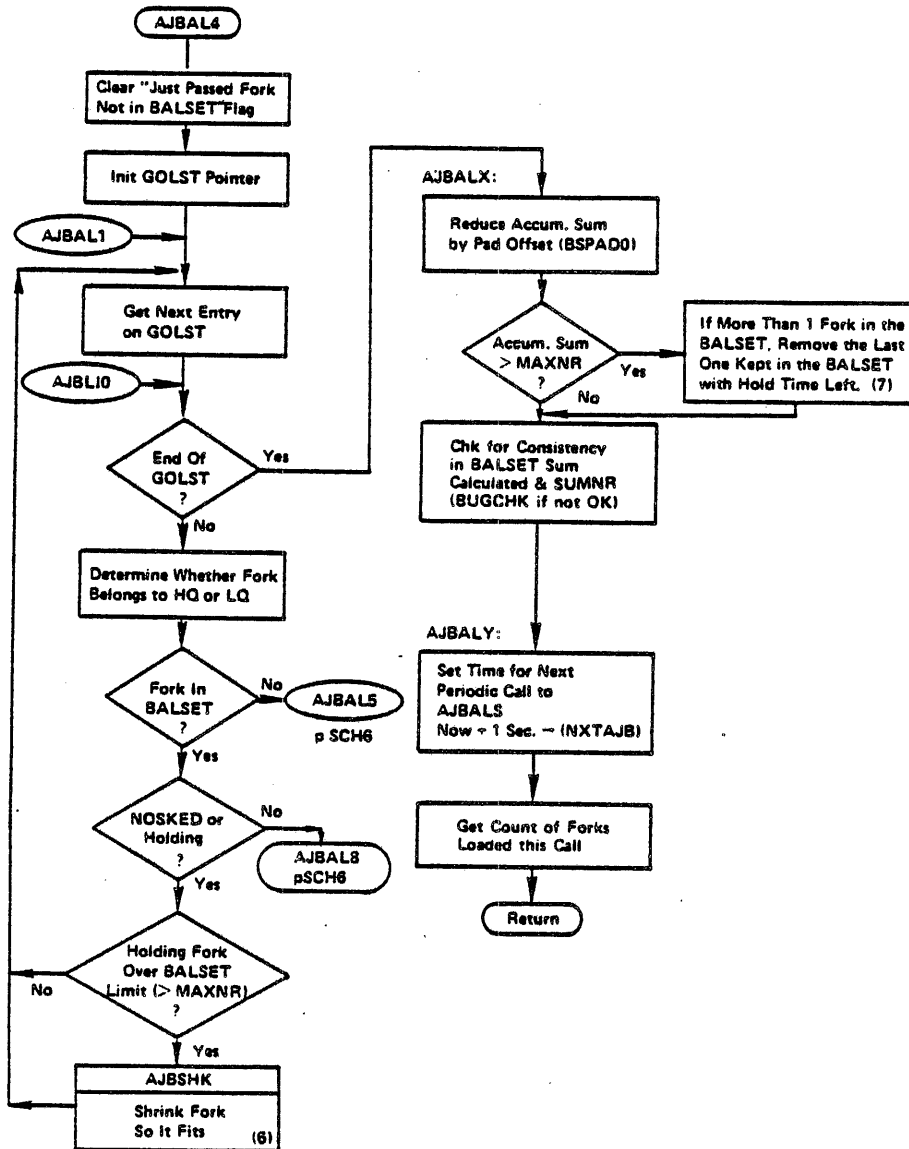


SCH3a

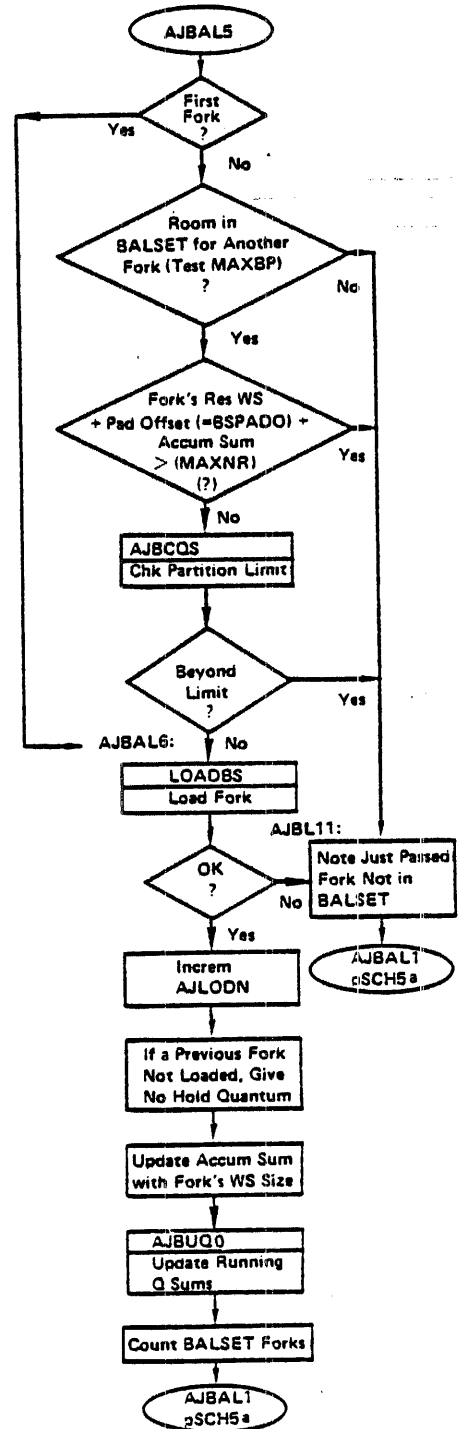
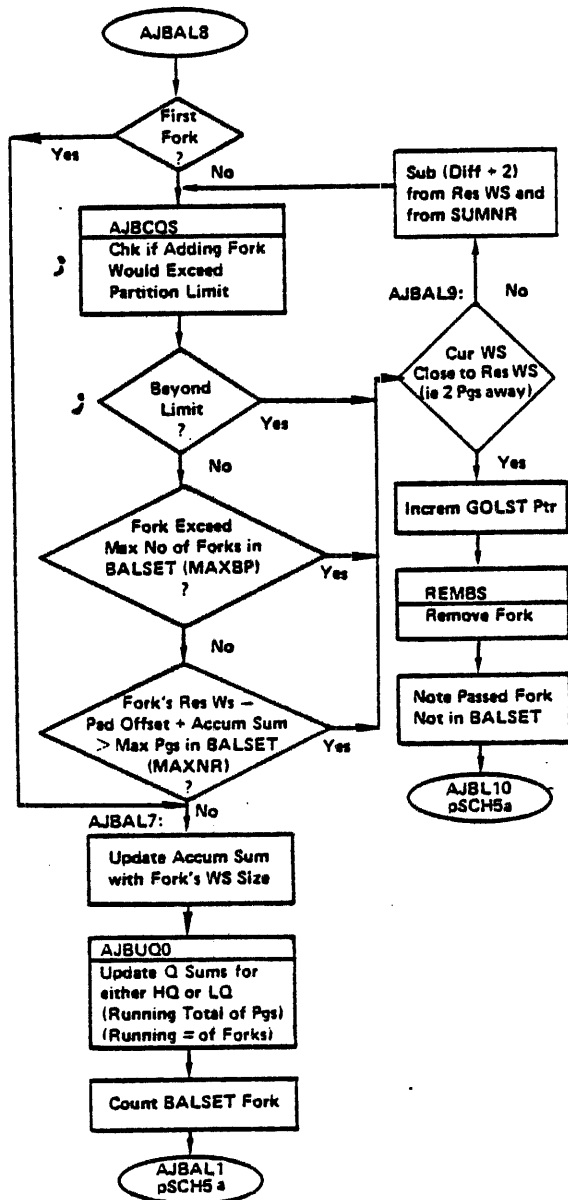
Test Waiting Balset Forks



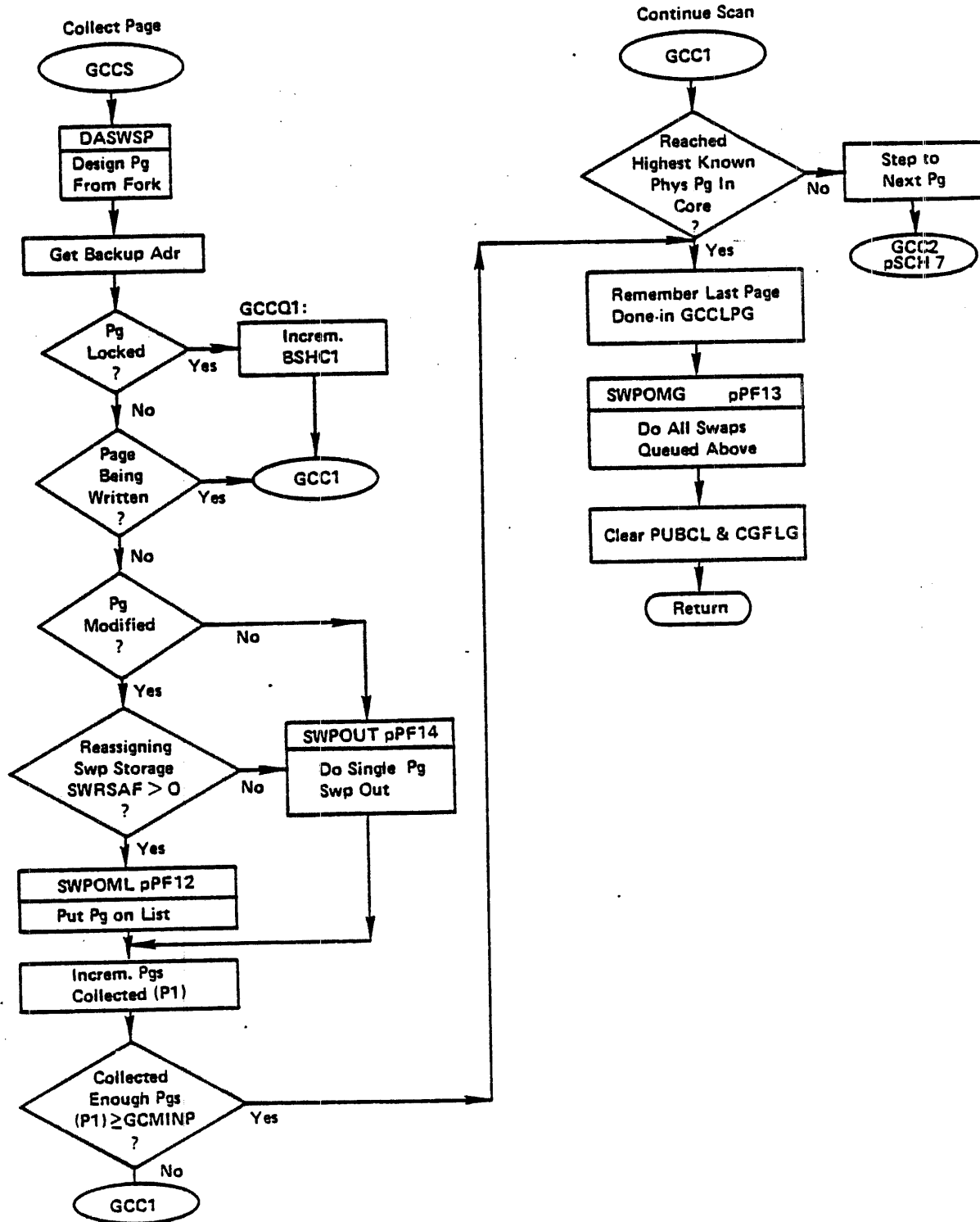




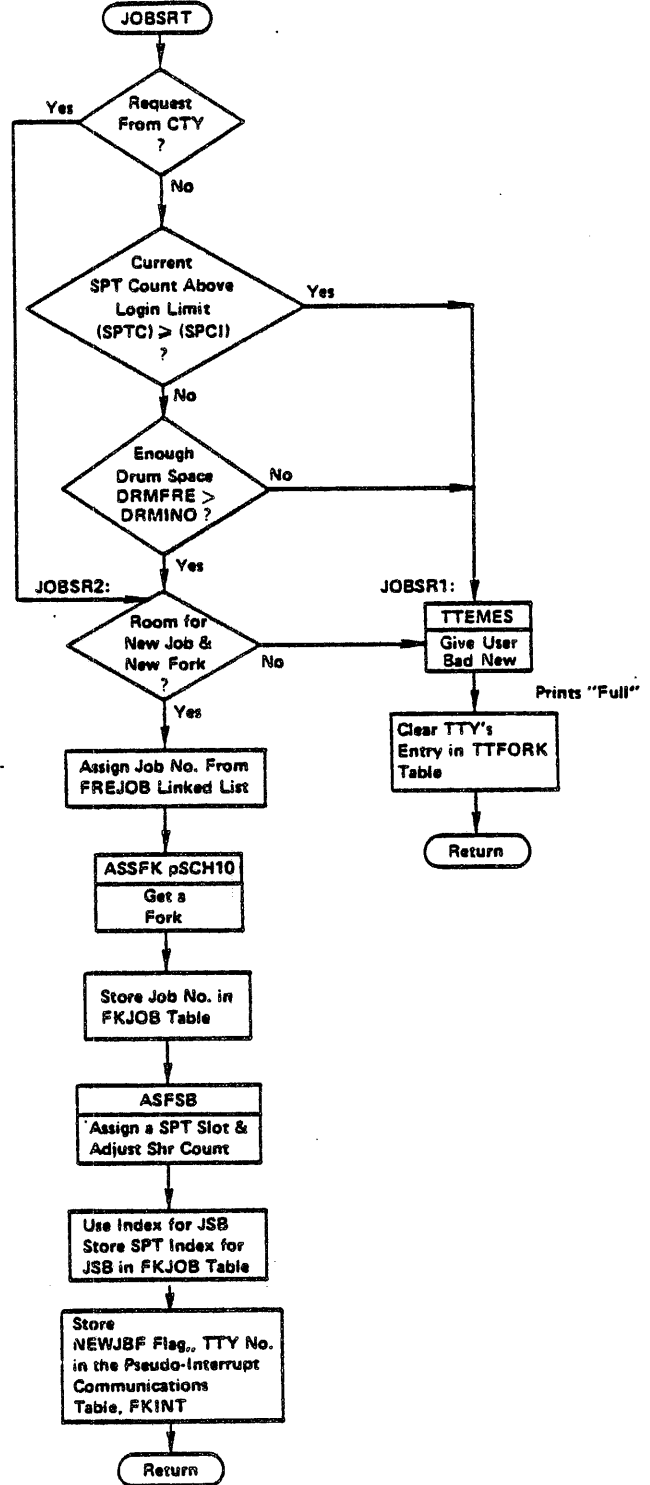
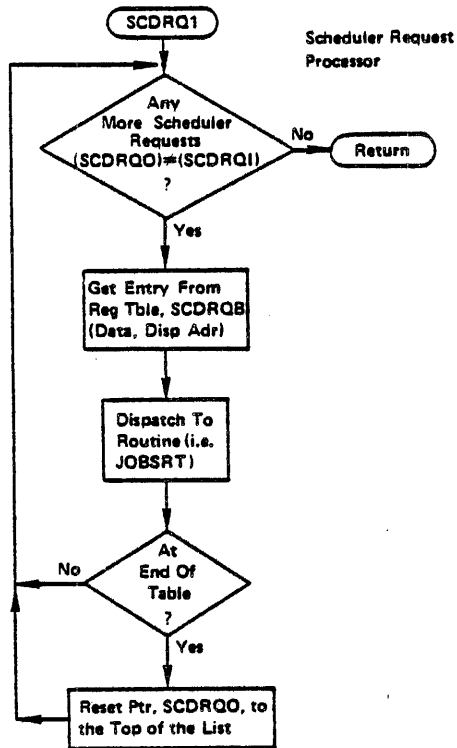
SCH5a



SCH6

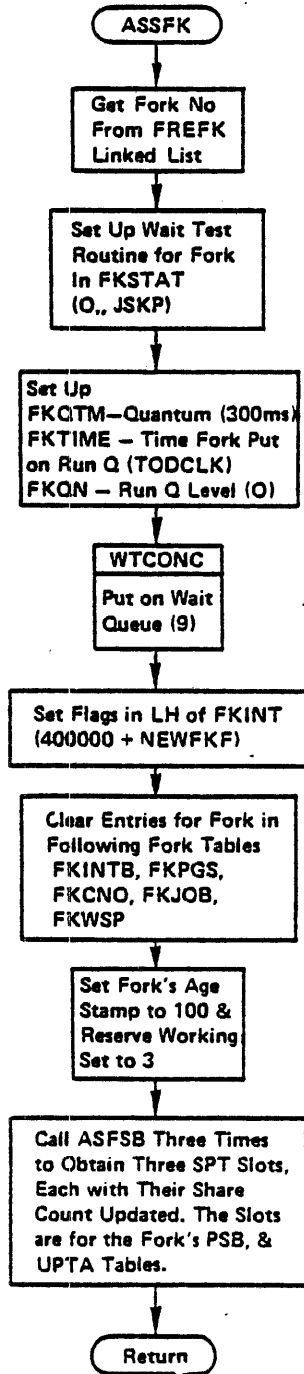


SCH8



SCH9

Assign Fork Slot



SCHIO

Scheduler Comments

SCHEDO:

- (1) Running averages, exponentially weighed over intervals of 1, 5, and 15 minutes, are maintained for the number of runnable processes overall, as well as for those in High Run Queues and those in the Low Run Queues.
- (2) Final phase of powerdown seq. clears the priority interrupt system and causes the system to loop in the ACs until power actually vanishes. If the power fail interrupt was spurious, the loop will time out after a few seconds and the system will be continued at address SYSRST.
- (3) A very limited set of central functions for debugging purposes has been built into the Scheduler. To invoke a function, the appropriate bit or bits are set into loc 20 (SCTLW) via MDDT. The word is scanned from left to right (JFFO); the first bit found set on the scan selects the function.

Bit 0 Causes the scheduler to dismiss the current process and to stop timesharing. Useful to effect a clean manual transfer to Exec-mode DDT. System may be resumed at SCHEDO if no IOB reset is done.

Bit 1 Causes job specified by (20)_{RH} to be run exclusively.

Bit 2 Forces running of Job 0 back-up function before halting the system.

If loc 30 (SHLTW) is set not equal to 0, the system will crash. (Same as setting bit 2 of SCTLW word.)

AJBALS

- (4) Upper Limit for LQ=MAXNR-MIN [Max HQ Reserve, HQ Load Avg.* (16)]
 Upper Limit for HQ=MAXNR-MIN [Max LQ Reserve, No. of LQ forks * (32)]
- (5) SUMNR reflects the number of timesharing pages in use. Its value after AJBALS equals the number of pages reserved for balance set members plus BALSHC (the number of pages shared, but not owned, by balance set members plus the number of locked pages).

BSPAD reflects the number of pages set aside for balance set members as their working set reserves grow. The real value of BSPAD is offset by a factor of BSPADO. When forks are trying to stay in the balance set, the adjustment algorithm allows the pad offset to be subtracted from the accumulated sum before it checks if the fork can fit.

$$\text{i.e.,} \quad \left(\text{BSPAD} + \sum_{i=1}^n \text{Res. WS}_i \right) - \text{BSPADO} + \text{Res. WS}_{n+1} \geq \text{MAXNR}$$

The adjustment algorithm does the opposite (i.e., adds the BSPADO factor) for forks trying to get into the balance set. The overall affect of this is to ensure (as much as possible) a certain number of pages be available for balance set forks.

- (6) The shrink algorithm shrinks the fork's reserve working set by:
 $\text{MIN} [\text{Reserve WS} - \text{Current WS}, \text{Accum. Sum} + \text{Fork's Res WS} - \text{MAXNR}]$

Notice that the fork's reserve working set will not be reduced below its current working set.

- (7) This is the rare case of forks, with hold-time left, expanding. The lowest priority one is removed. If there is only one fork in the balance set, it is not removed. (Note: it is possible for one fork to be greater than MAXNR due to the BALSHC count changing).

GCCOR

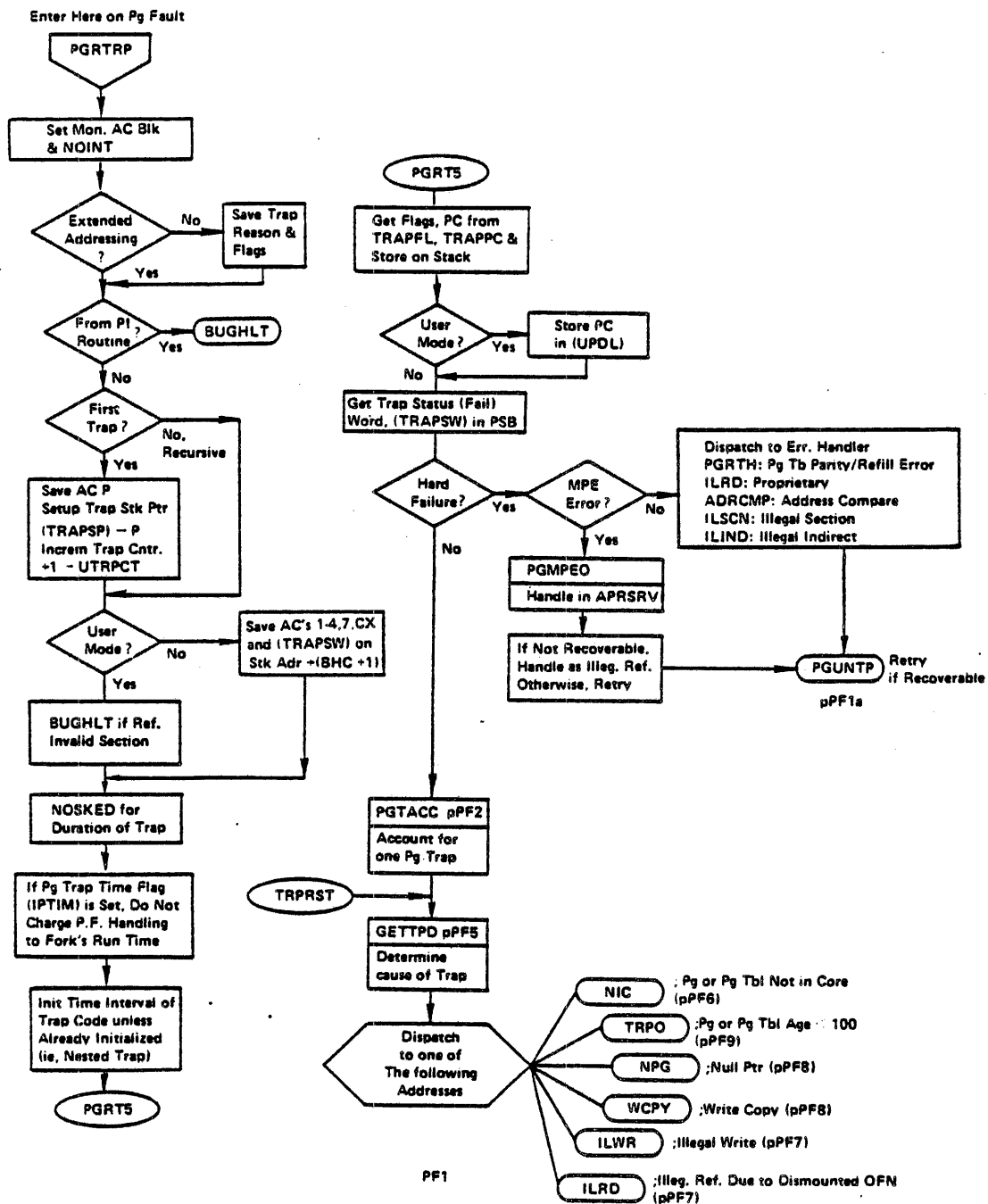
- (2) If it is a forced clear, then GCMINO is made very large so all of core will be collected. However, its usual value is much lower. (Currently 64 decimal).

ASSFK

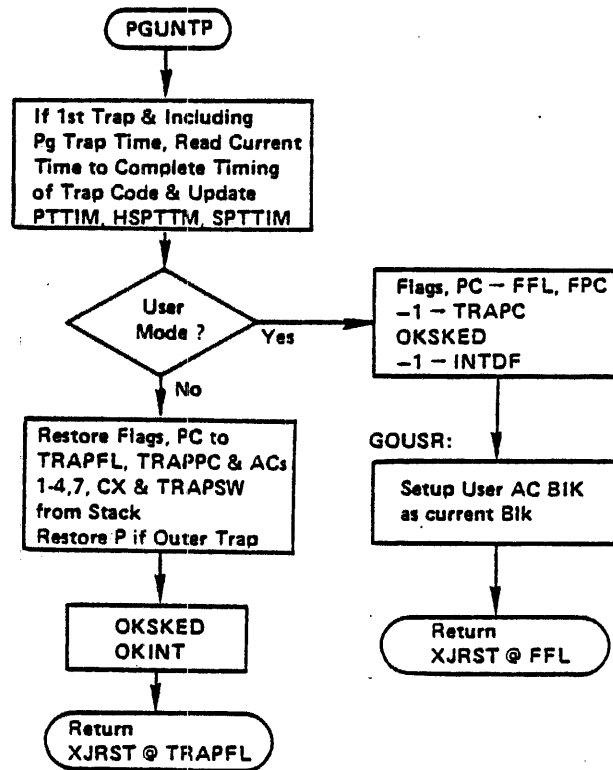
- (9) The fork is actually placed on the GOLST at this time. WTCNC, after putting a fork on WTLST, checks if the wait condition is satisfied. The test routine, JSKP, gives a skip return indicating that the wait is satisfied. This causes UNBLK1 to be called which in turn calls SCHEDJ to unblock the fork and to requeue it from the WTLST to the GOLST.

PAGE FAULT HANDLING FLOWCHARTS

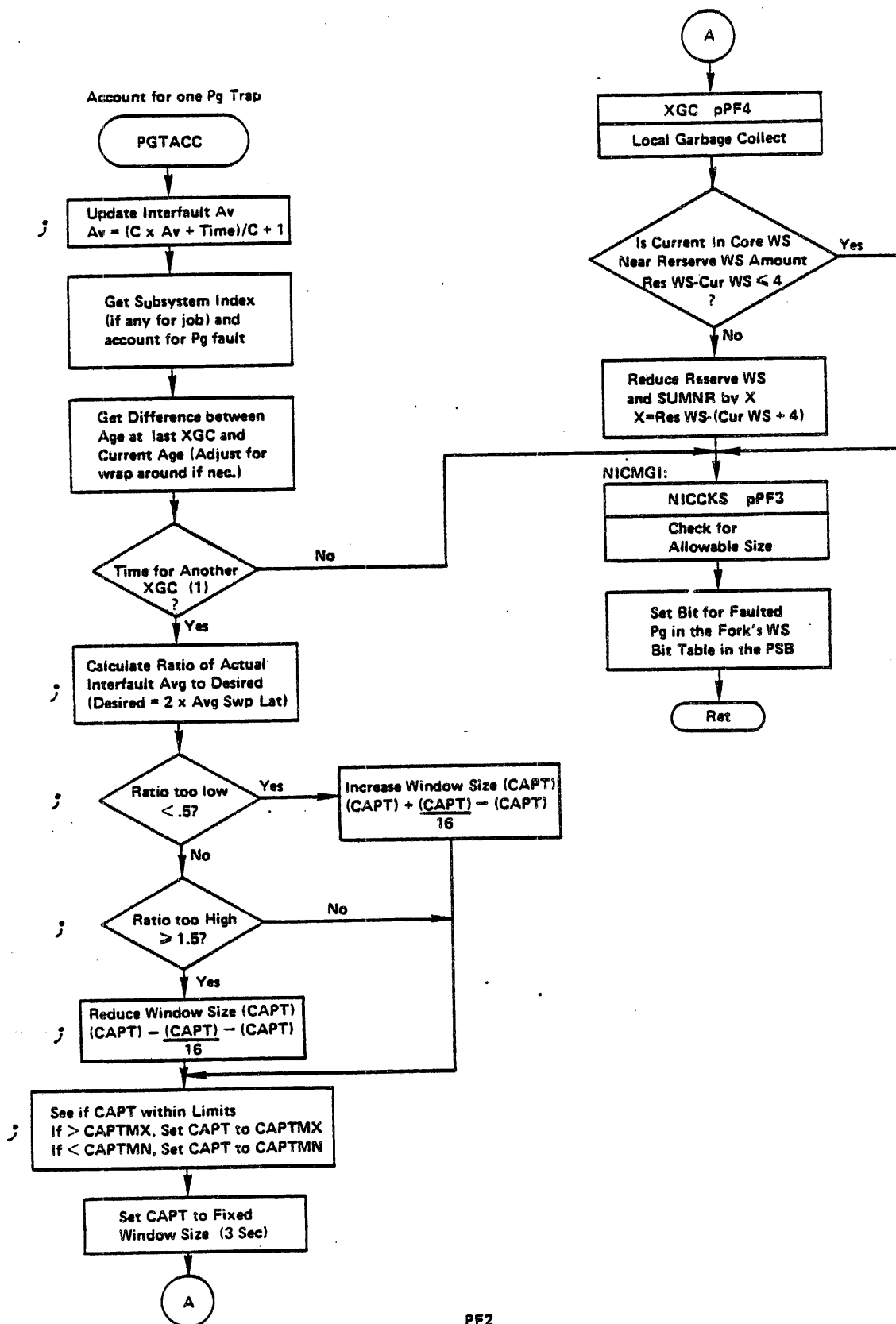
PGRTRP -	Performs the Principal Accounting, Analysis, and Resolution of Page Faults	PF1
PGTACC -	Accounts for Page Traps	PF2
XGC -	Local Garbage Collection	PF4
	SWPOUT - Swapping Out a Page	PF14
NICCKS -	Check In-Core Size Limits	PF3
GETTPD -	Determine Cause of Trap	PF5
NIC -	Not in Core Trap	PF6
	SWPINW - Swap In and Wait	PF10
	SWPIN - Swap In a Page	PF11
WCPY -	Write Copy Trap	PF8
ILRD -	Illegal Read Trap	PF7
ILWR -	Illegal Write Trap	PF7
TRPO -	Age <100 Trap	PF9

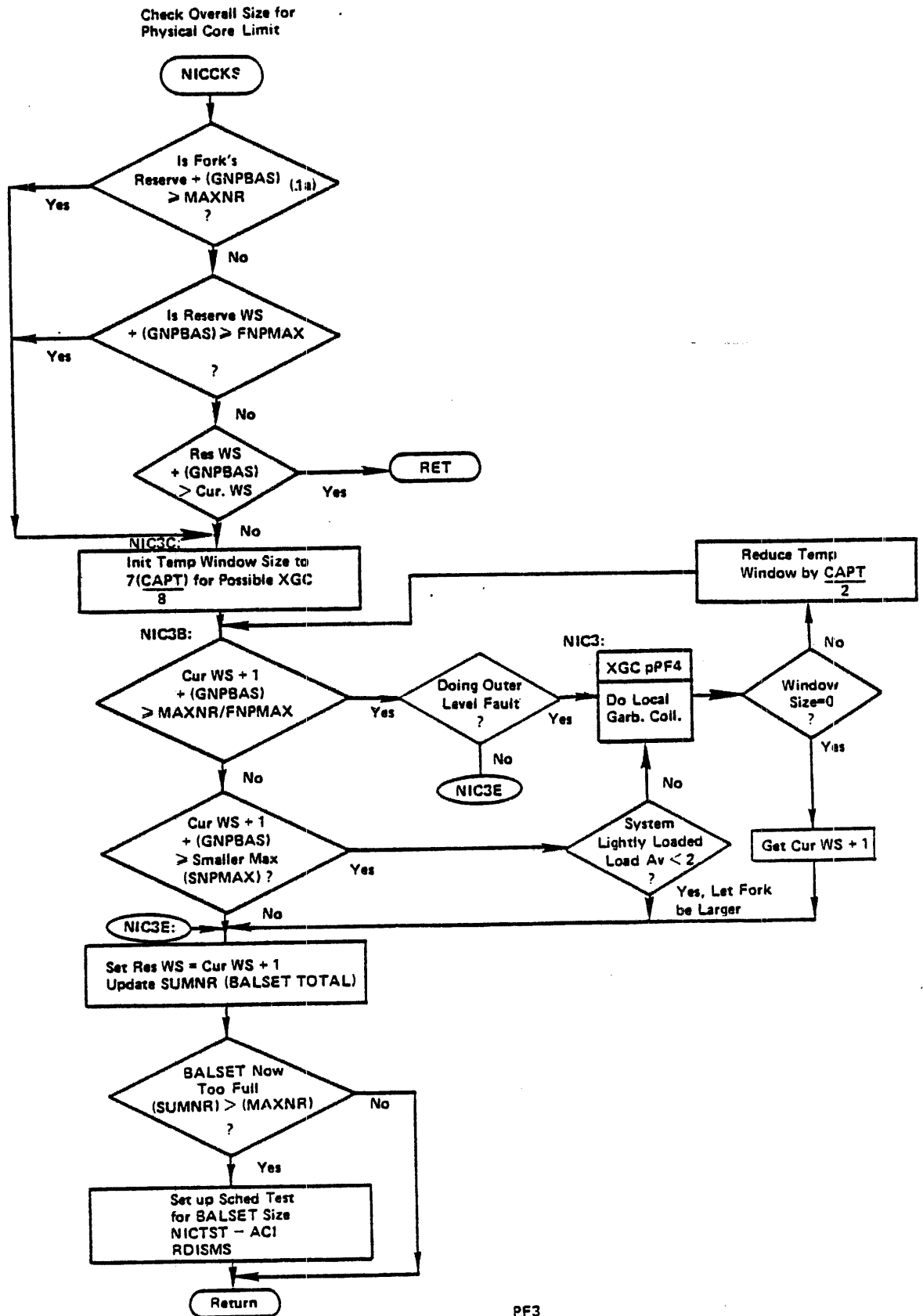


Page OK to Ref.



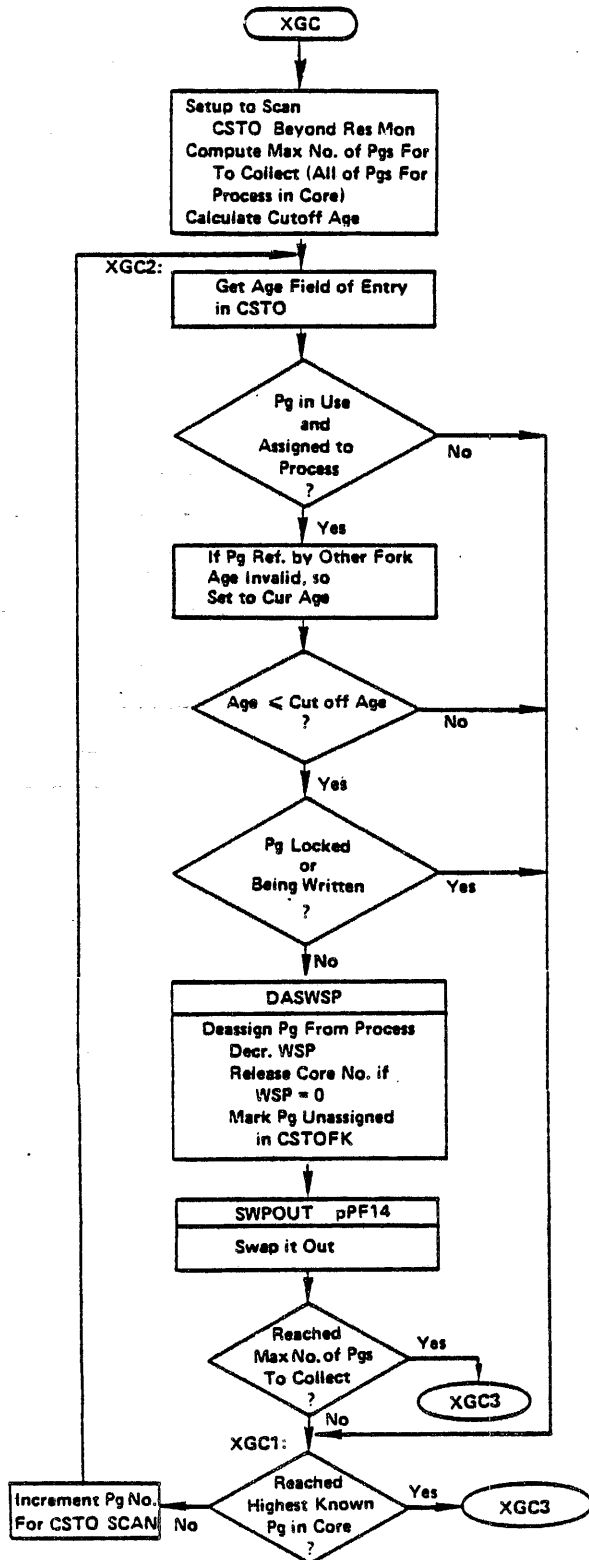
PF1a



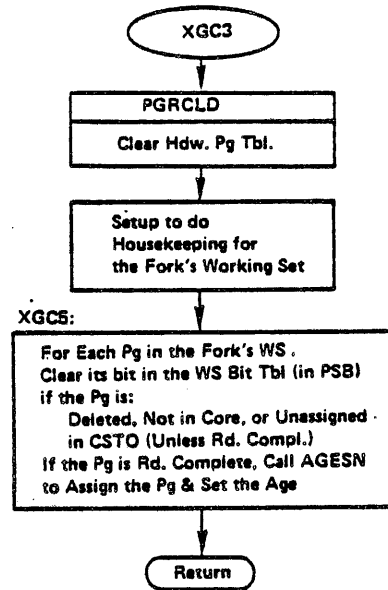


PF3

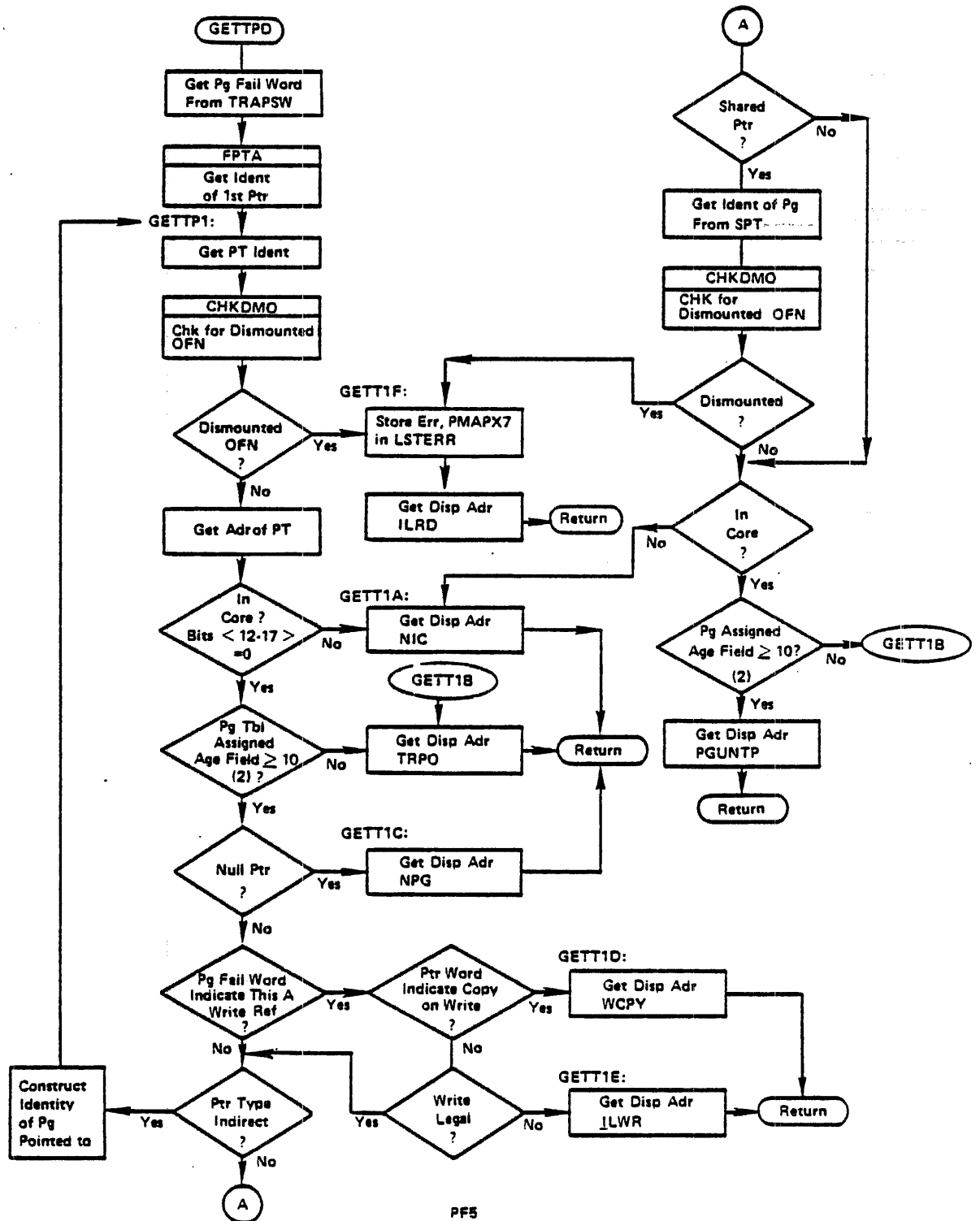
Local Garbage Collection



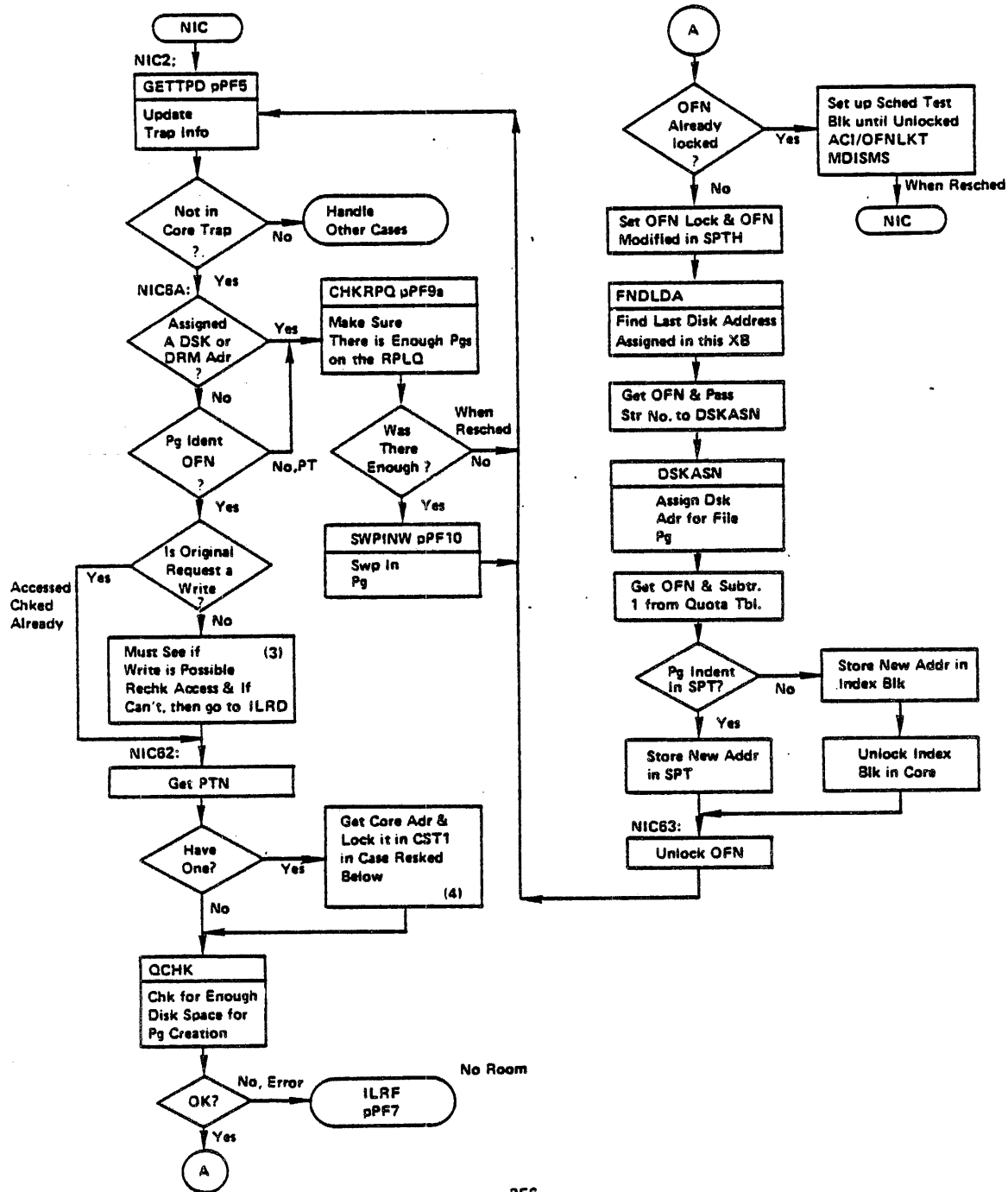
pF4

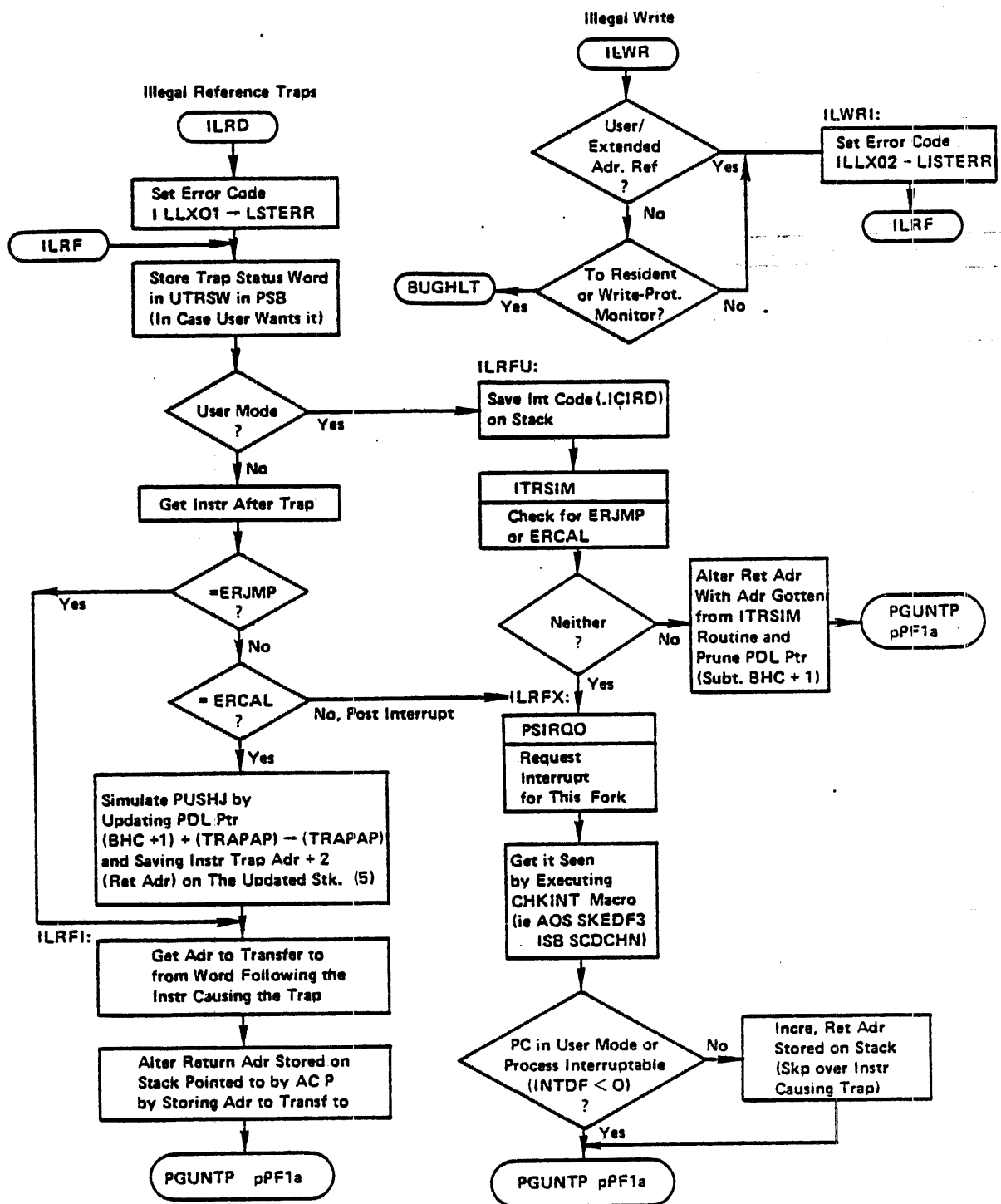


Determine Cause of Trap



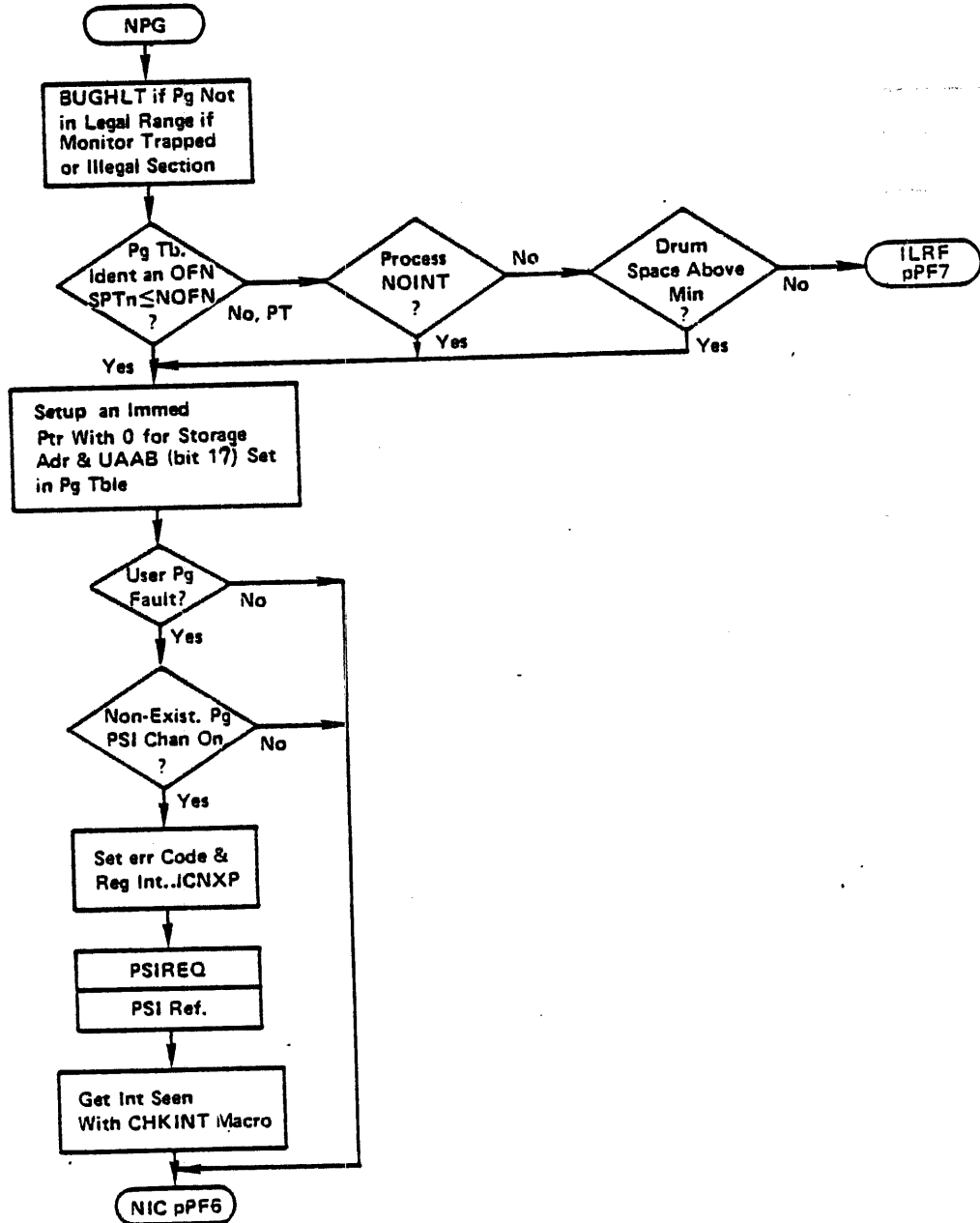
PF5

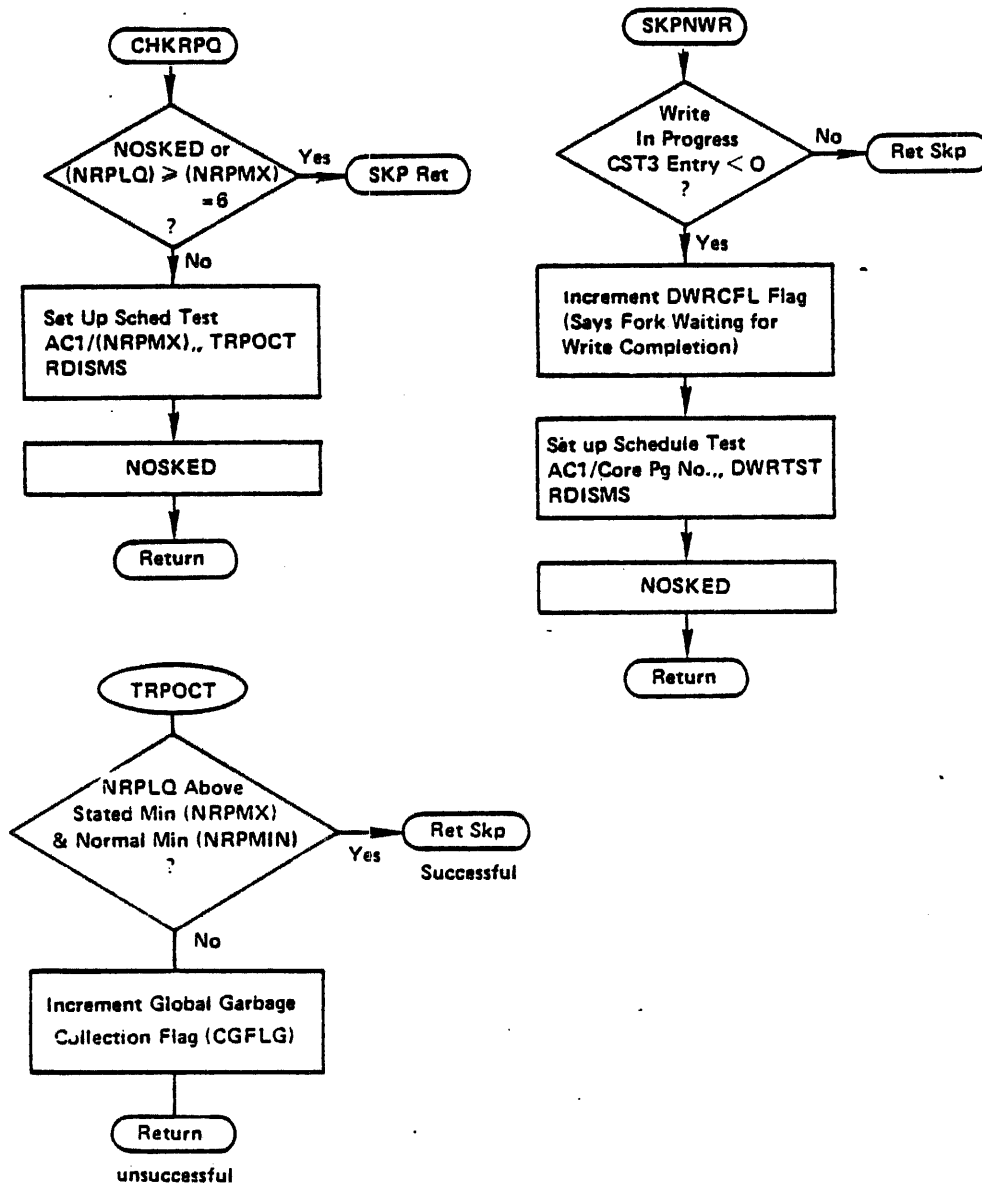




PF7

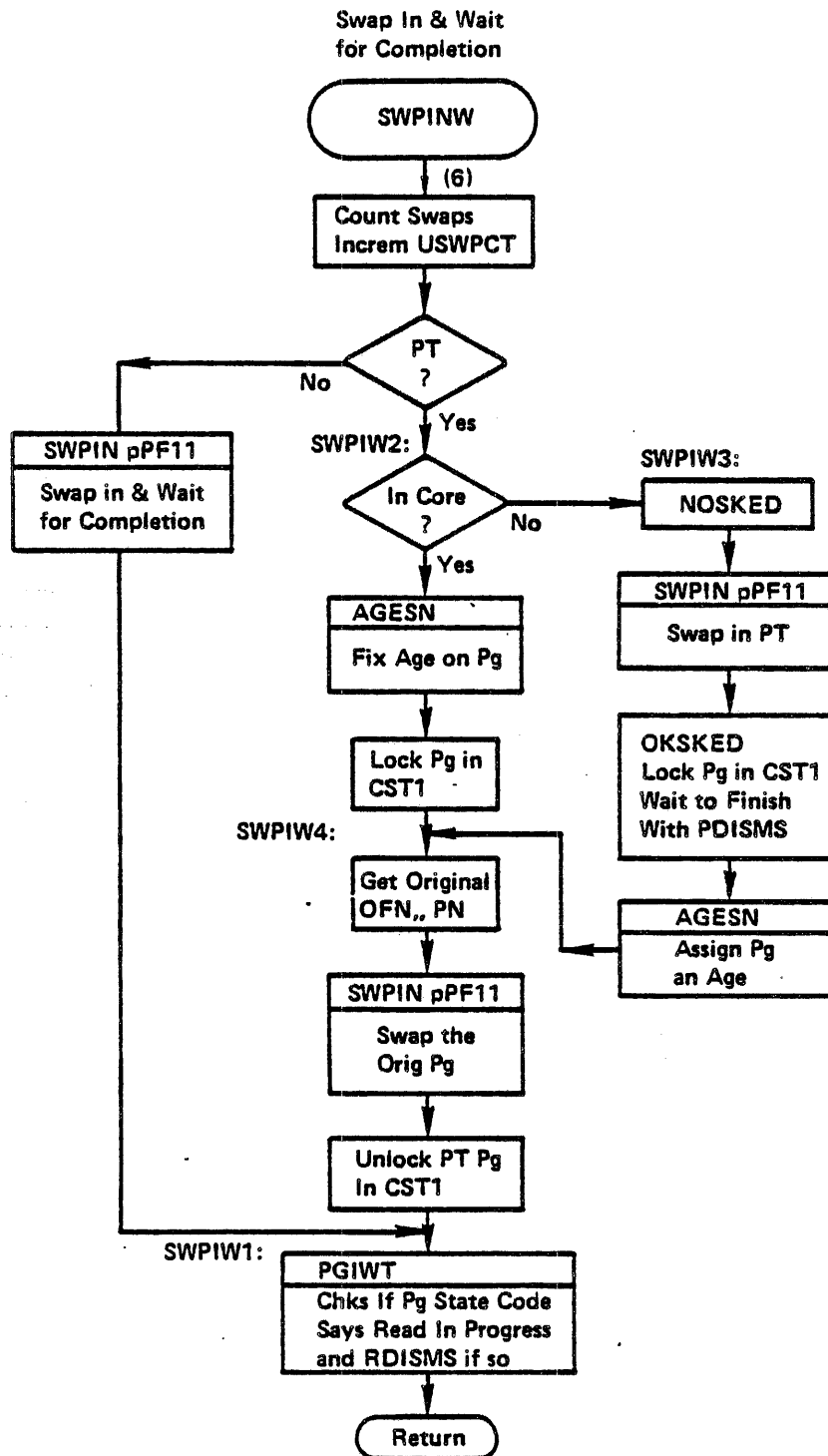
Pg Not in Existence Trap



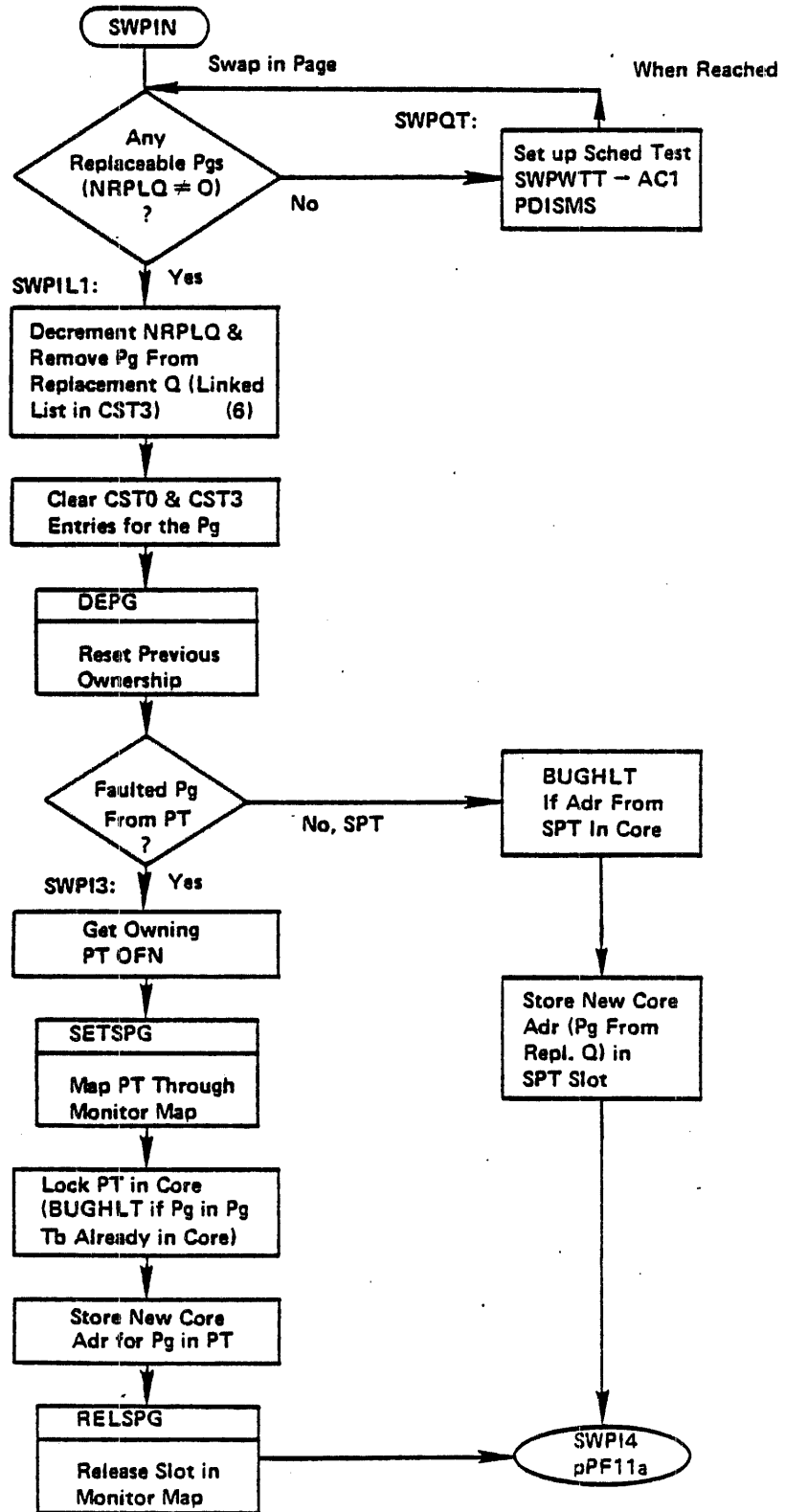


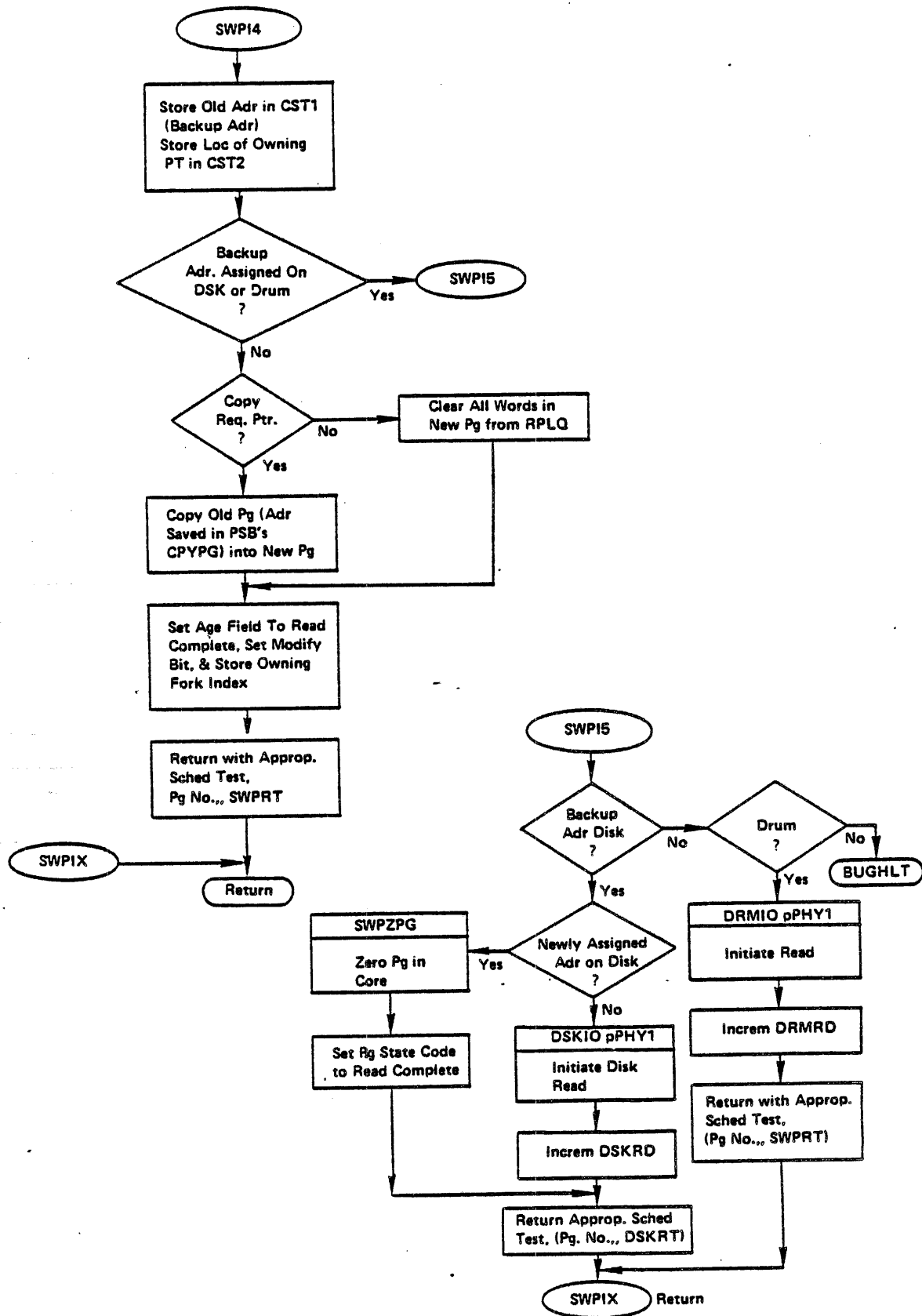
PF9a

REQUESTING DRUM OR DISK READ
(PAGE LEVEL)



REQUESTING DRUM OR DISK READ (Continued)
(PAGE LEVEL)





MULTIPLE PAGE SWAP OUT ROUTINE

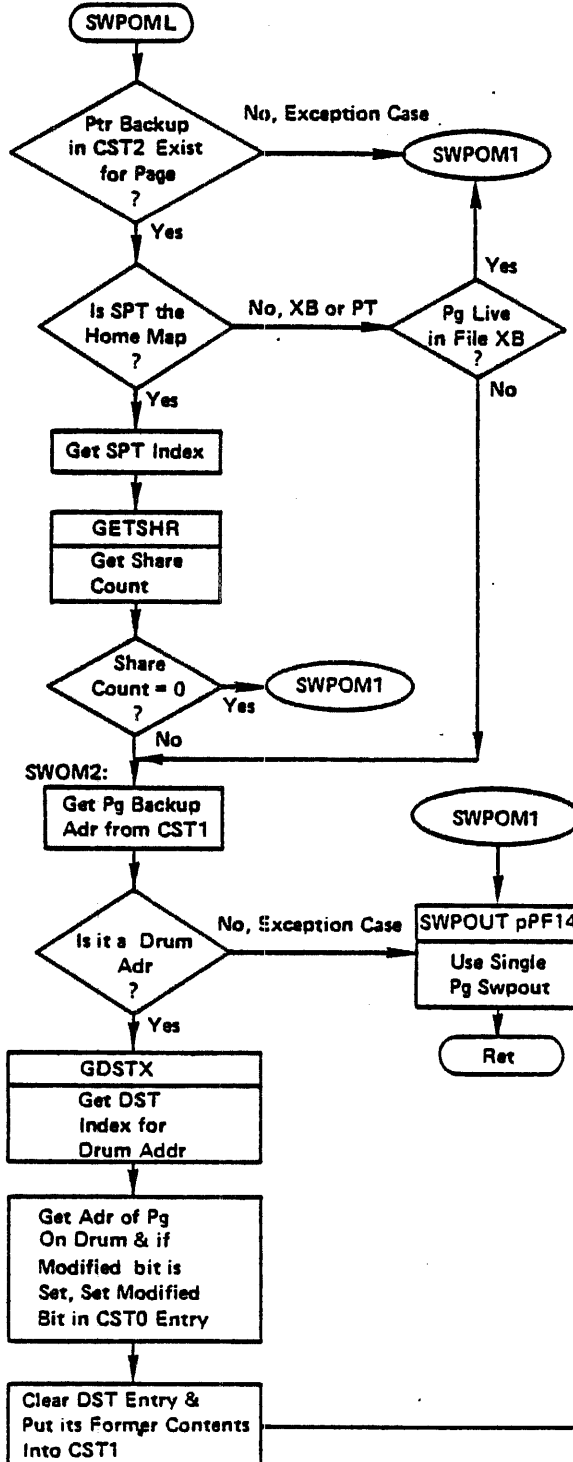
SWPOMI - Init List

SWPOML - Called to Add Page to Swap Out List if Possible

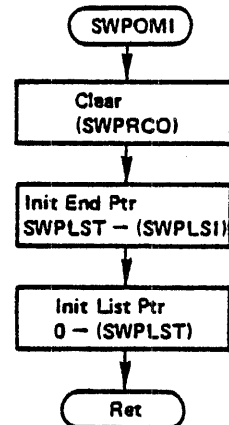
SWPOMG - To Begin I/O for All Pages on Swap Out List

SWPOUT - Initiate Swap Out of Single Page

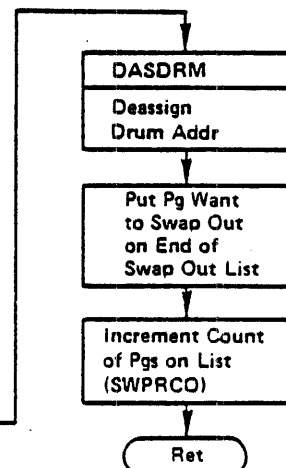
Add Pg to Swap Out List if Possible



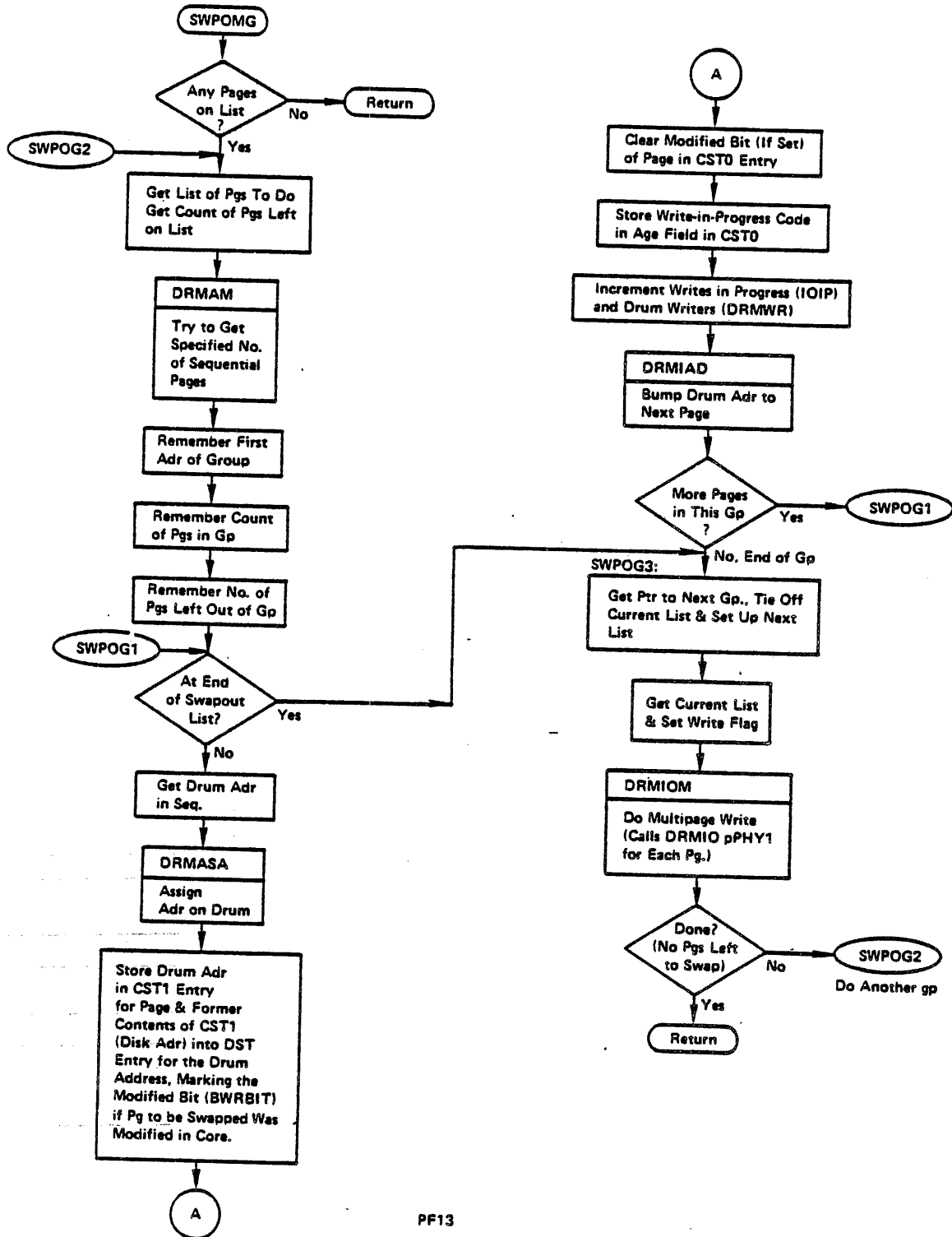
Init Swap Out List



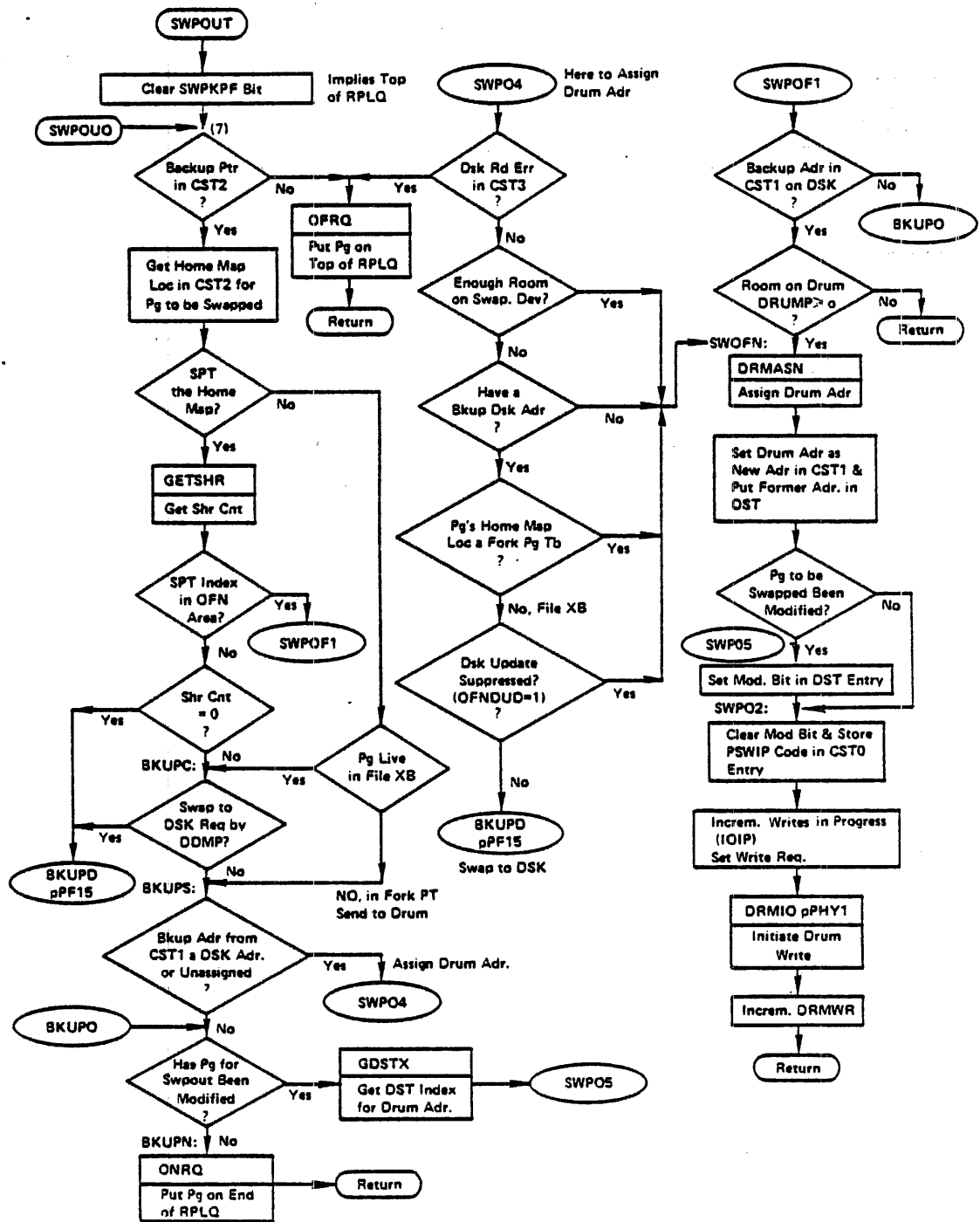
Pg Can't be Written on Drum or has no Drum Address



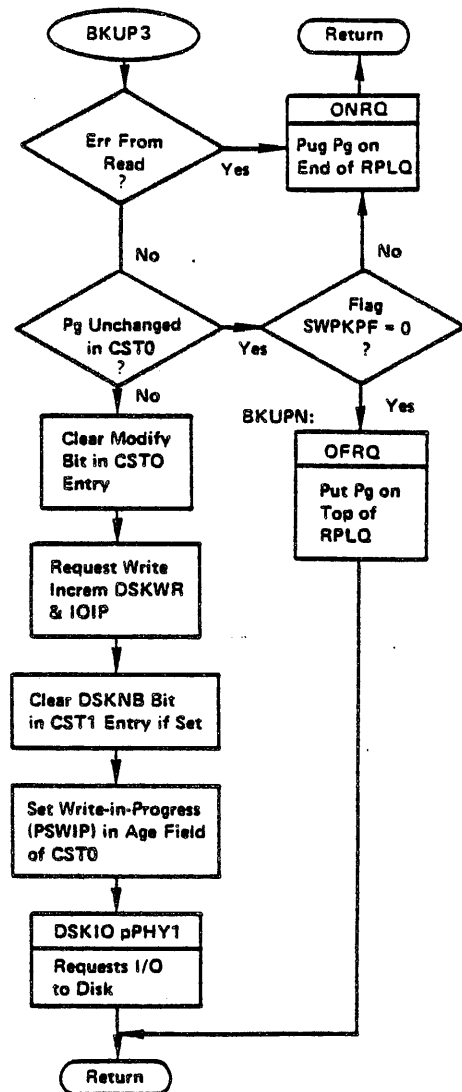
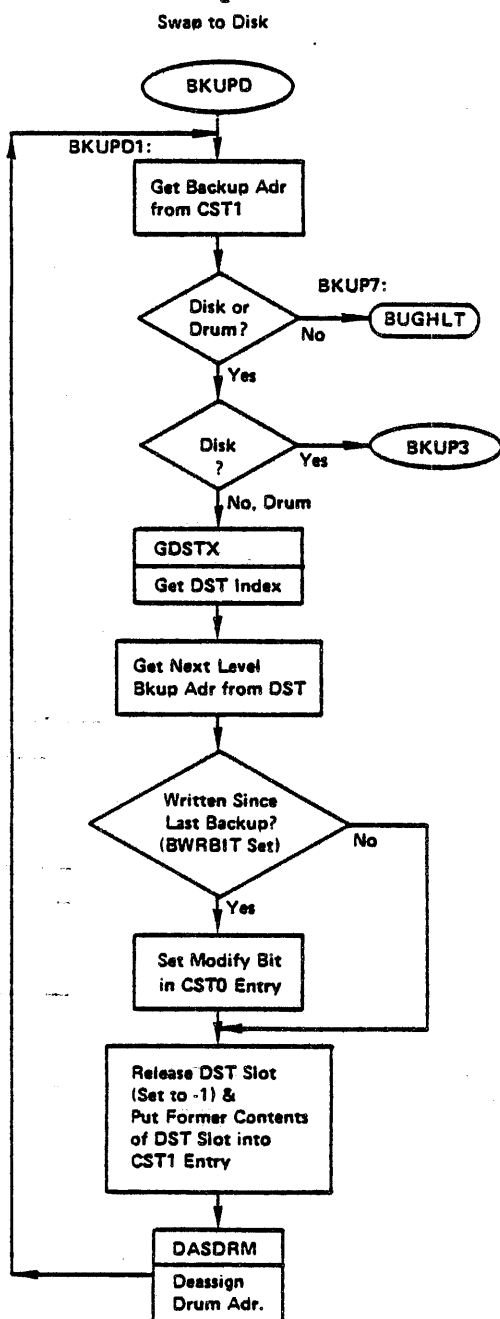
Assign New Drum Storage &
Initiate I/O for All Pages on
SWPOUT List



PF13



PF14



Page Fault Handling Comments

PGTACC

- (1) Checks if process has accrued more than or equal to the number of age ticks of GCRATE. Currently, this is set to 50, which implies 2 sec. of process virtual time (i.e., the age stamp is incremented every 40 ms of process run time).

NICCKS

- (1) GNPBAS is currently initialized at system startup to zero and is incremented/decremented only when pages are locked/unlocked. It is currently only tested by NICCKS as well.

GETTPD

- (2) The age field when used to hold the age stamp, will always have a value of 100 or greater. This checks if any of the lefthand 5 bits of the age field are set.

NIC

- (3) Could take the ILRD path, for example, when OPENed file for write, but PMAped for each of a nonexistent page. A page would have to be created which would then imply a write which was not enabled under PMAP.
- (4) If file page faulted does not have its own SPT slot, but has to be mapped (using indirect pointer) via the index blk slot in the OFN area, then the index blk will be locked in core. (So can't be swapped in case of reschedule.)
- (5) Note in the predispatch code that AC1 was stored in BHC + 1 and AC, P, which holds a push down list pointer, was saved in TRAPAP.

SWPINW

- (6) SWPINW will invoke SWPIN to swap in a page into a page from the RPLQ. However, this same code can also be entered with different flag settings and be used to swap in a page into a page from the special memory queue (SPMQ), a queue used by the memory error handling code.

SWPOUT

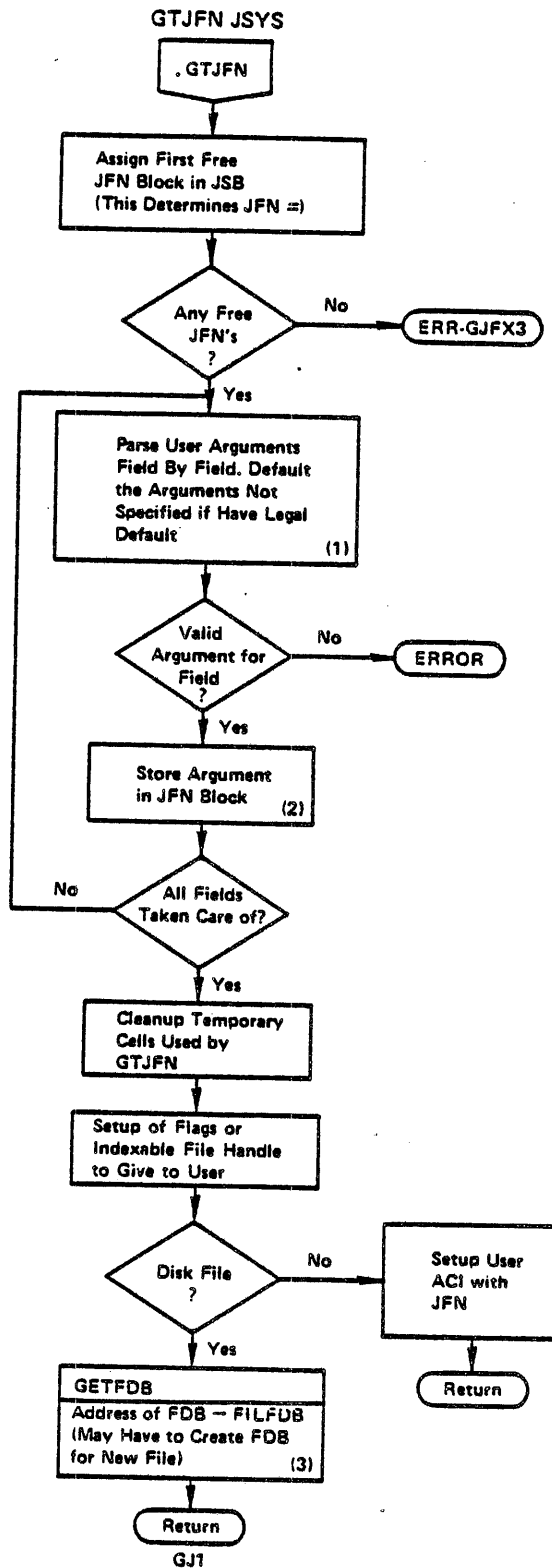
- (7) SWPOUO is called from:

SWPOTO which clears the SWPKPF bit (for top of RPLQ) before calling SWPOUO and

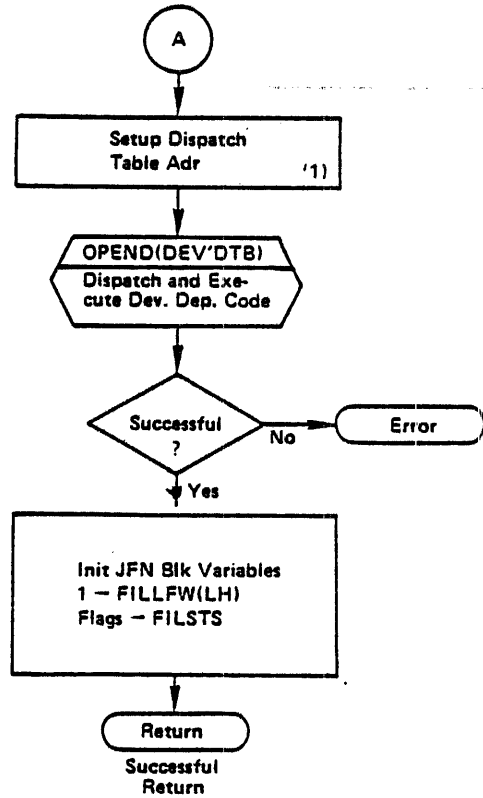
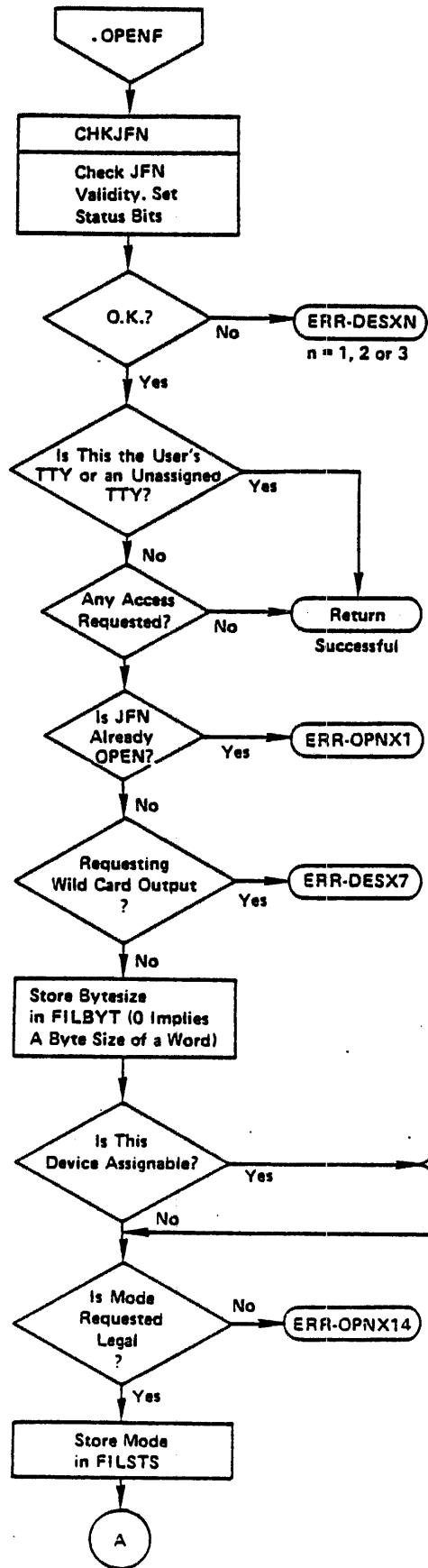
SWPOTK (called from the UPDPGS JSYS) which sets the SWPKPF bit (for end of RPLQ) before calling SWPOUO.

JSYS CALL FLOWCHARTS
DEVICE INDEPENDENT LEVEL

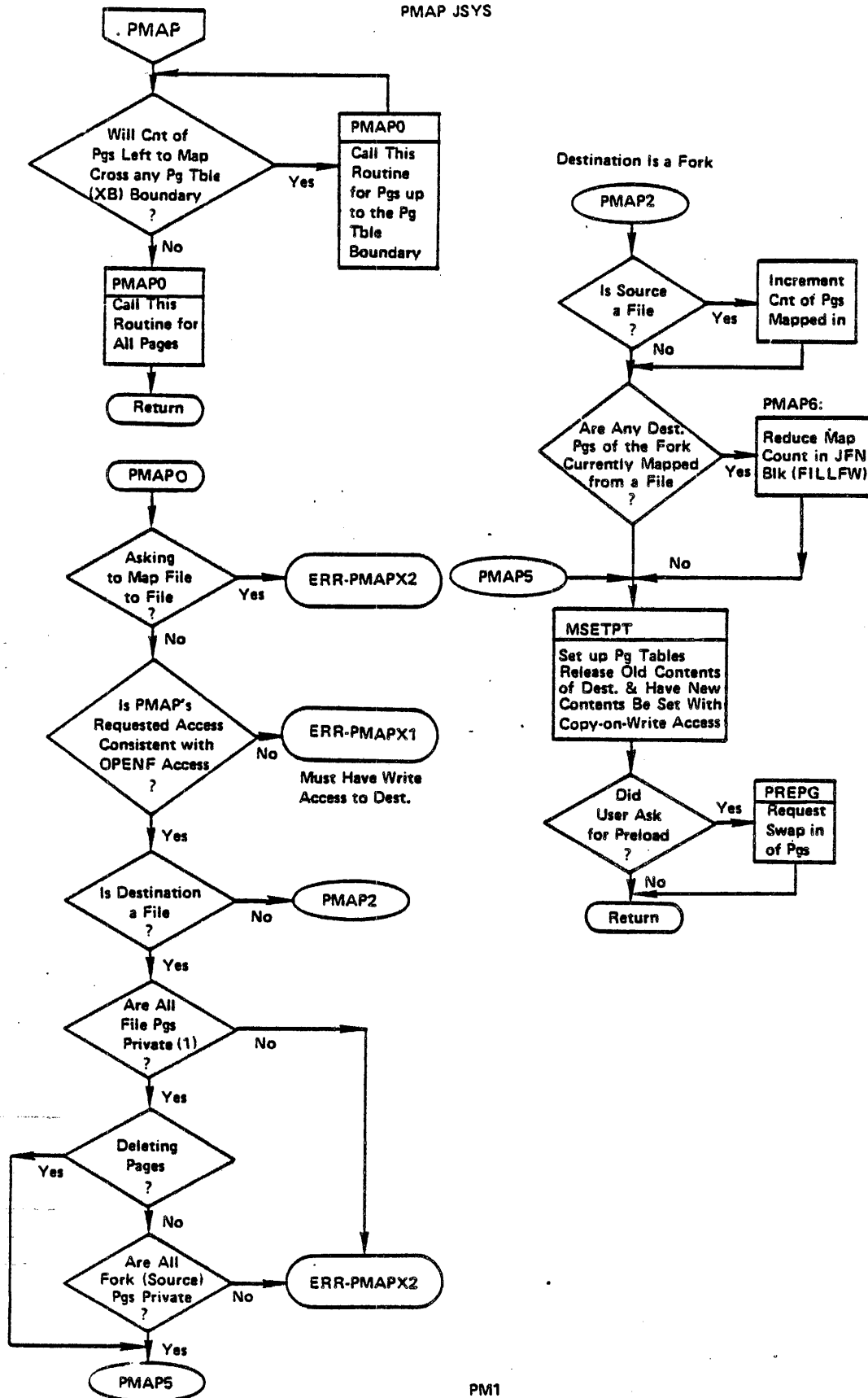
GTJFN -	Get a JFN	GJ1
OPENF -	Open a File	OP1
SIN/SINR-	Sequential Input	S1
	BYTINA - Call Device Dependent Code to Get a Byte	S2
	SIOR2 - String I/O Multiple Byte Transfer	S2
SOUT/SOUTR -	Sequential Output	S3
	BYTOUA - Send Byte to a Service Routine	S4
PMAP -	Map a File or Fork	PM1
UFPGS -	Update File Pages	UD1
CLOSEF -	Close a File	CL1



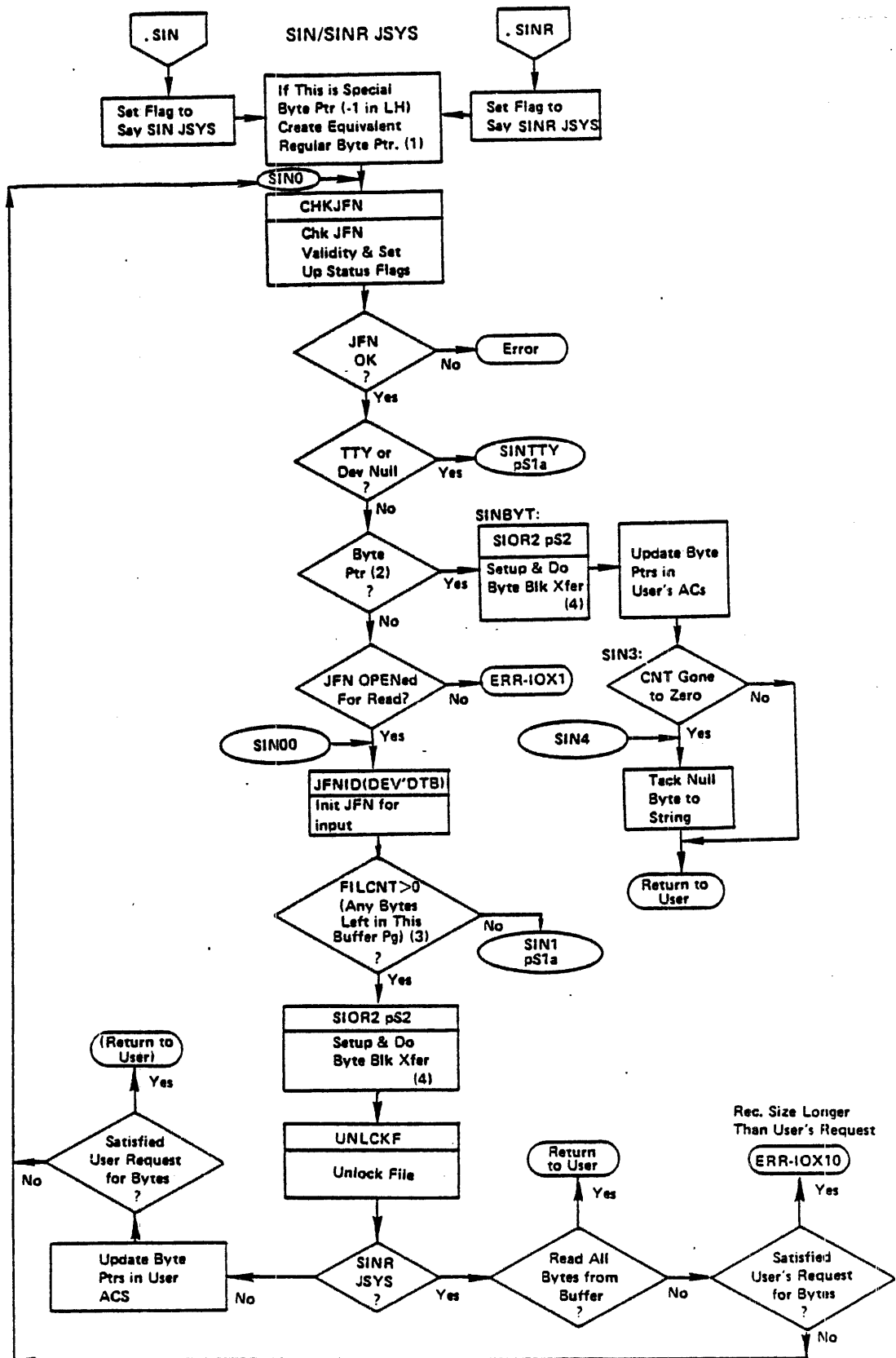
OPENF JSYS



PMAP JSYS

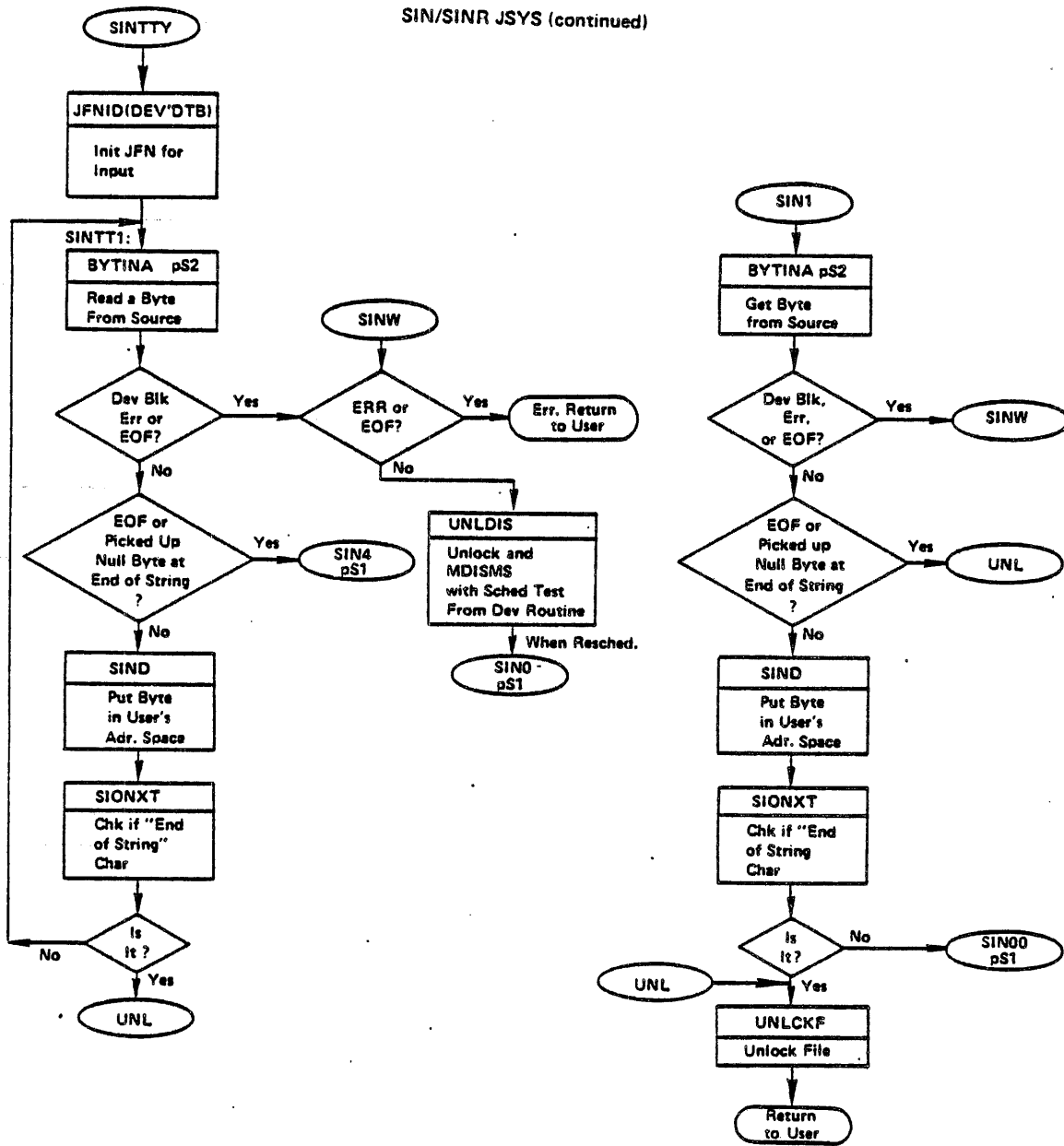


PM1

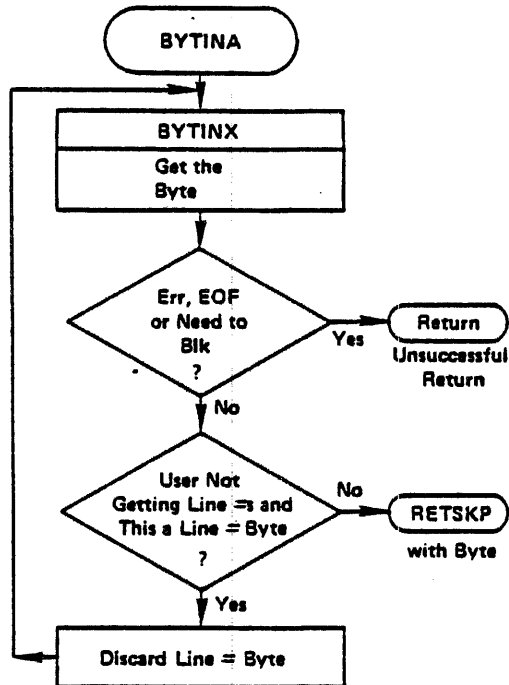


S1

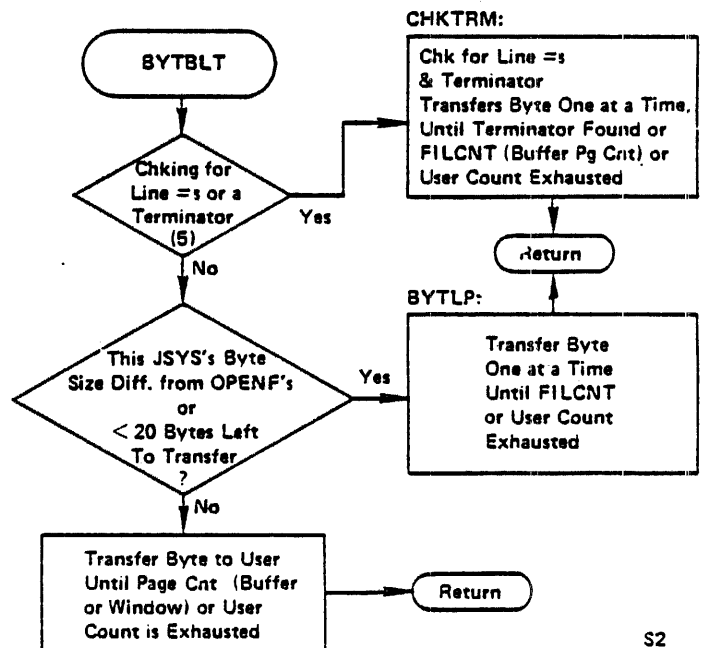
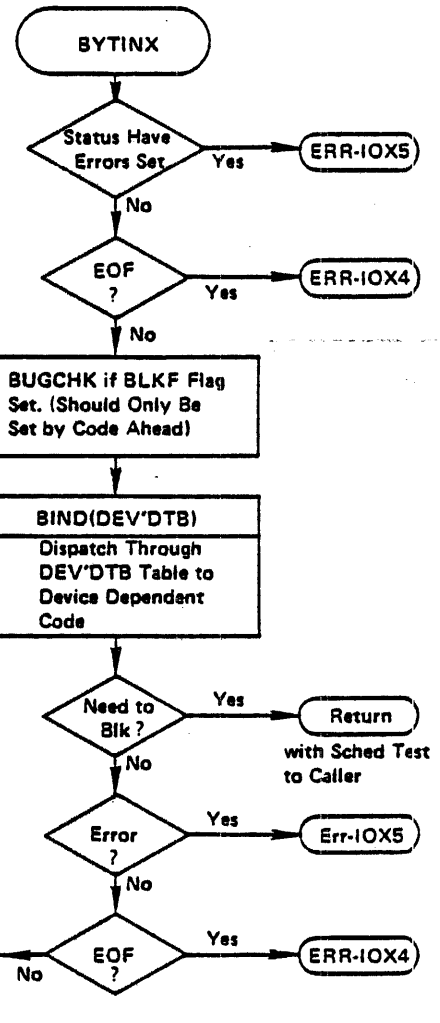
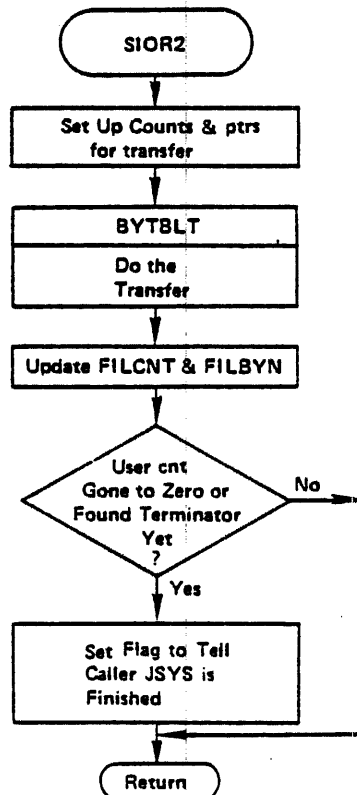
SIN/SINR JSYS (continued)



Call Device Dep. Code to
Get a Byte

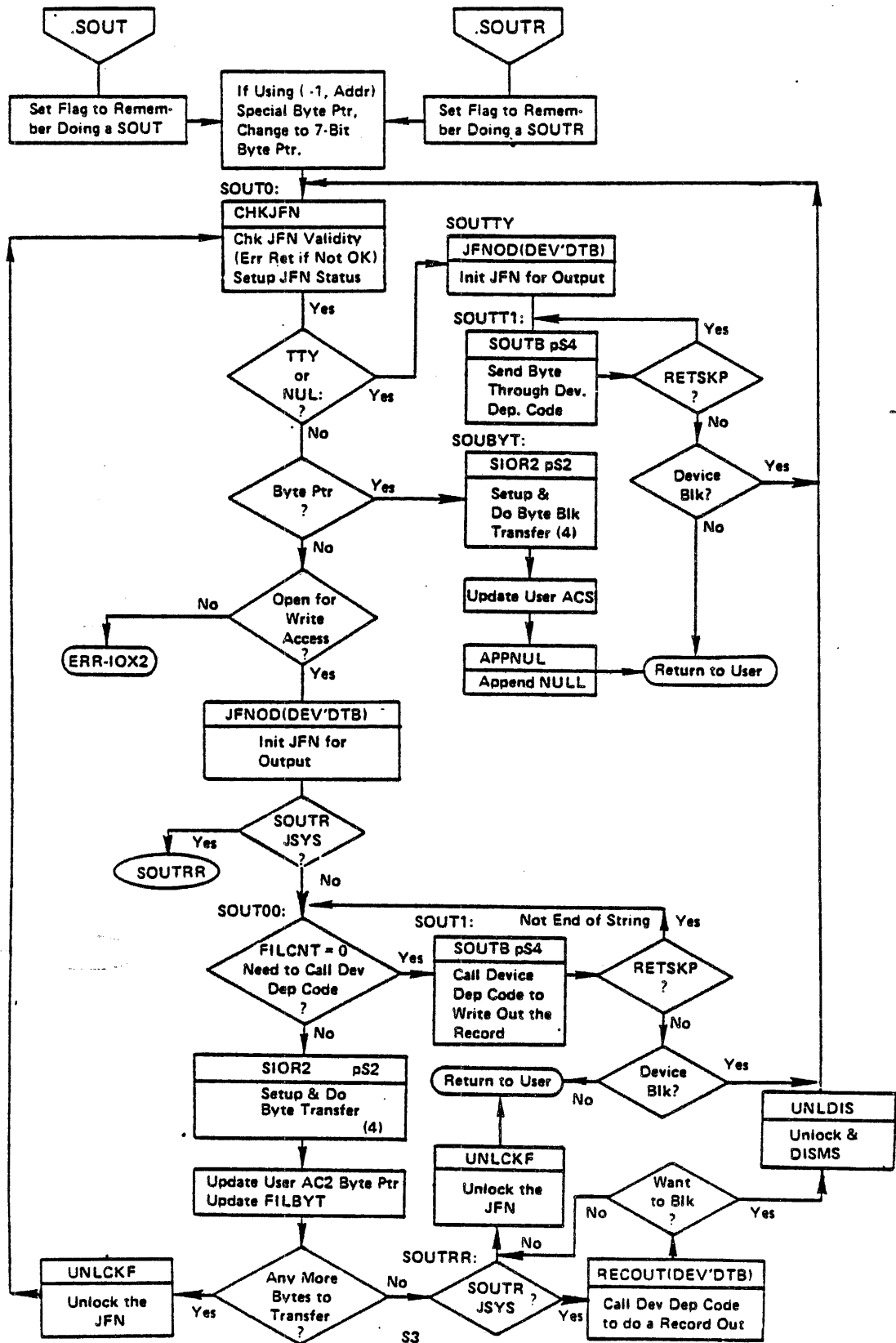


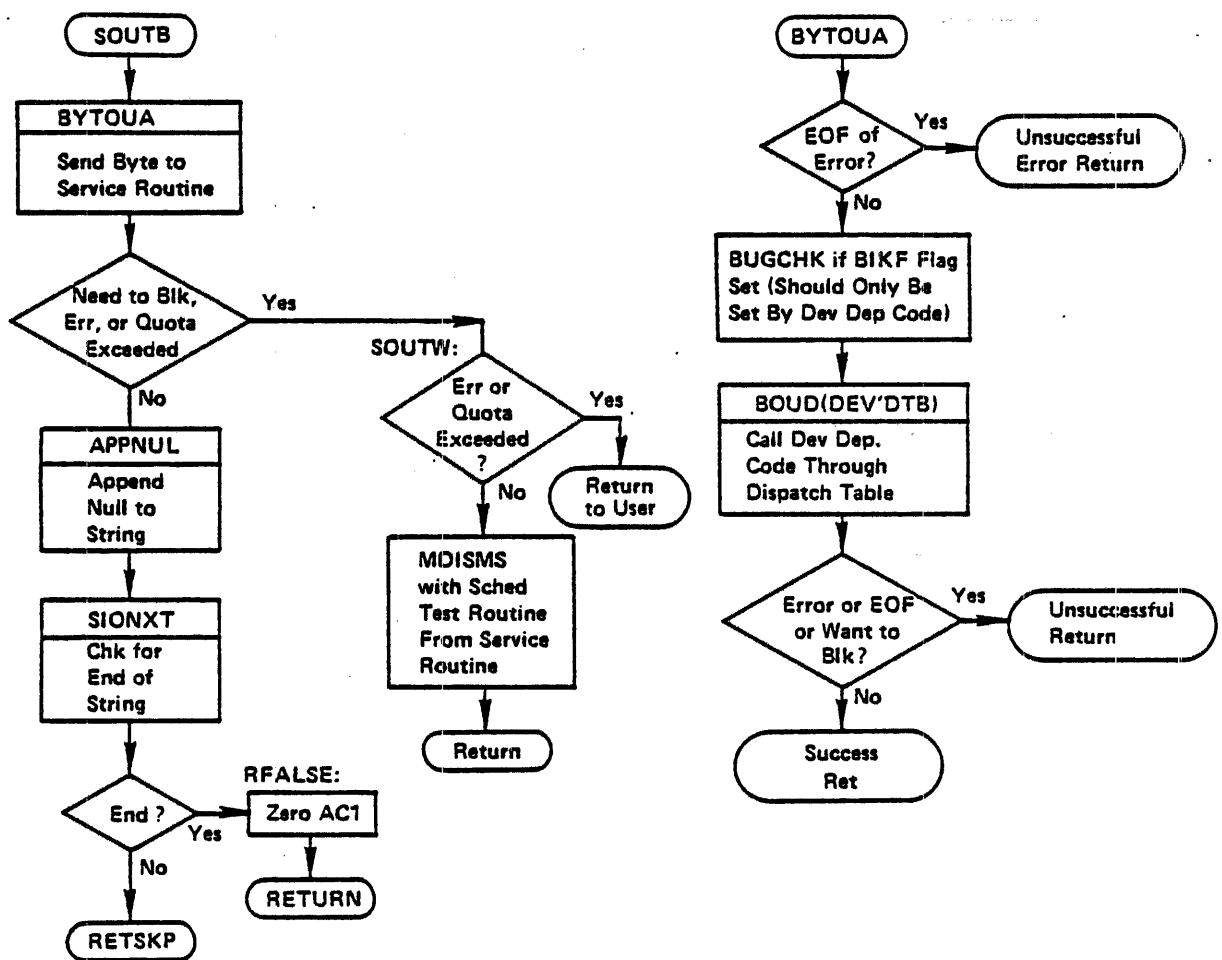
String Input/Output Multiple Byte Xfer



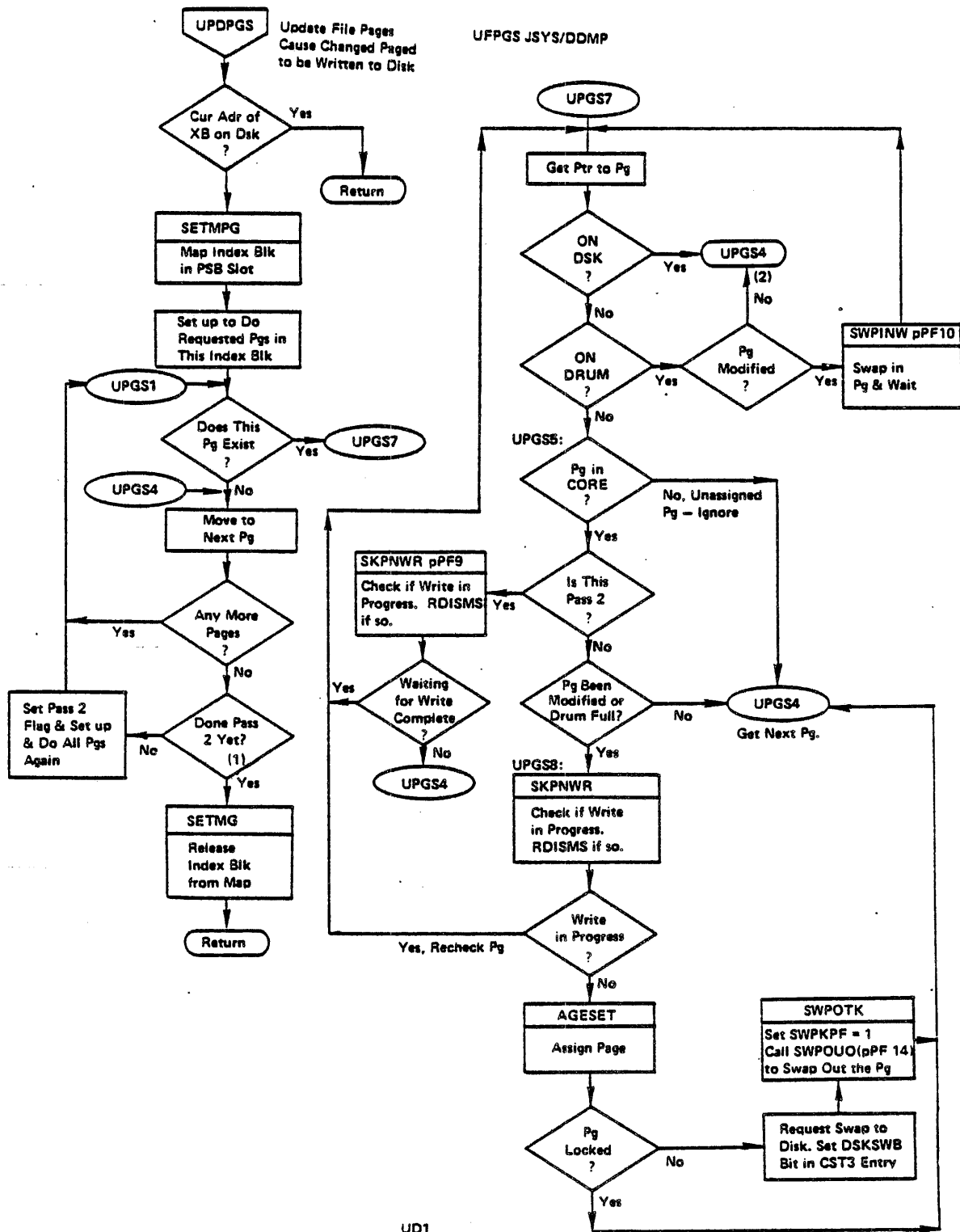
S2

SOUT/SOUTR JSYS

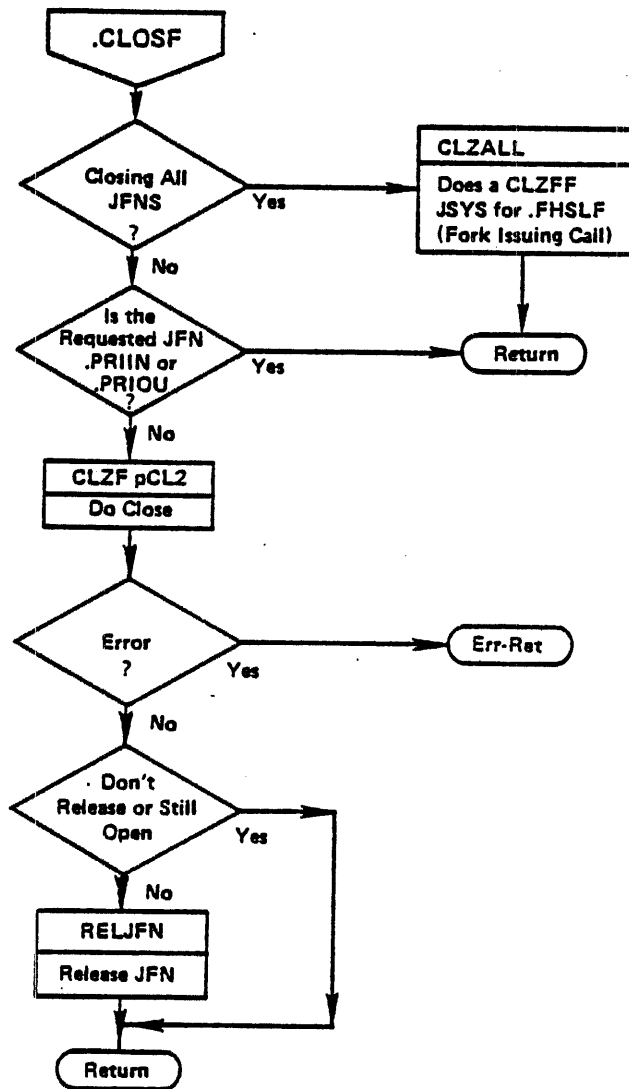




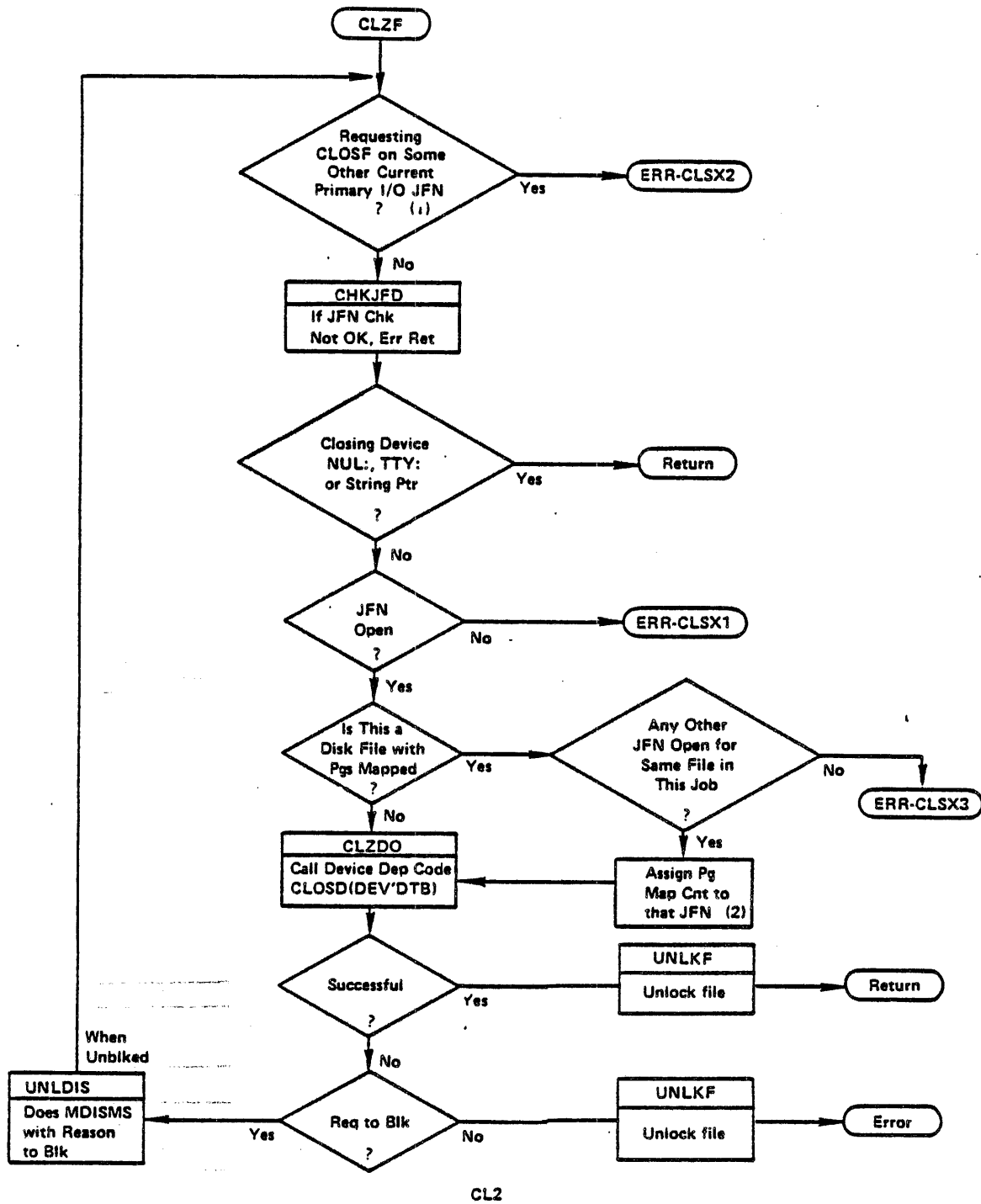
S4



CLOSF JSYS



CL1



GTJFN Comments

- (1) This code is looking for a file specification of the form:

Dev:Directory Name,type,gen;T(temporary);P(protection);A(account

One or more fields can be defined by logical names.
If any fields are omitted from the specification, the
system will default the values as follows:

Device	DSK:
Directory	Connected directory
Name	No default for disk
	Null for other devices
Generation	Highest existing for input
	Next highest for output
Protection	As specified for directory or
	protection of next lower generation
Account	Current user account

- (2) The internal GTJFN code uses several locations in the JFN block as temporary cells. These locations have two names in the JFN block table descriptions. The JFN block storage locations set up or used by GTJFN are:

FILLCK*	FILDDN	FILNEN
FILTMP*	FILPRT	FILVER
FILACT*	FILSTS1	FILCOD (LH)
FILOPT*	FILLNM*	
FILDEV	FILDNM	

*Used internally only by the GTJFN JSYS.

- (3) The creation process of the FDB simply asks for space in the directory for the FDB.

.OPENF Comments

- (1) Cell FILDEV in the JFN blk has the device dispatch table address. For example, for disk, GTJFN sets the dispatch table address to DSKDTB. If spooling to disk, GTJFN sets the dispatch table address to SPLDTB, but the OPENF code changes the dispatch table address to DSKDTB and sets up a file specification in the JFN block.

.SIN/.SOUT Comments

- (1) TOPS-20 allows a user to specify a special byte pointer of -1,, Address which is interpreted as a 7-bit byte size beginning on the word boundary, Address.
- (2) A user can do I/O from one place to another in core by specifying byte pointers for both source and destination. This differs from BLT in that the use can transfer on non-word boundaries.
- (3) For disk files, FILCNT will be the number of bytes remaining in the window page. For magtape and other devices it will be the number of bytes remaining in the current page of the buffer.
- (4) The routine BYTBLT only moves data up to the page boundary of the current buffer page.
- (5) If the user has not specified OF%PLN in the OPENF, line numbers are stripped off the beginning of each line. (See SIN JSYS in Monitor Calls manual for definition of terminator.)

.PMAP Comments

- (1) A page is private if it is not shared between a file and a fork.

UPDPGS Comments

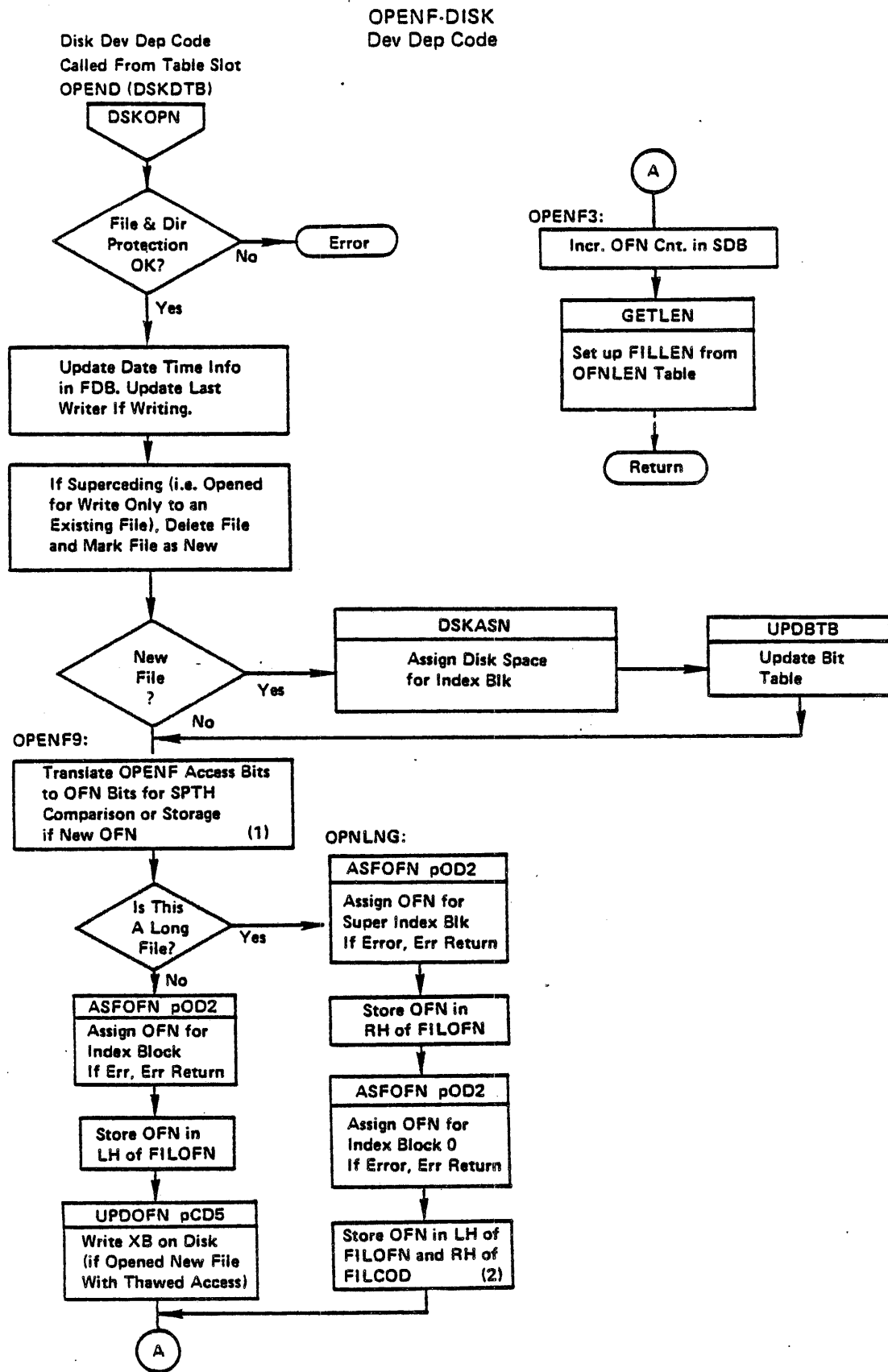
- (1) Routine scans page table twice: first time to request writes on all changed pages. Second time to wait for completion of writes. (This is faster than waiting for each write to complete as it is requested.)
- (2) If page has not been modified, a check is made to see if the drum is full and if so, to release this page back to the drum. The map pointer to the page will be changed to its disk address.

CLOSF Comments

- (1) If user has switched primary I/O to some other JFN and attempts to close it, an error results.
- (2) The page map count in FILEW reflects the number of pages mapped and a CLOSF can't be done on a file if this count is greater than 0.

JSYS CALL FLOWCHARTS
DSK DEPENDENT LEVEL

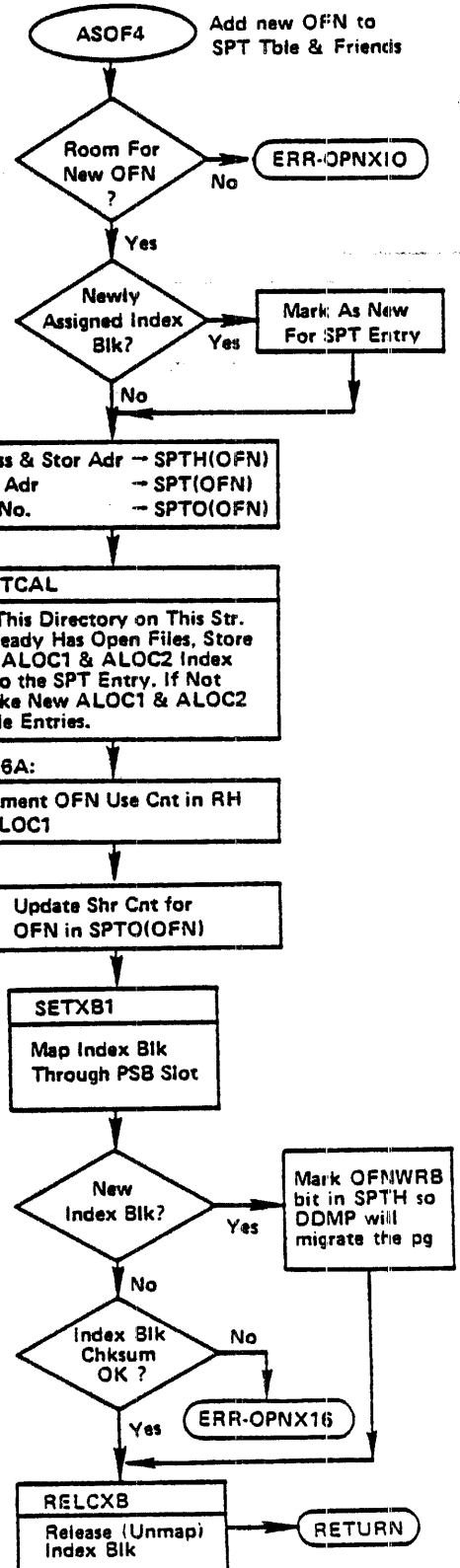
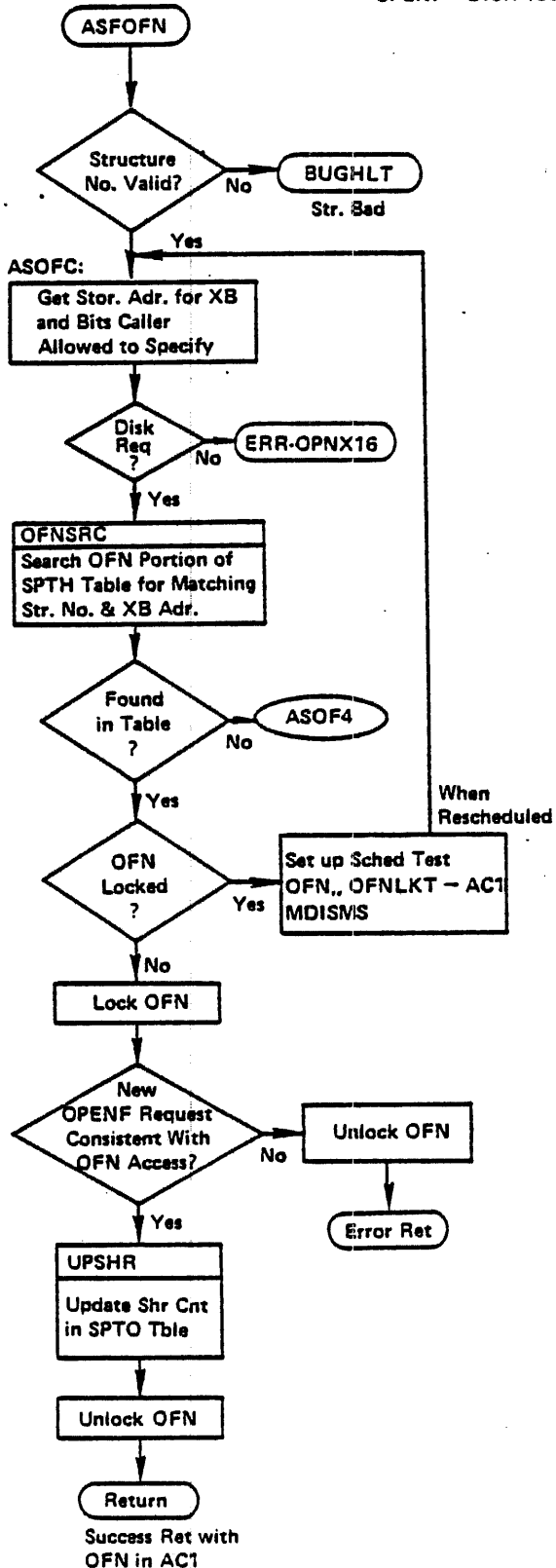
DSKOFN - Disk Opening of a File	OD1
ASFOFN - Assign OFN	OD2
UPDOFN - Update OFN	OD1
DSKSQI/O -Disk Sequential Input/Output	SD1
NEWWND - New Window Page (Next Page of File)	SD2
DSKCLZ - Disk Closing of a File	CD1
RELOFN - Release OFN	CD2
DASOFN - Deassign OFN	CD3
MOVDSK - Move Page Back to Disk .	CD3



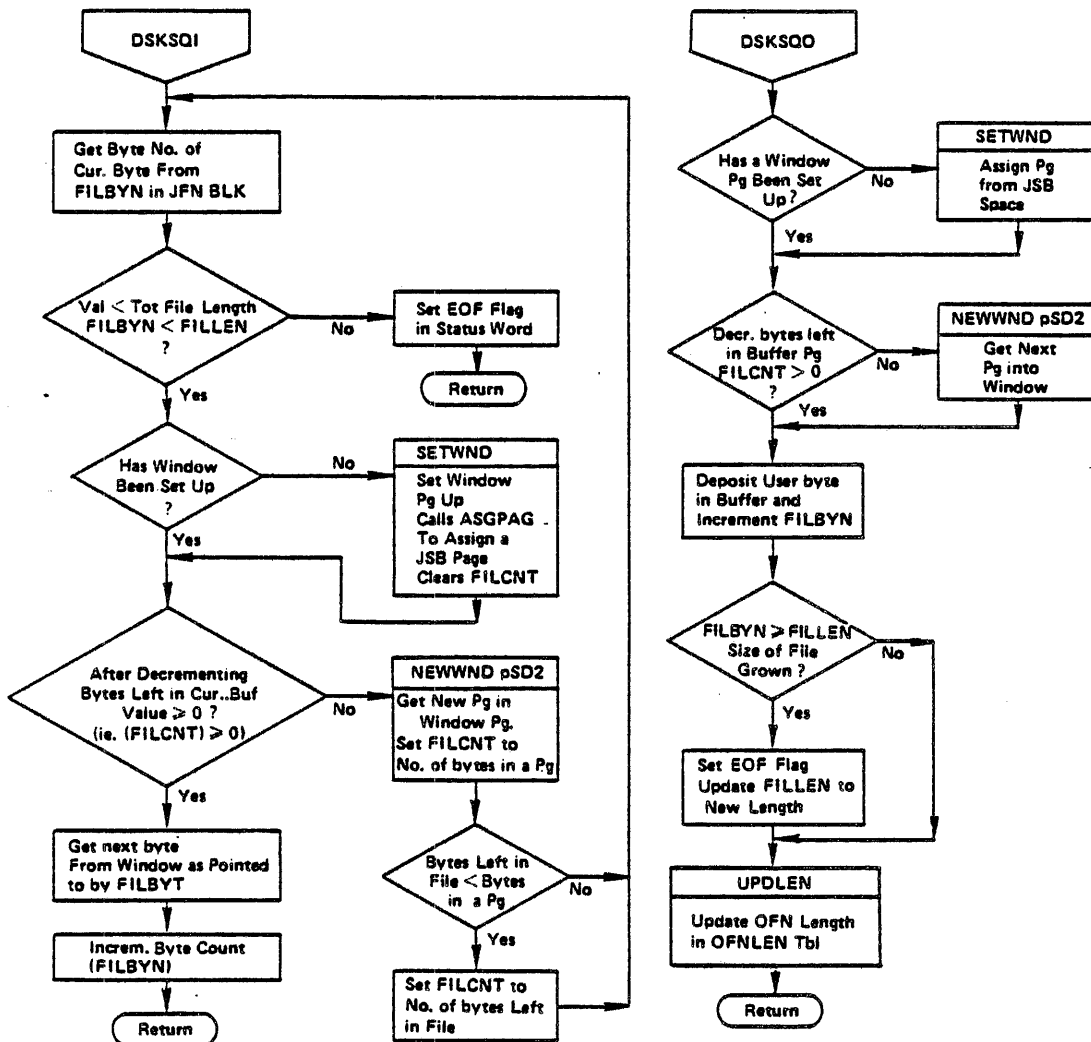
OD1

Assign OFN

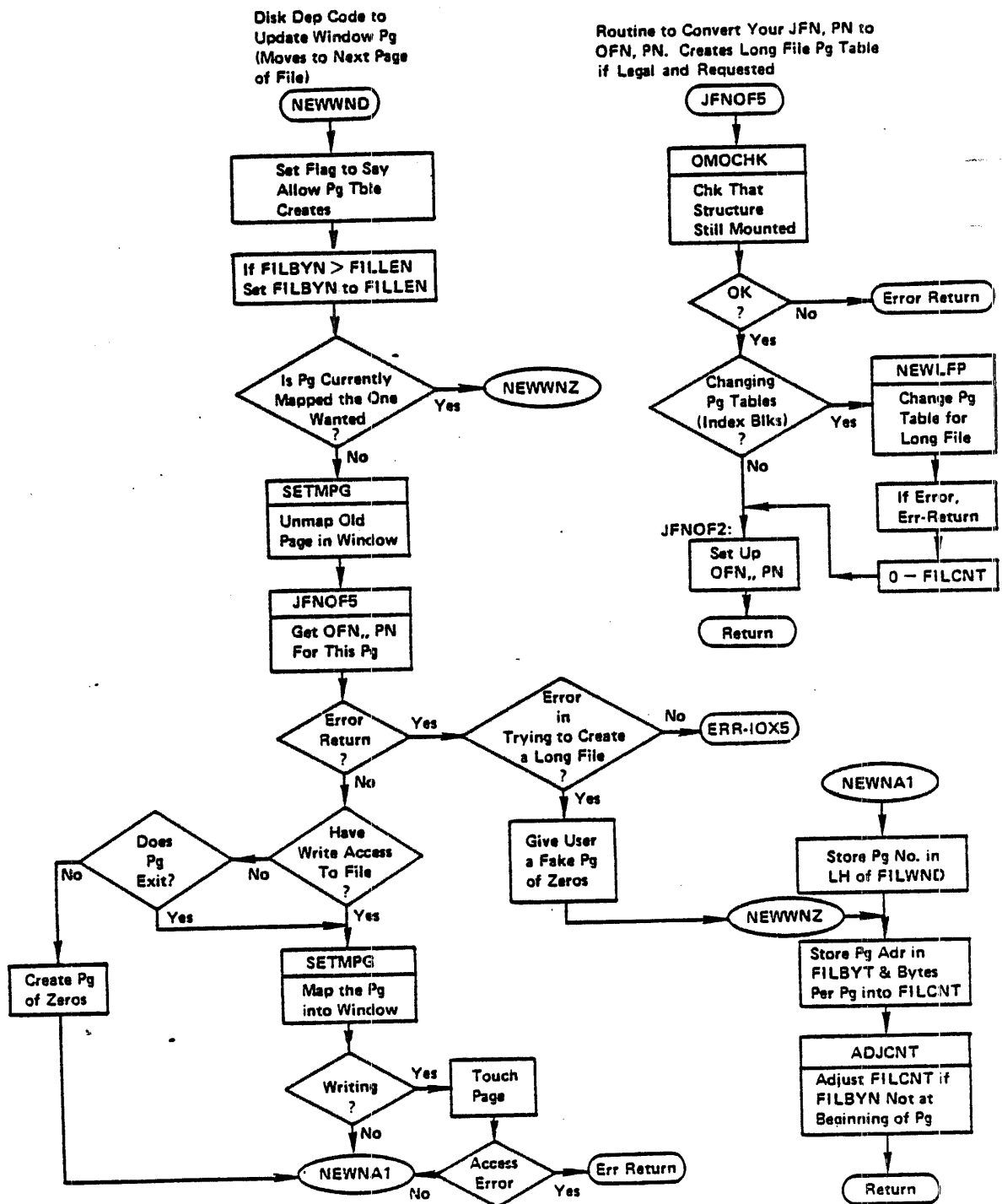
OPENF - DISK (Cont)



SEQUENTIAL I/O-DSK
(String & Byte Dev Dep Code)

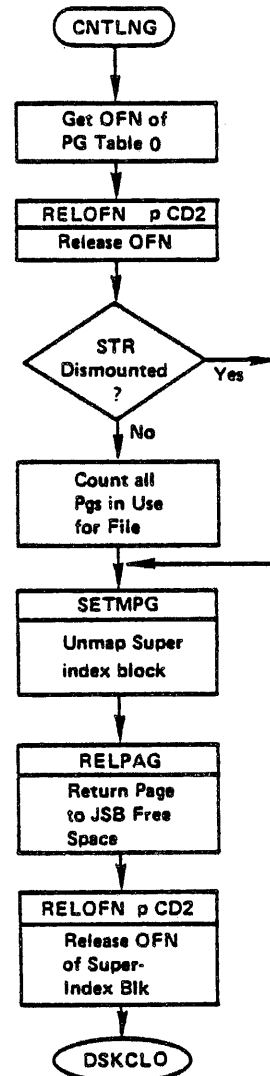
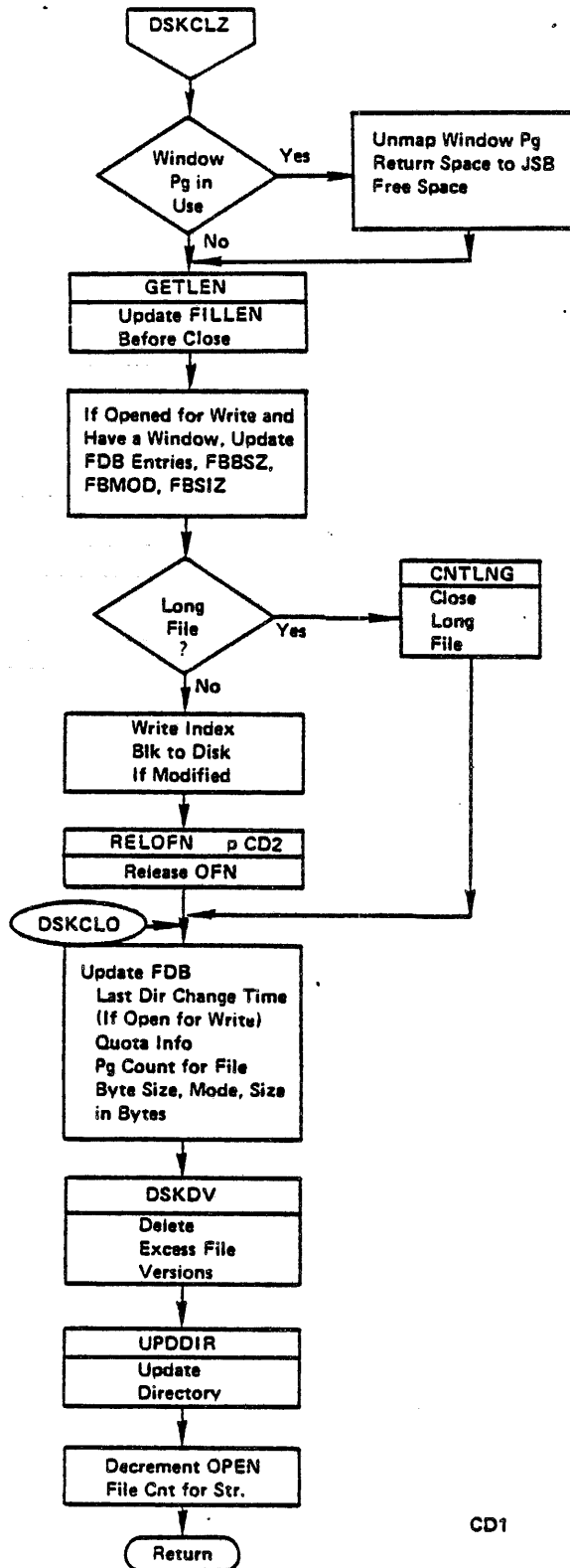


SD1



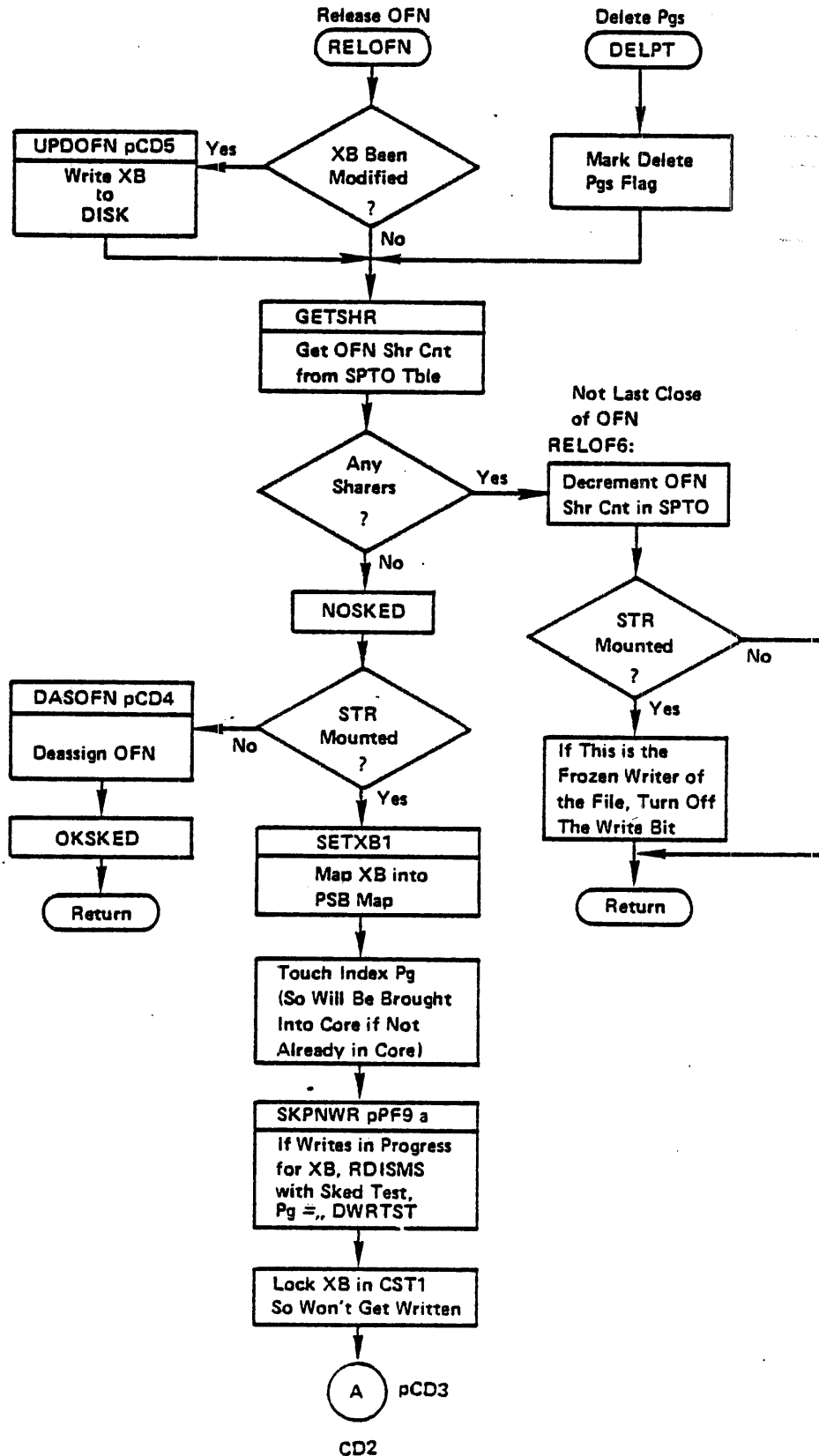
SD2

CLOSF - DSK
Dev. Dep. Code

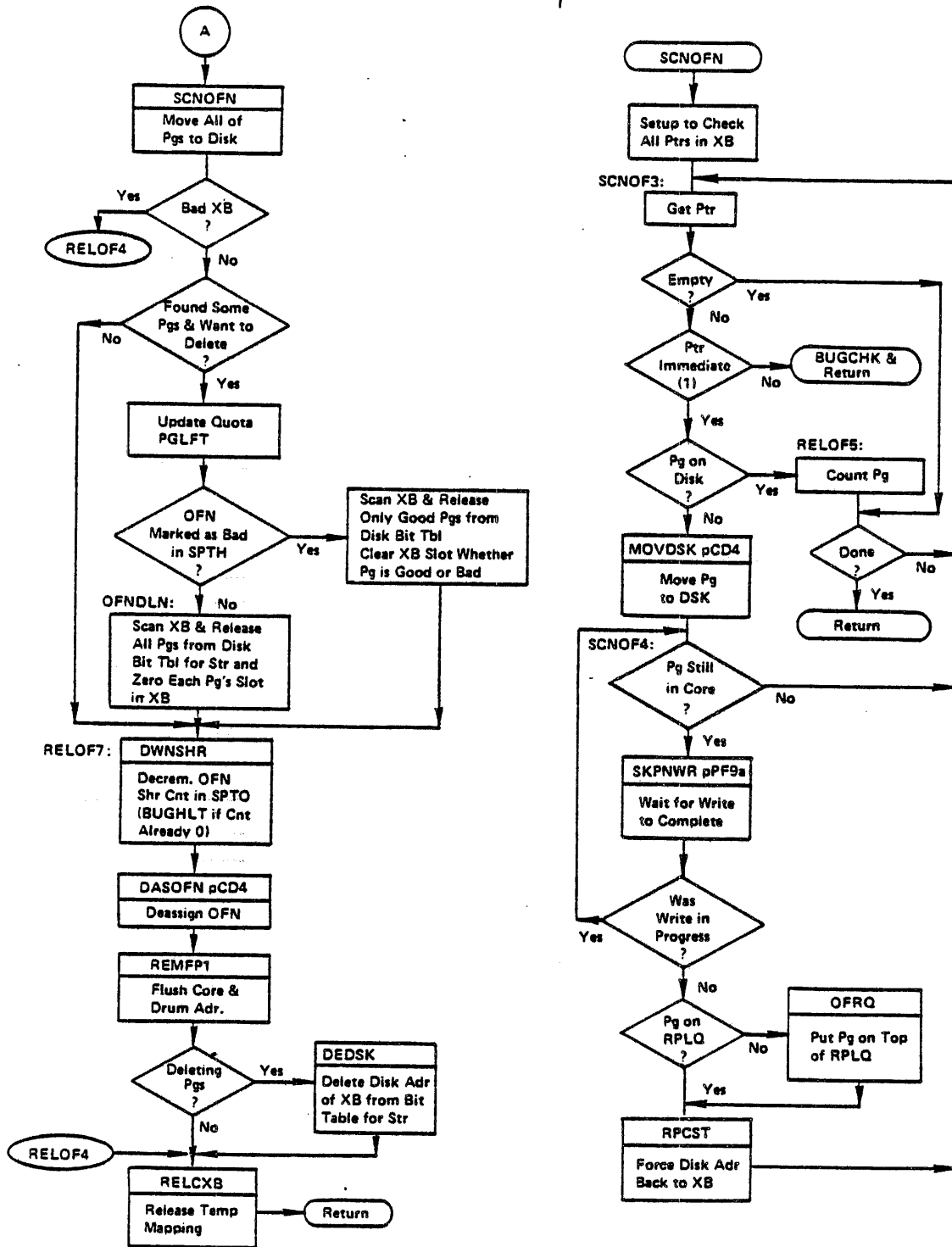


CD1

CLOSF-DSK (Continued)

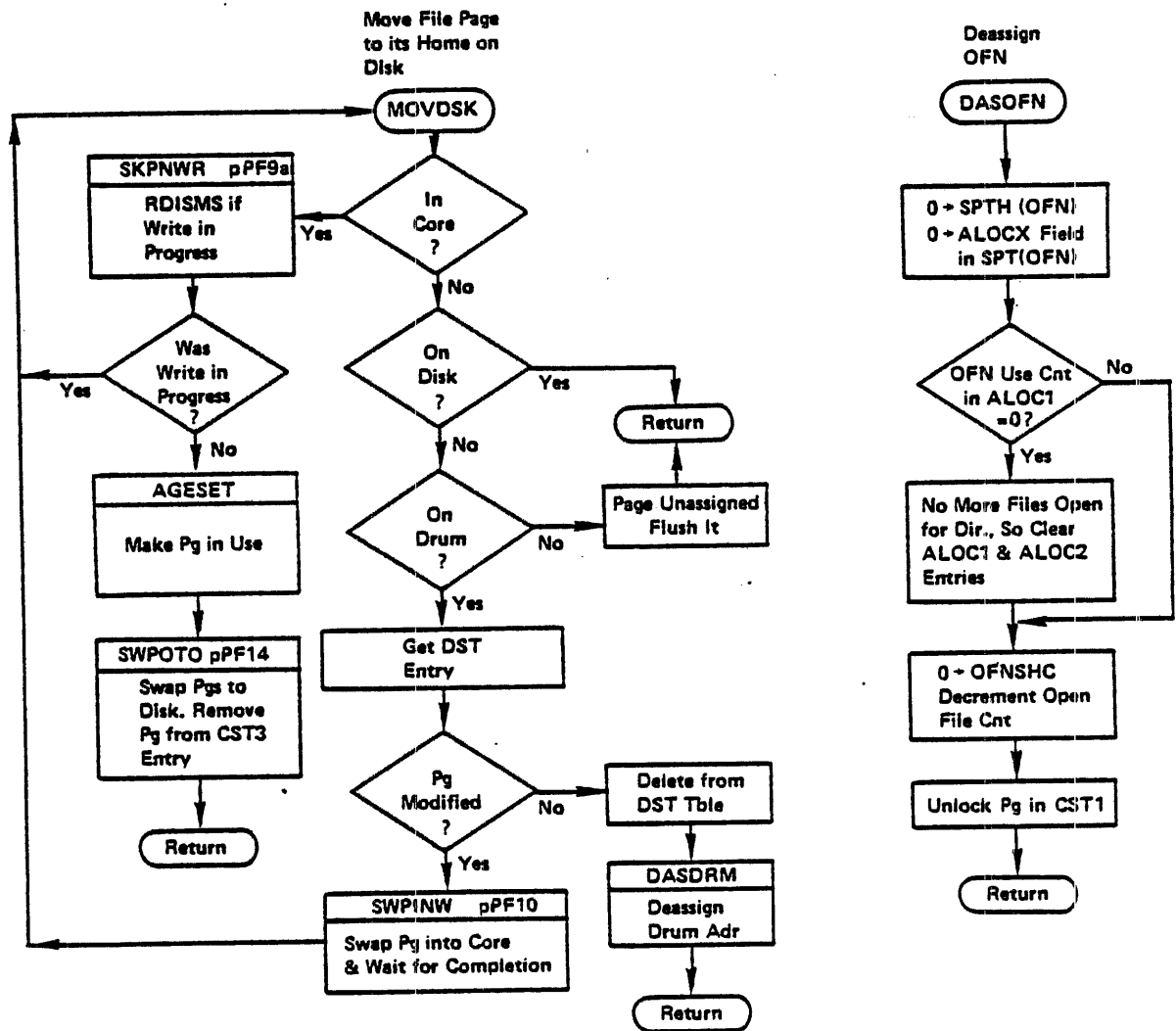


CLOSF-DSK (Continued)

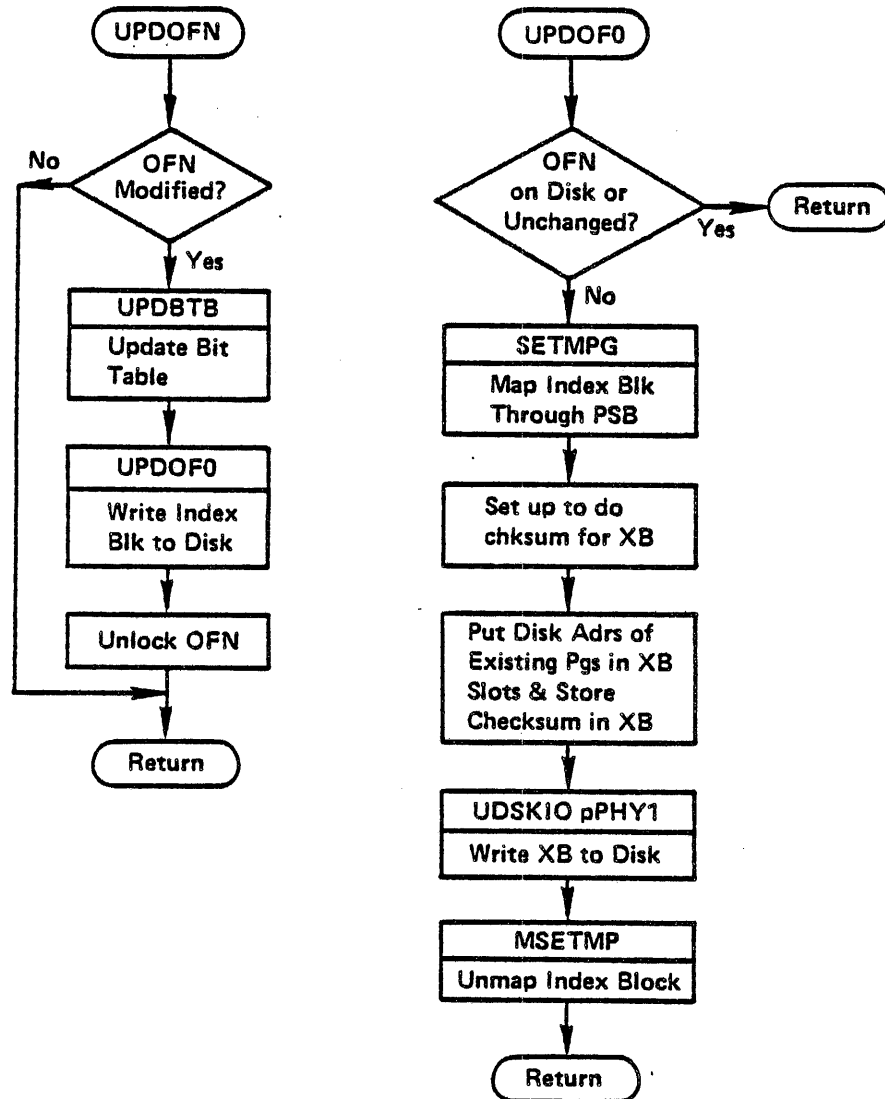


CD3

CLOSF - DISK (Continued)



CLOSF-DISK (continued)



OPENF-DISK Comments

- (1) OFN bits: 0=read, 10=write, 11=thawed, 01=restricted
- (2) For a long file, the OFN of index block 0 is remembered in the JFN blk and used as the identity of the file by the ENQ/DEQ facility.

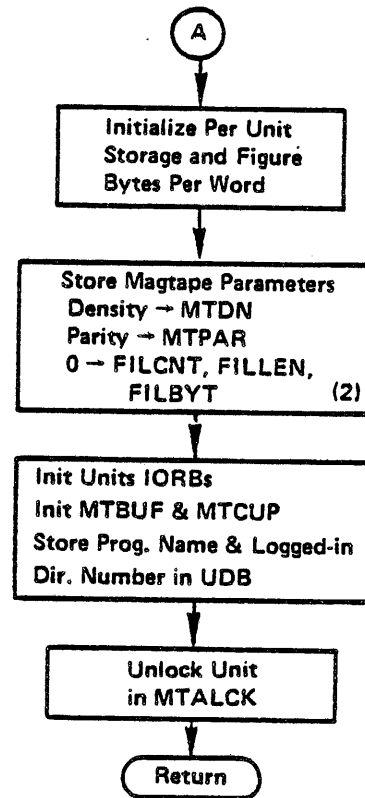
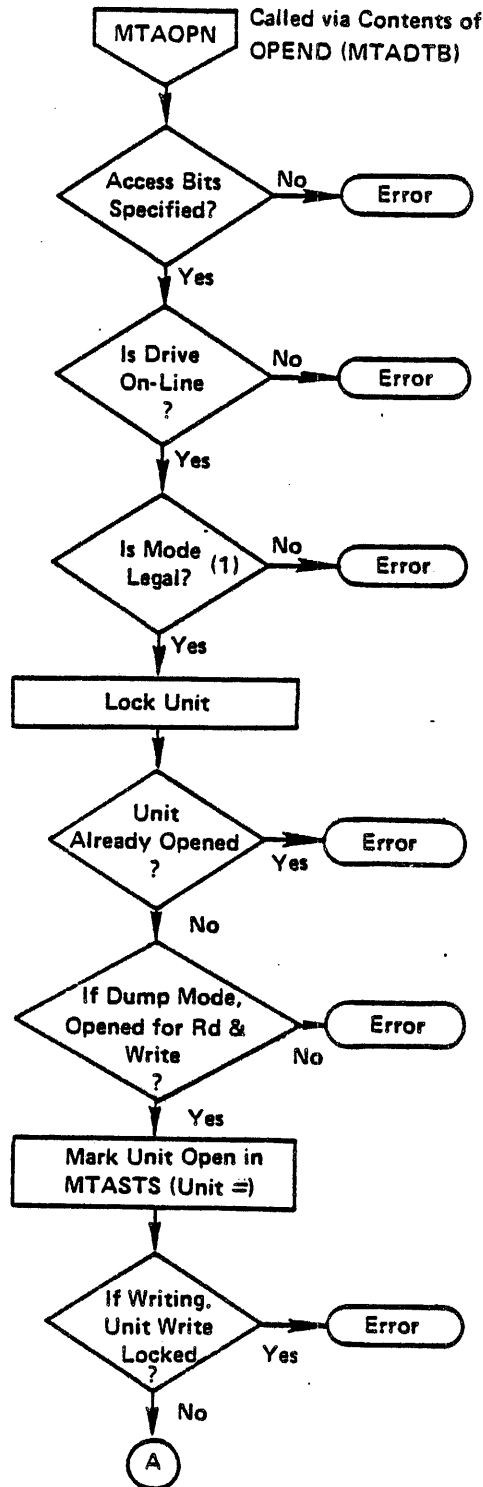
CLOSF-DISK Comments

- (1) All storage addresses placed in an index blk have the pointer type field set to immediate.

JSYS's CALLS
MTA DEPENDENT LEVEL

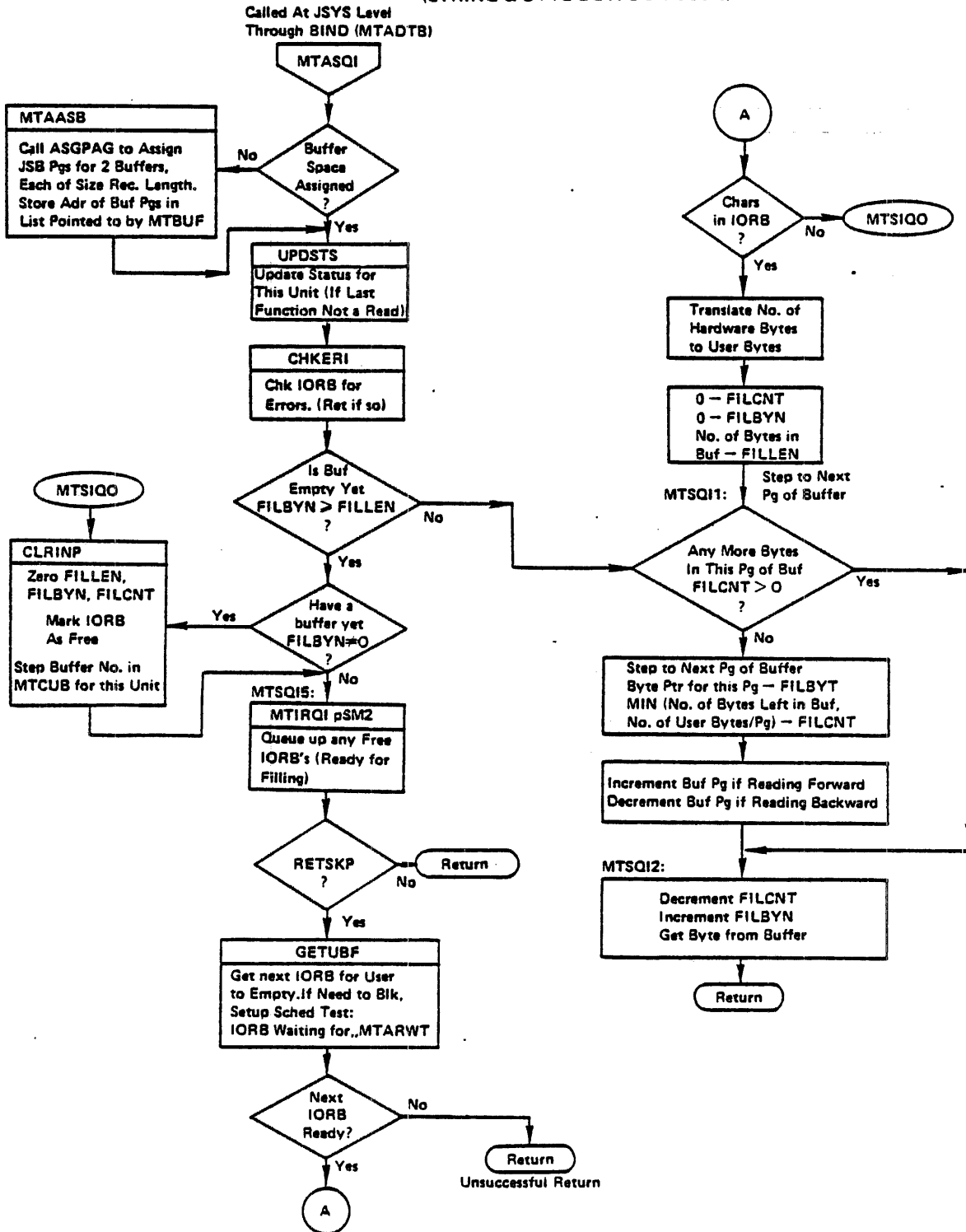
MTAOPN - Magtape Opening of a File	OM1
MTASQI - Magtape Sequential Input	SM1
MTAIRQ - Queue Up Specified IORB	SM2
MTASQO - Magtape Sequential Output	SM3
MTACLZ - Magtape Closing of a File	CM1

OPENF – MAGTAPE DEVICE DEPENDENT CODE

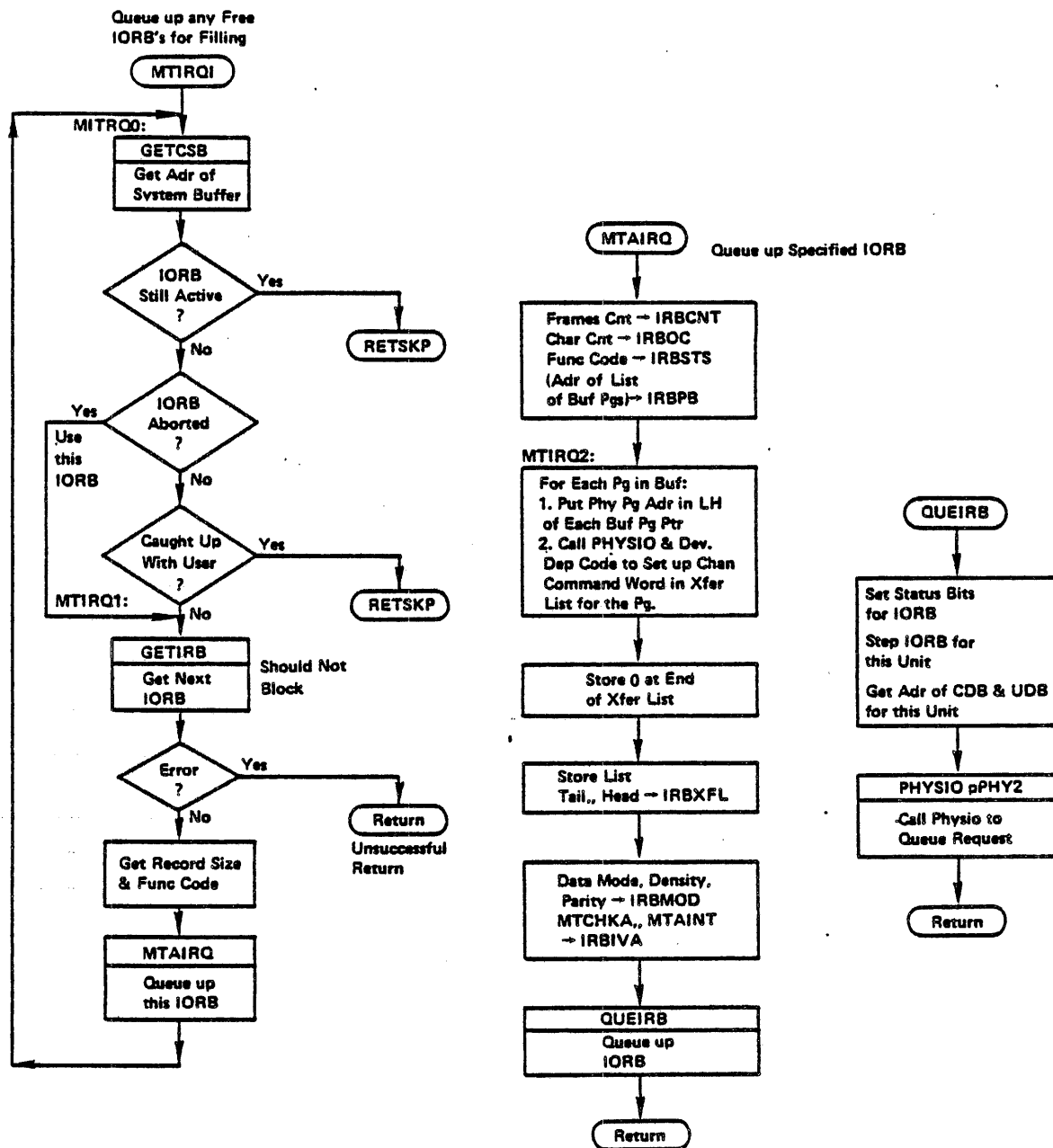


OM1

SEQUENTIAL INPUT - MTA
(STRING & BYTE DEV. DEP. CODE)

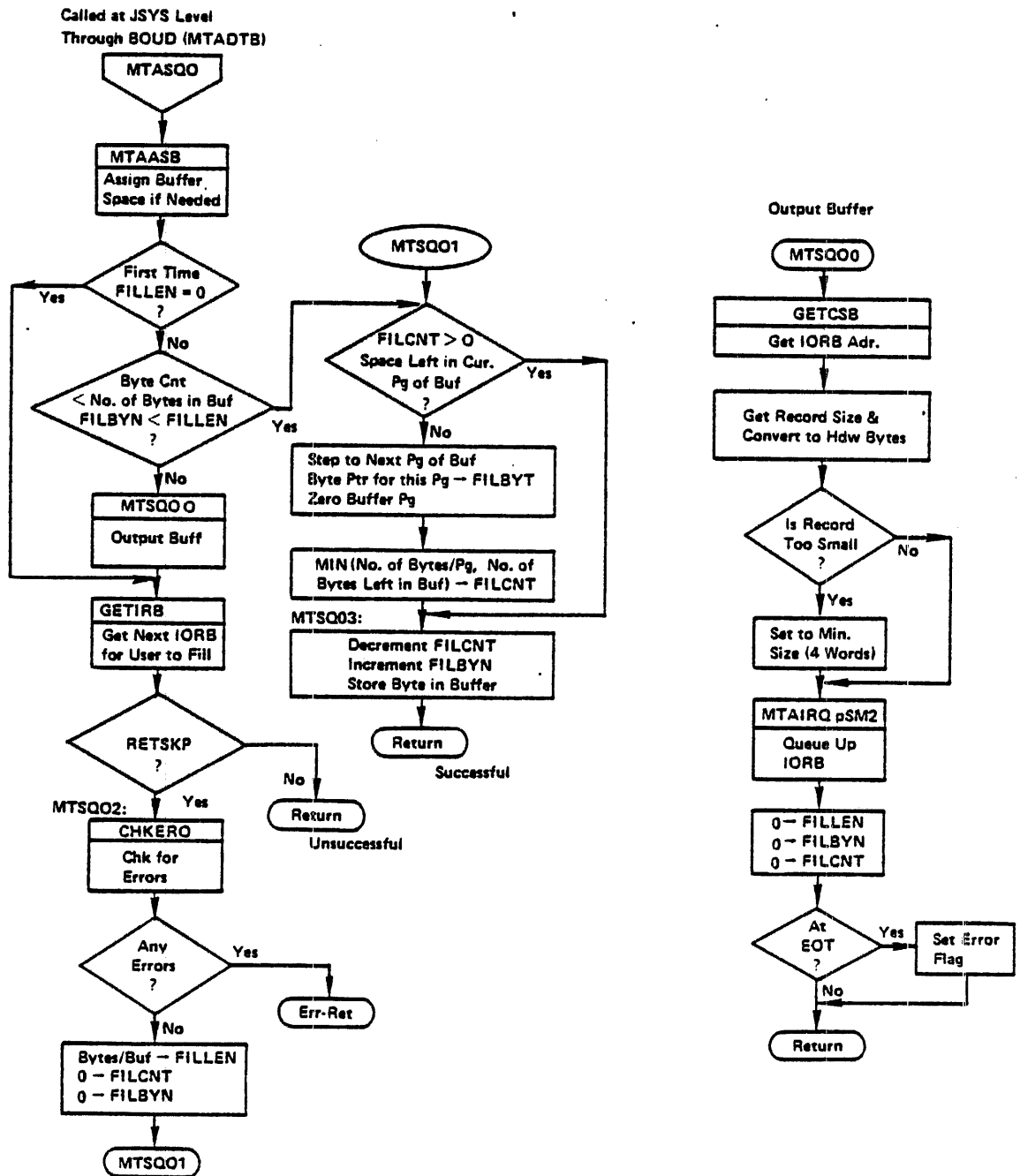


SM1



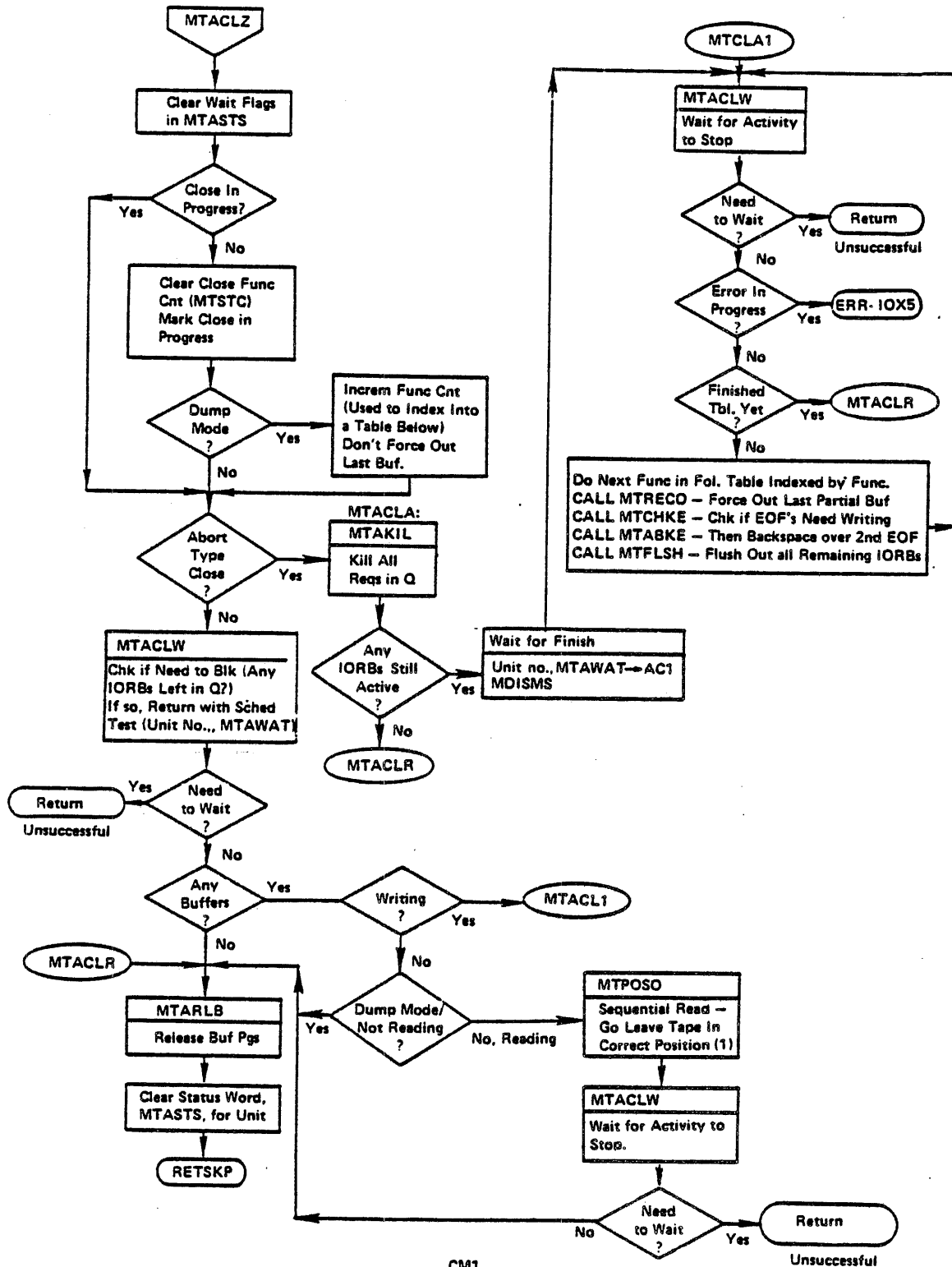
SM2

SEQUENTIAL OUTPUT – MTA STRING & BYTE DEV. DEP. CODE



SM3

CLOSF - MAGTAPE



CM1

OPENF-MAGTAPE Comments

- (1) One can open for read and write only in dump mode.
- (2) FILCNT/Count of bytes left to use in current page of
buffer.
FILLLEN/Count of bytes in buffer.
FILBYN/Buffer byte number user is referencing.

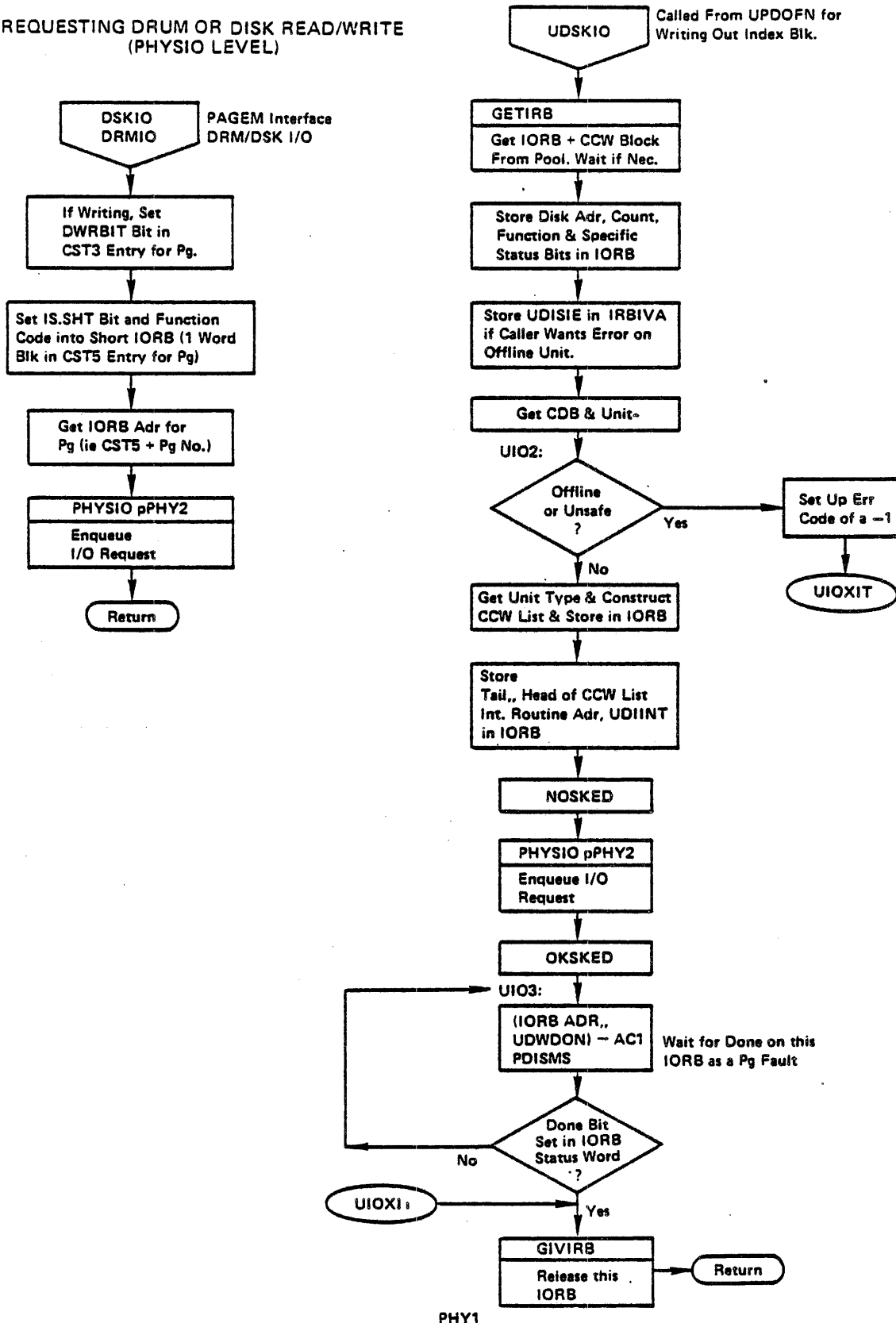
CLOSF - MAGTAPE Comments

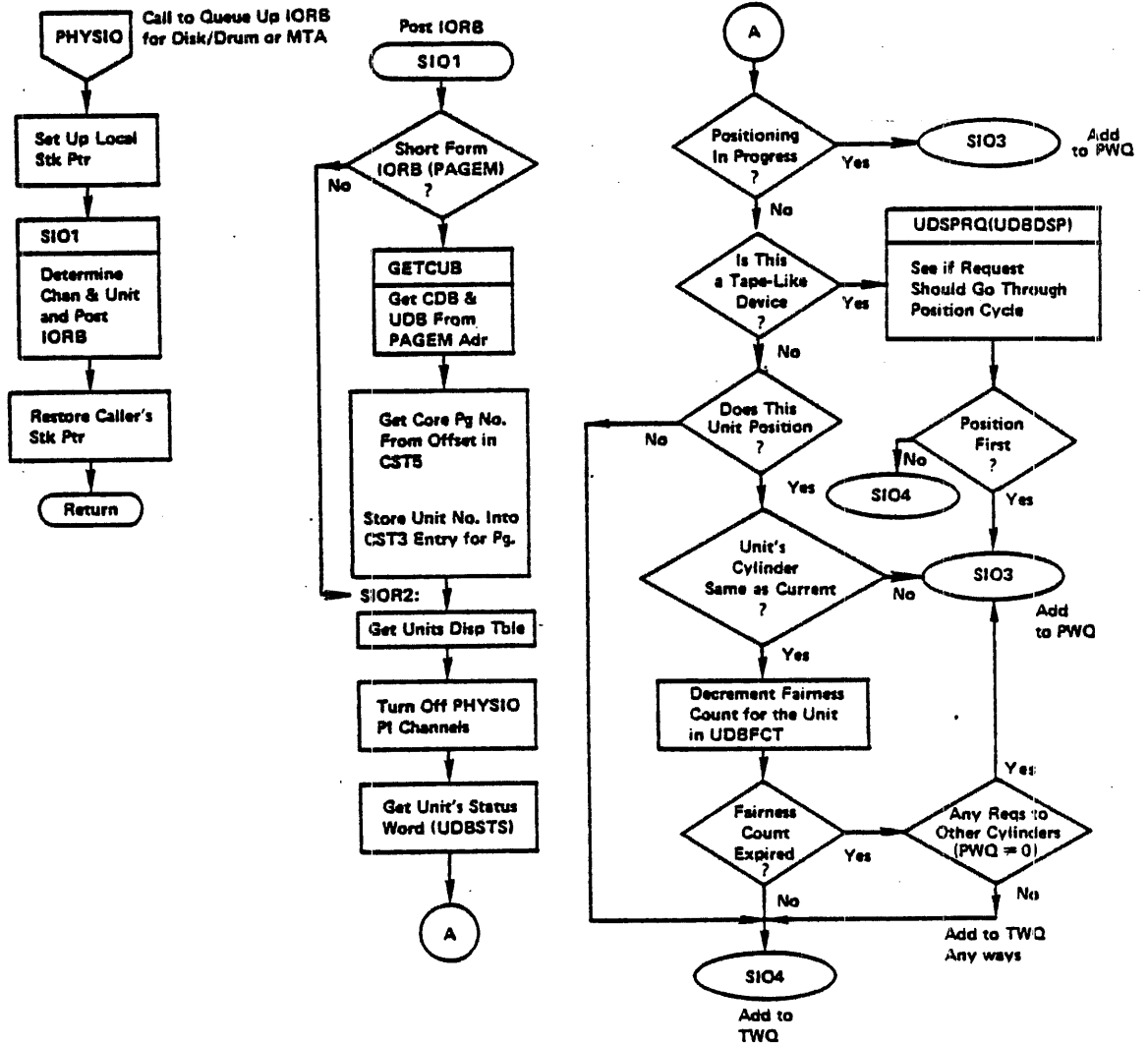
- (1) Since the monitor reads ahead, backspacing to just after last user record read may be necessary.

REQUESTING DISK/MTA I/O & INTERRUPT HANDLING FLOWCHARTS
(PHYSIO LEVEL)

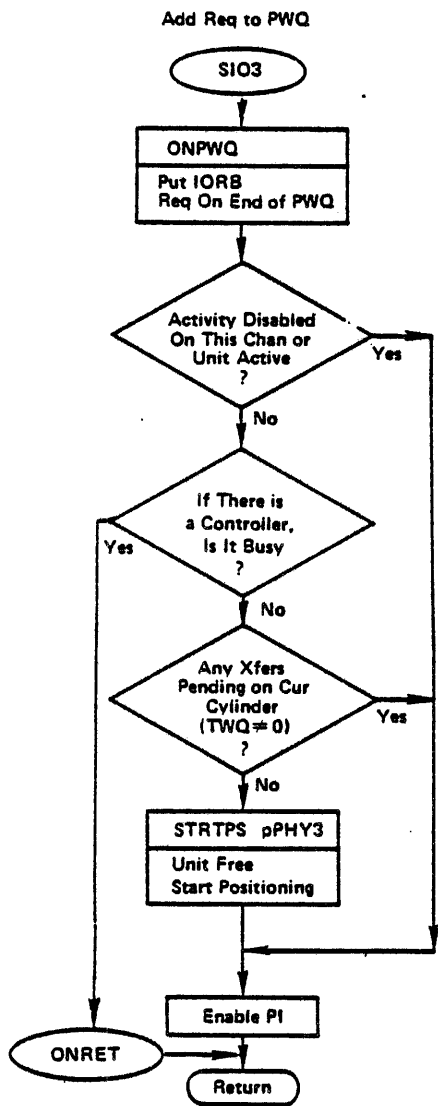
DRMIO/DSKIO/UDSKIO - Requesting Drum or Disk Read/Write	PHY1
PHYSIO - Queue Up IORB Request for Disk, Drum or Magtape	PHY2
SIO1 - Post IORB	PHY2
STRTPS - Start Unit Positioning	PHY3
STRTIO - Start Unit Transferring	PHY3a
PHYINT - Disk and Magtape Interrupt Handler	PHY4
DONIRB - Post IORB as Done	PHY5
SWPDON/UDIINT - Housekeep for Drum/Disk Done	PHY8
MTAINT - Housekeep for Magtape Done	PHY9
SCHSEK - Schedule "Best" Seek Request	PHY6
SCHXFR - Schedule "Best" Transfer Request	PHY7

REQUESTING DRUM OR DISK READ/WRITE
(PHYSIO LEVEL)

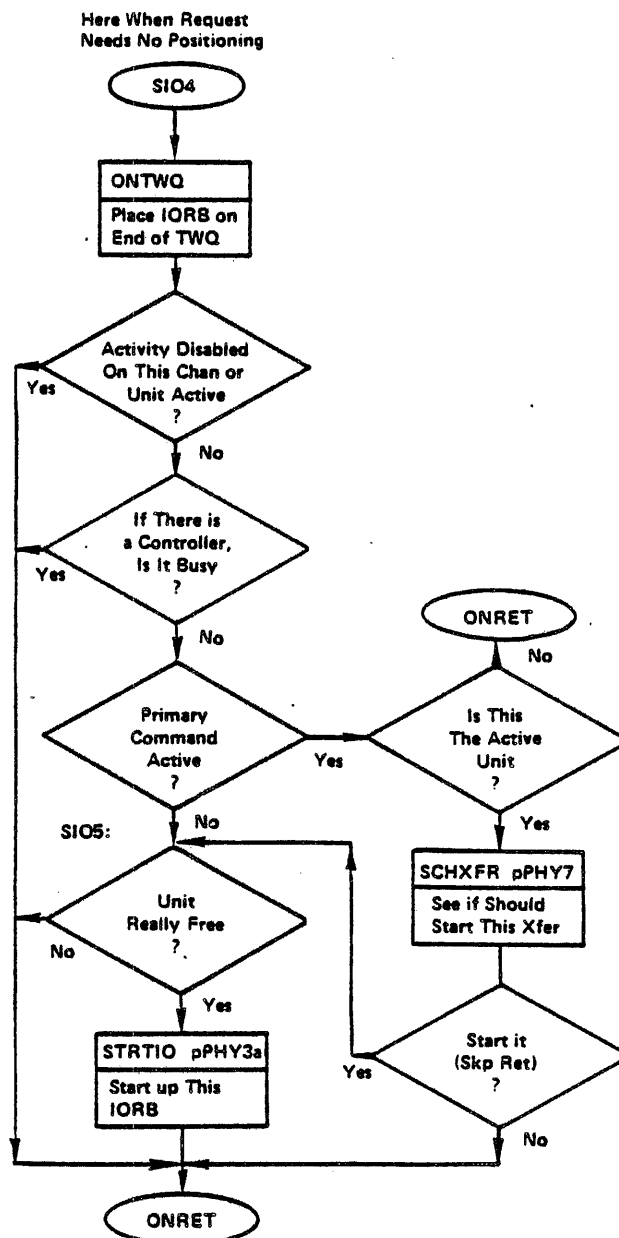




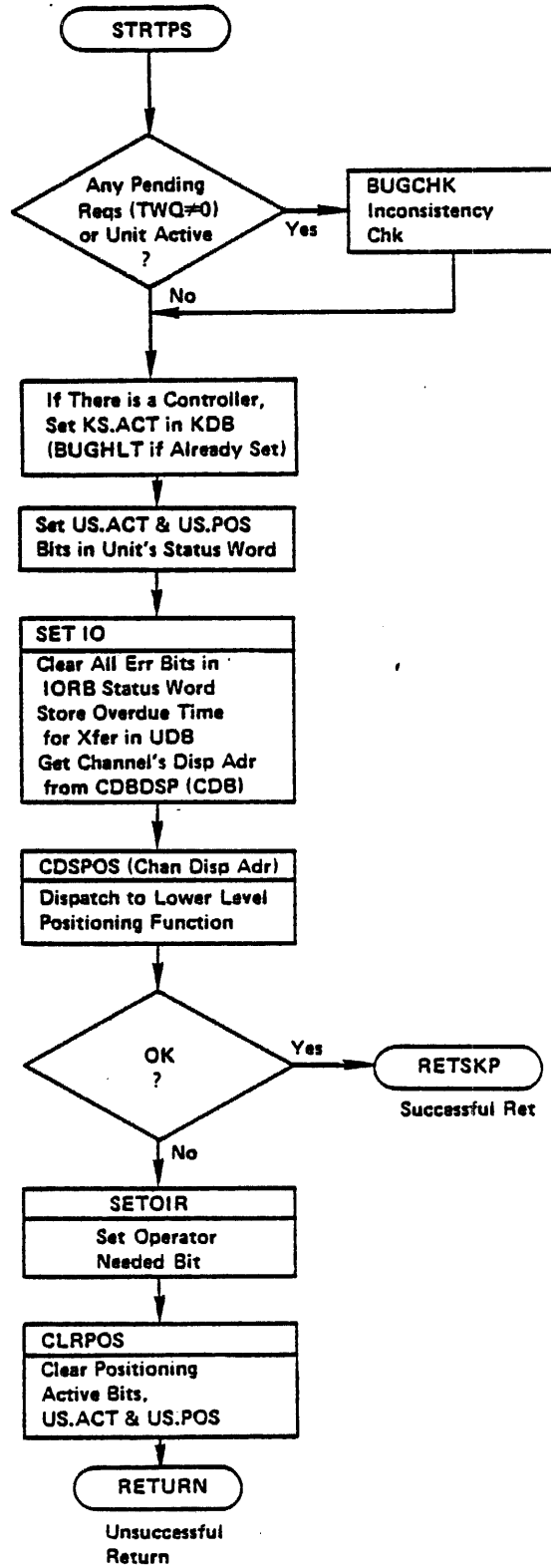
PHY2

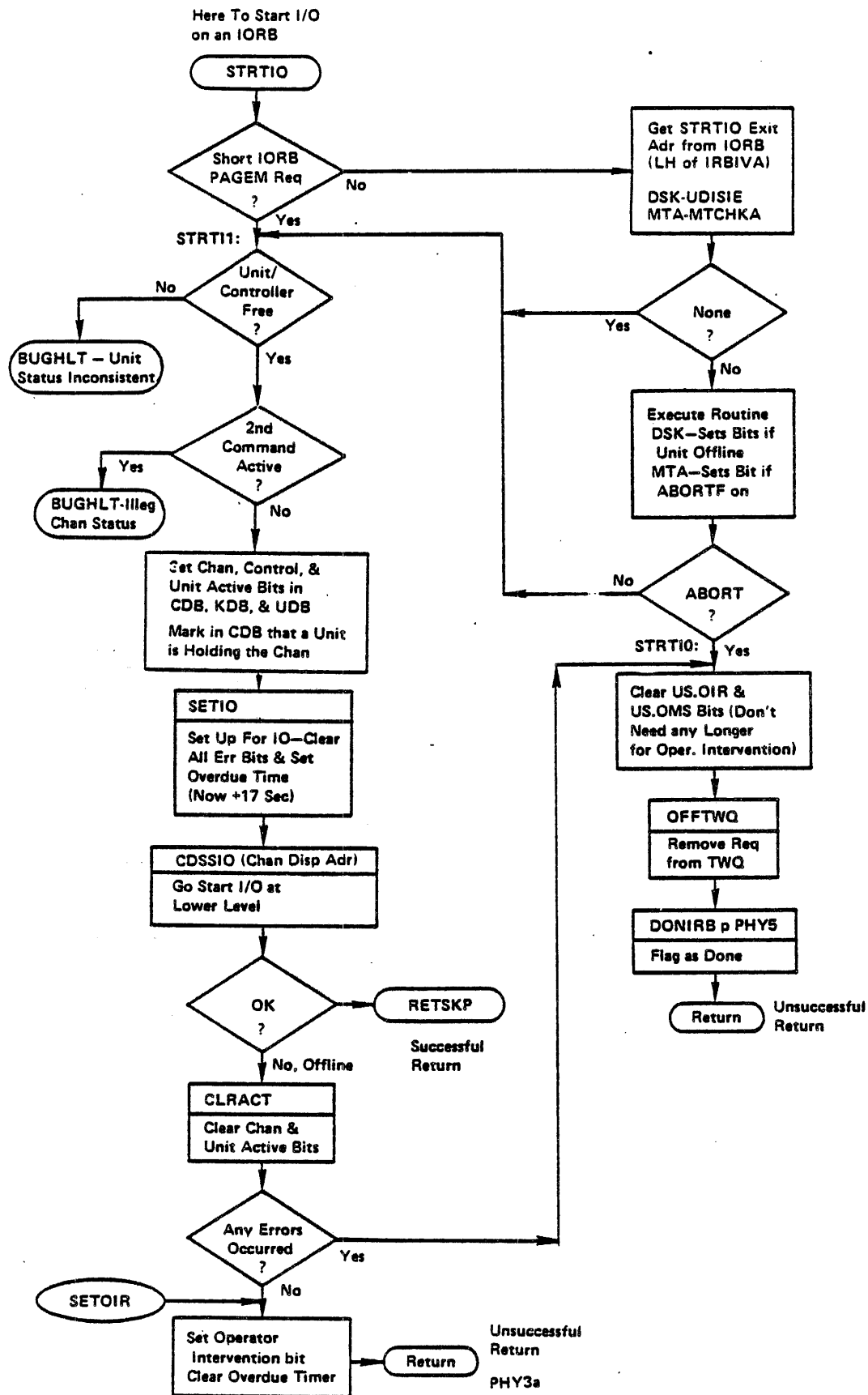


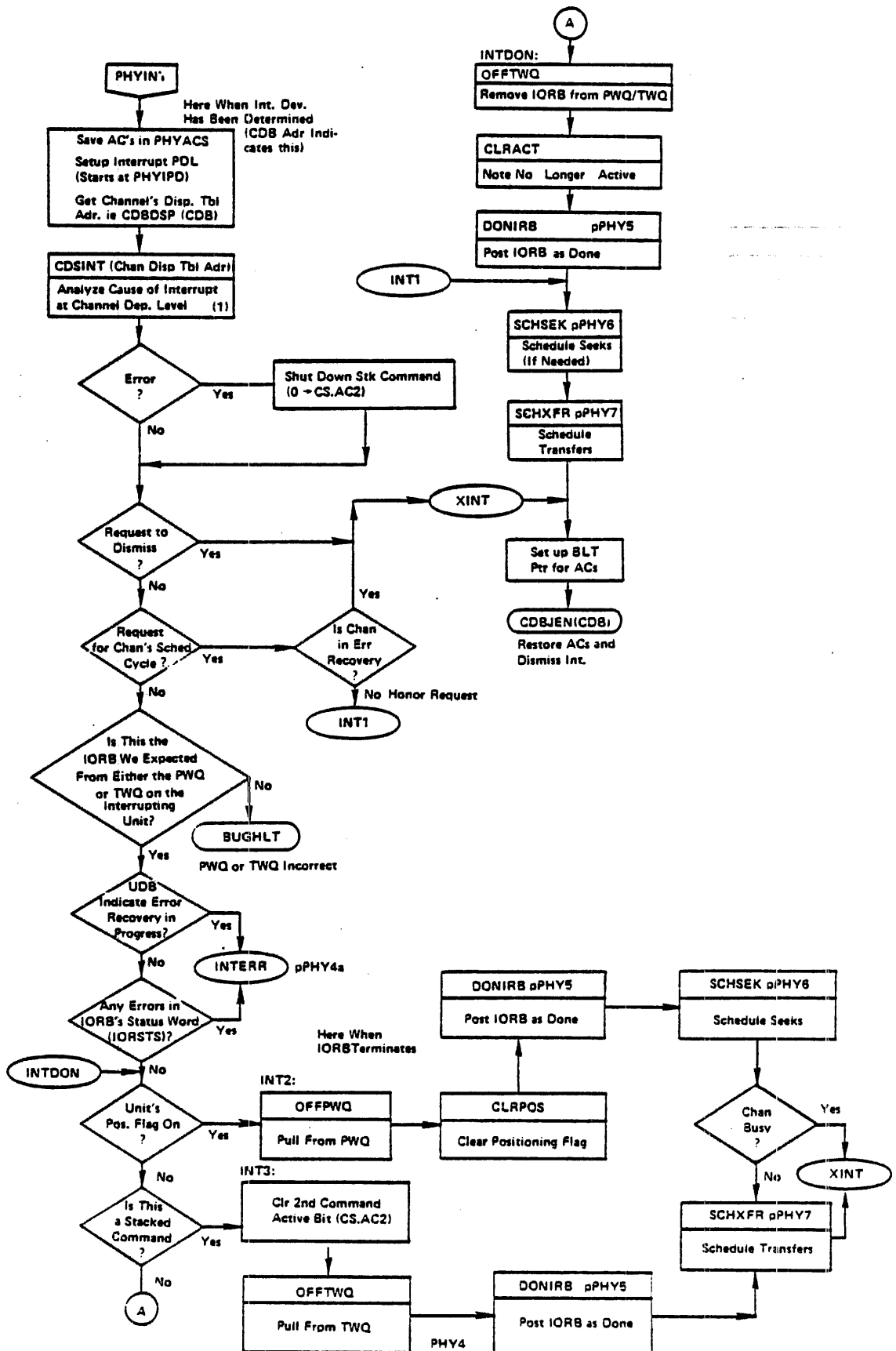
PHY2a

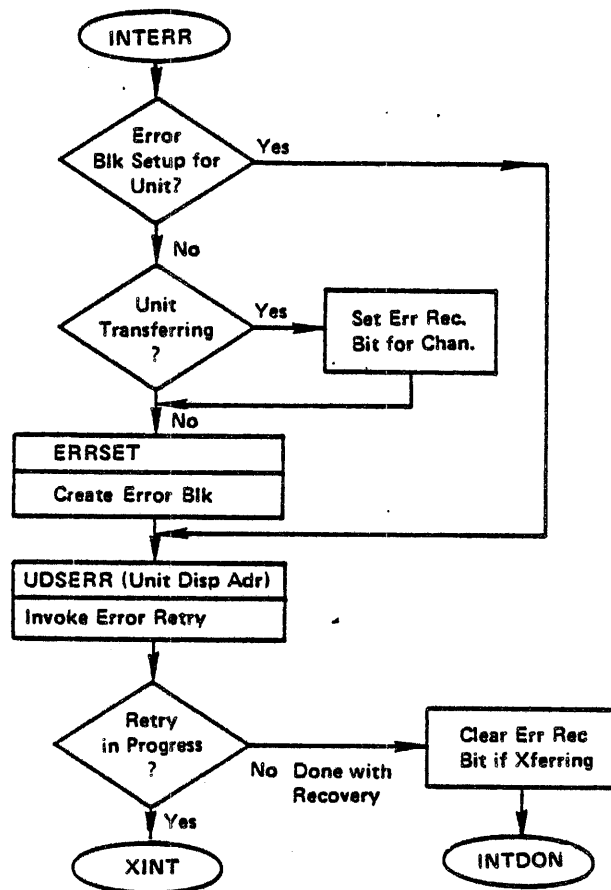


Here to Start Positioning
for an IORB

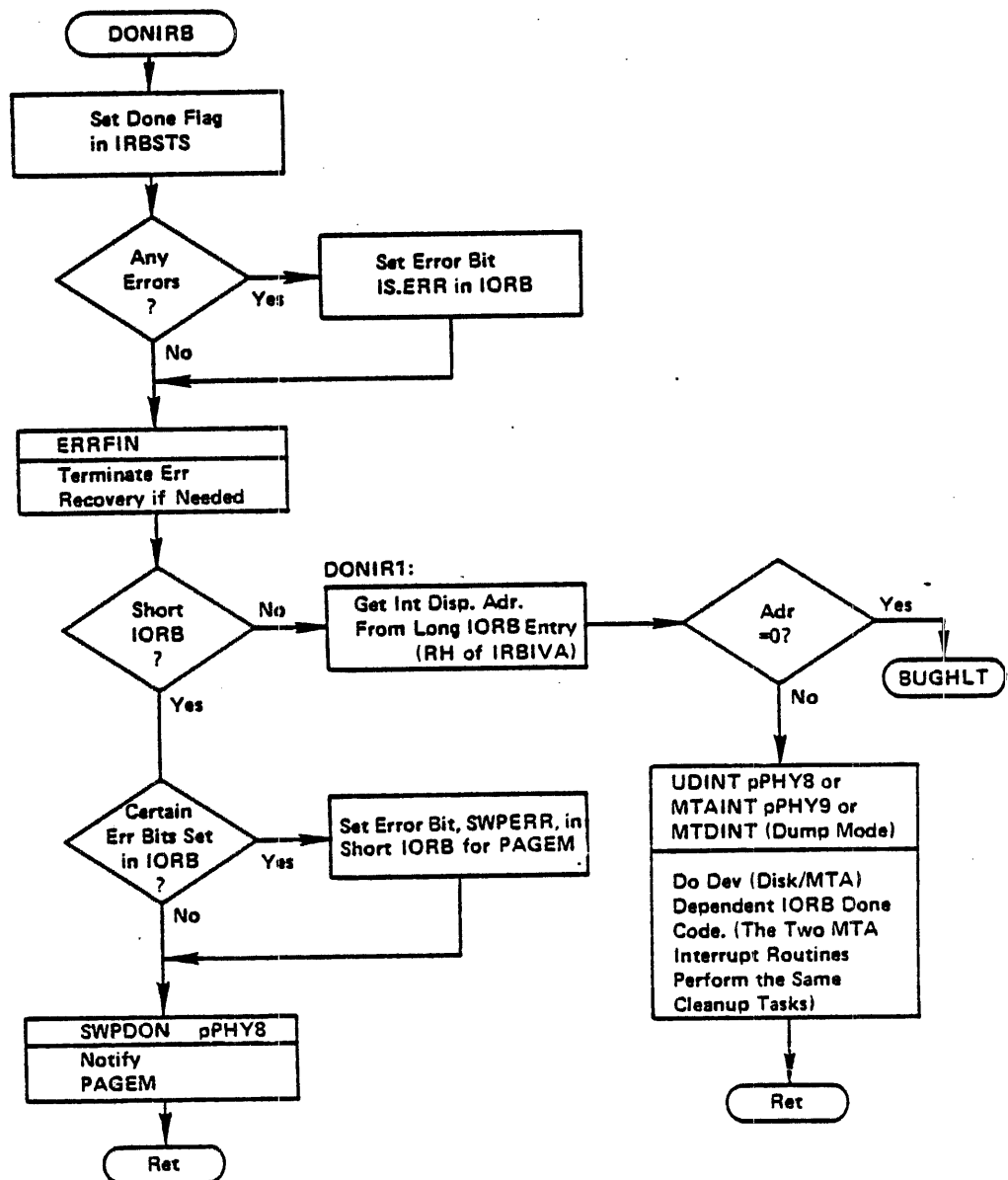




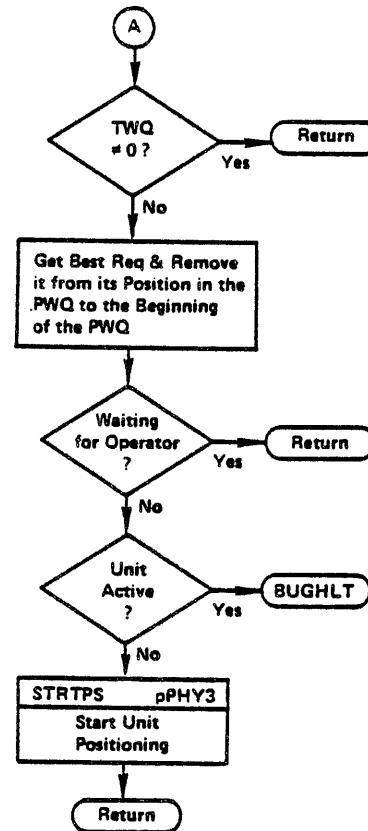
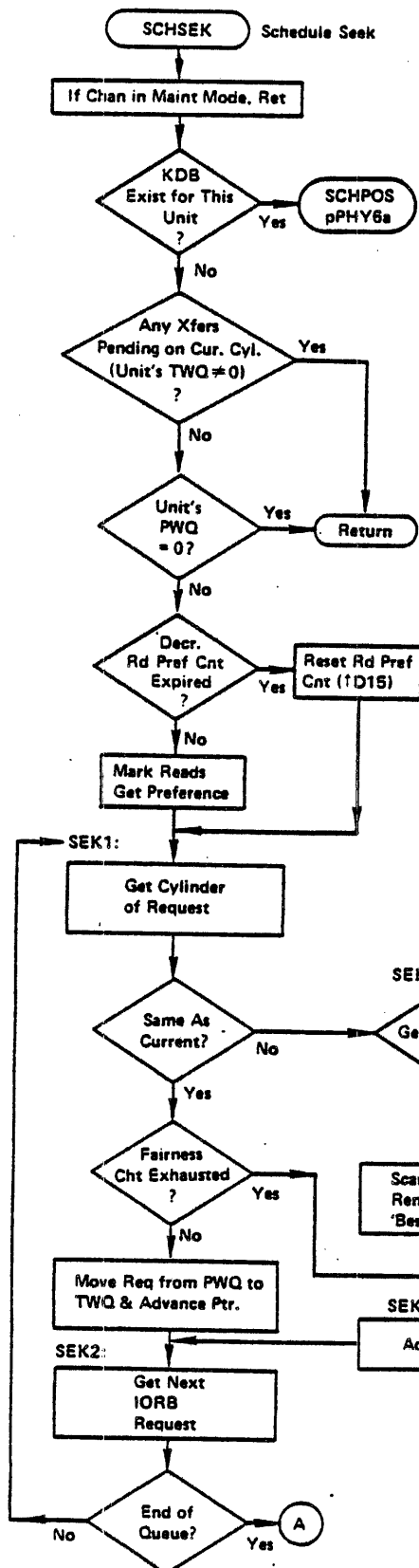




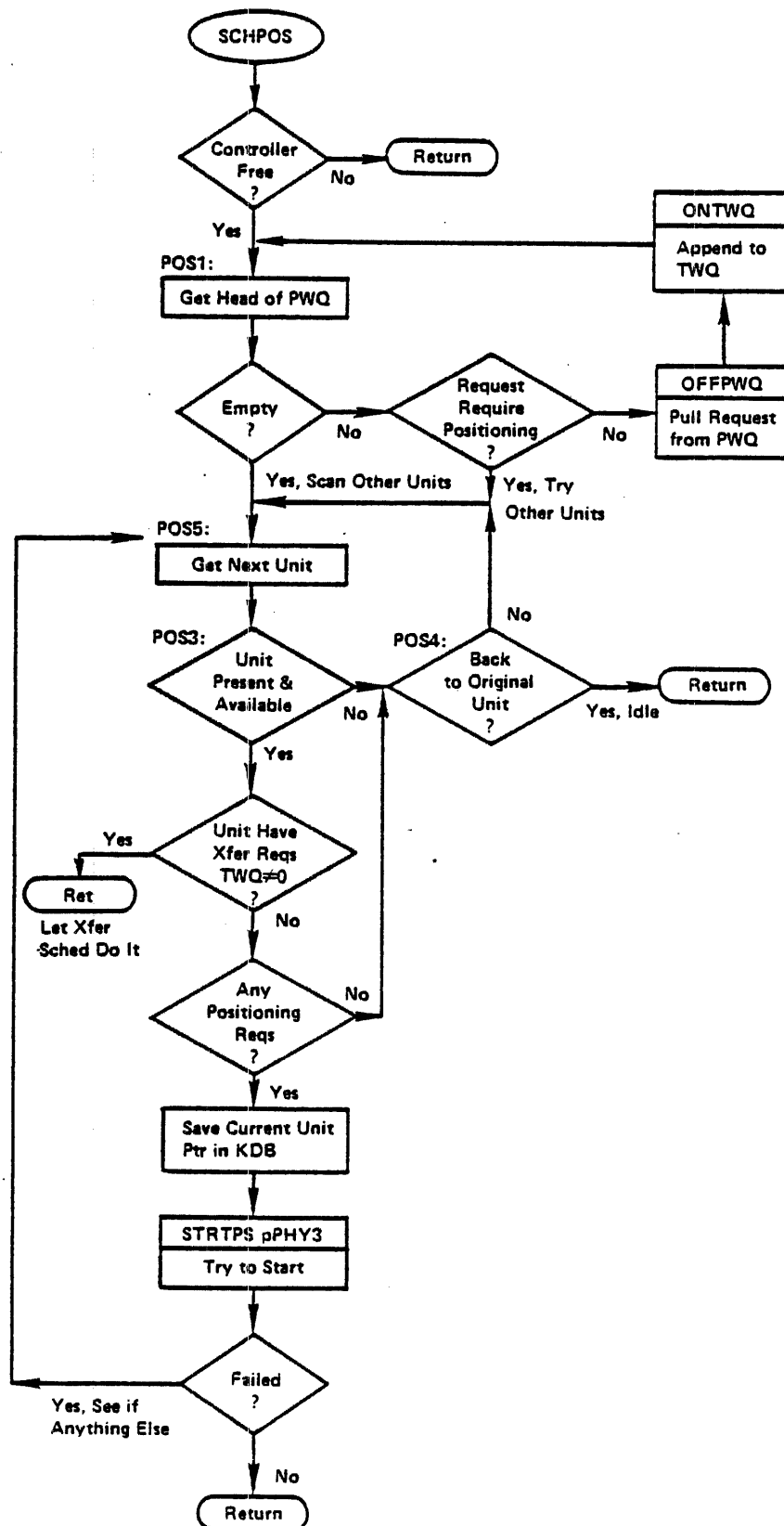
Here to Post an IORB Complete



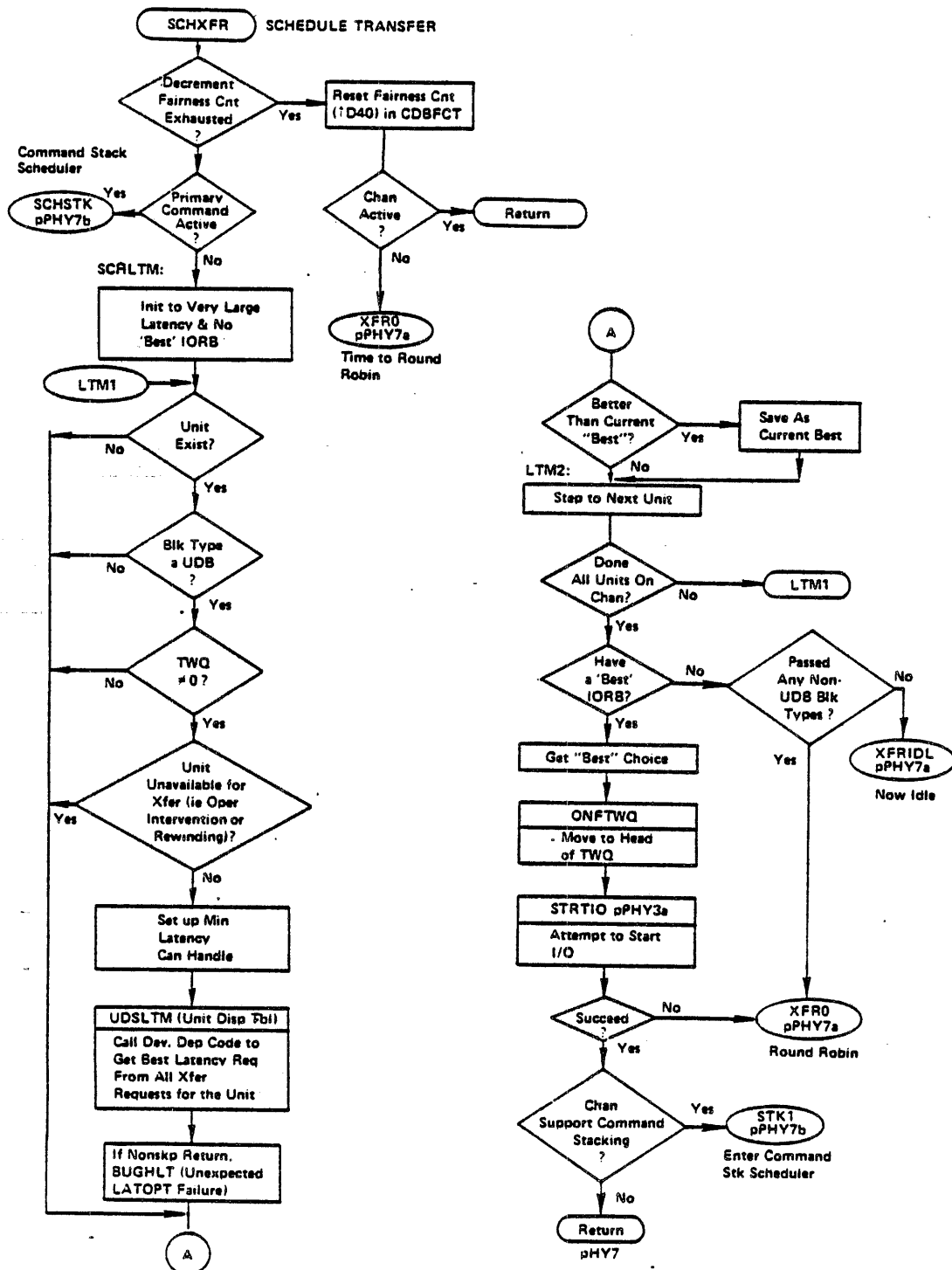
PHY5

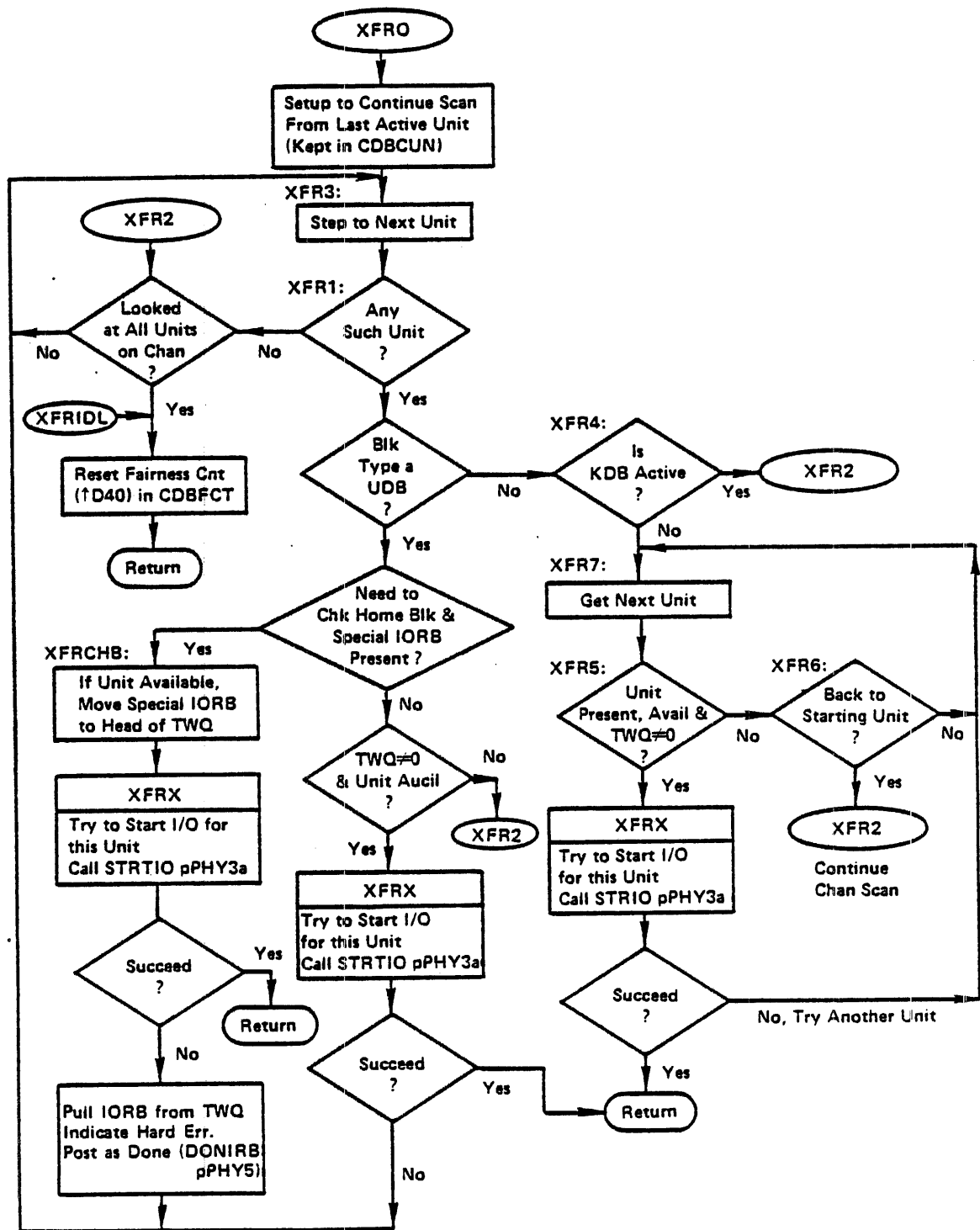


PHY6

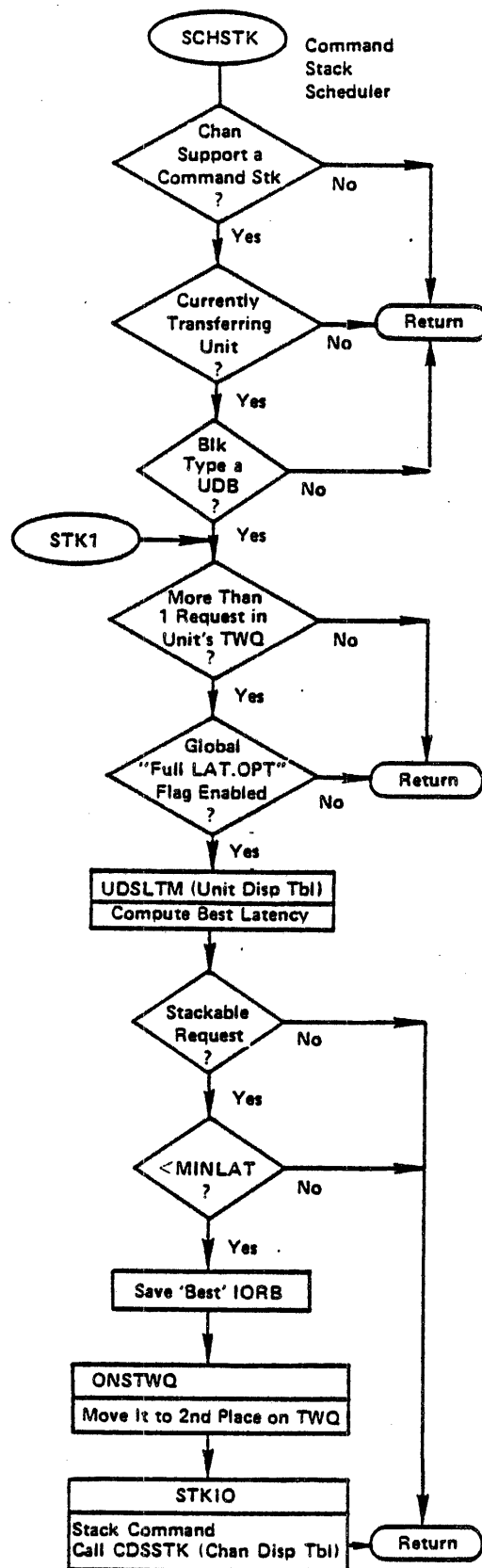


PHY6a



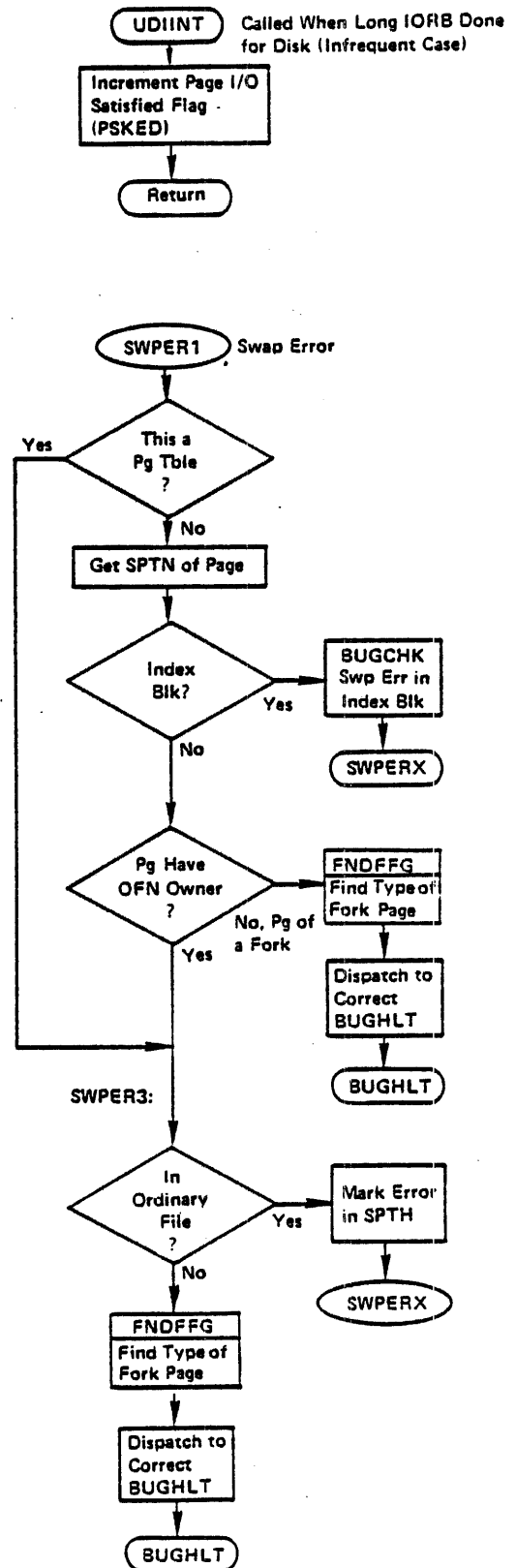
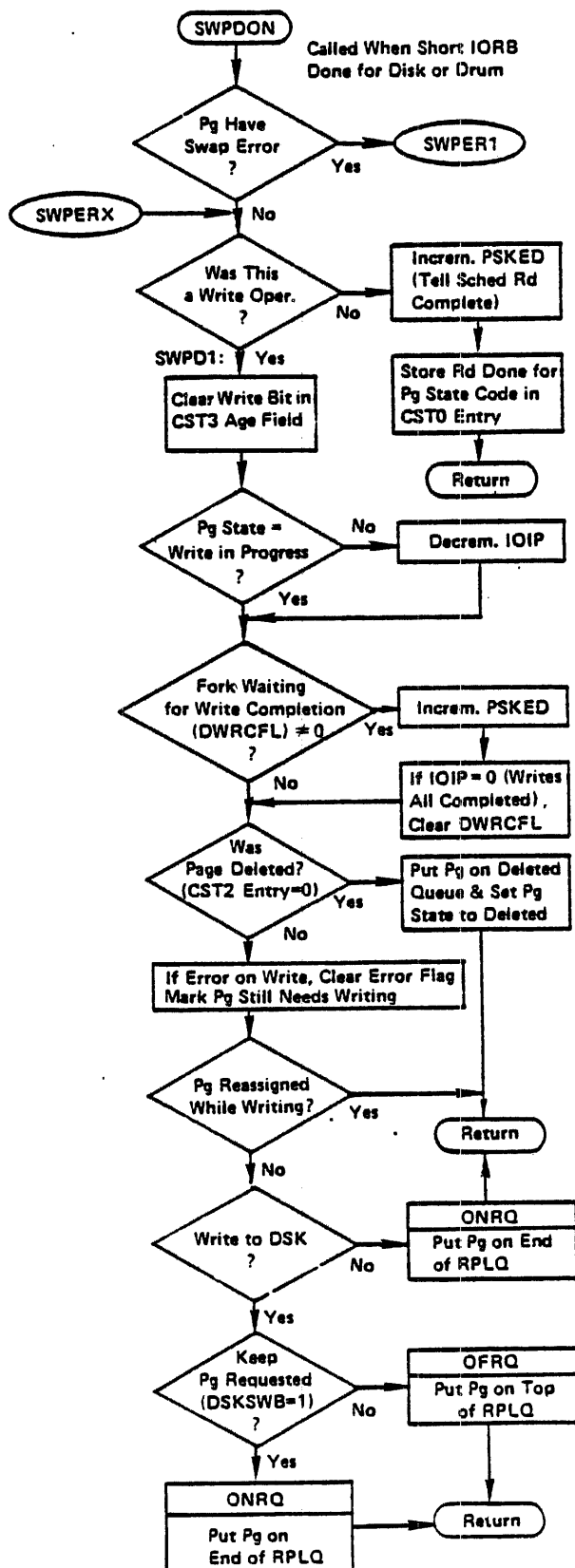


PHY7a



PHY7b

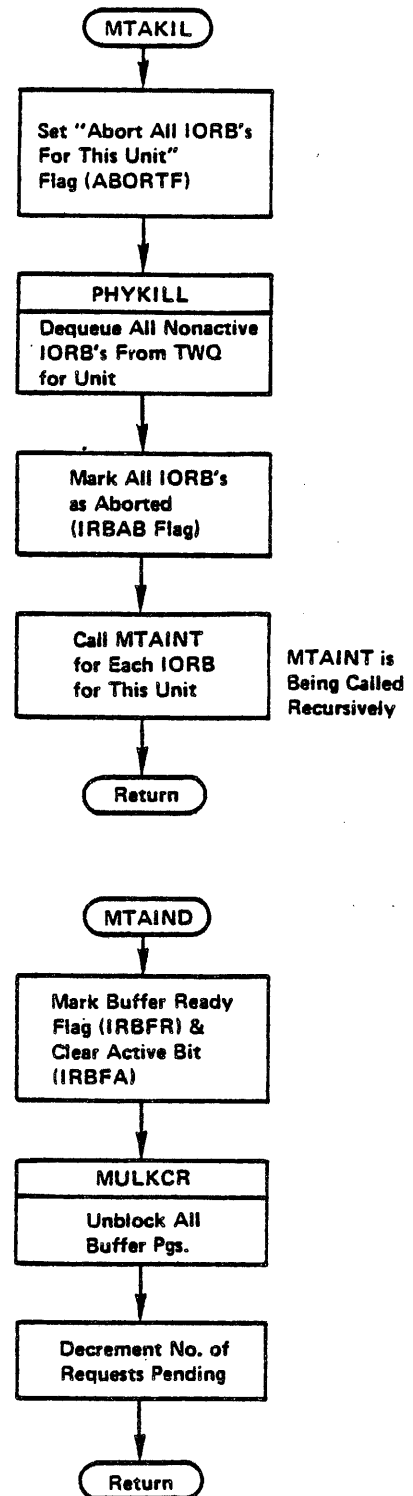
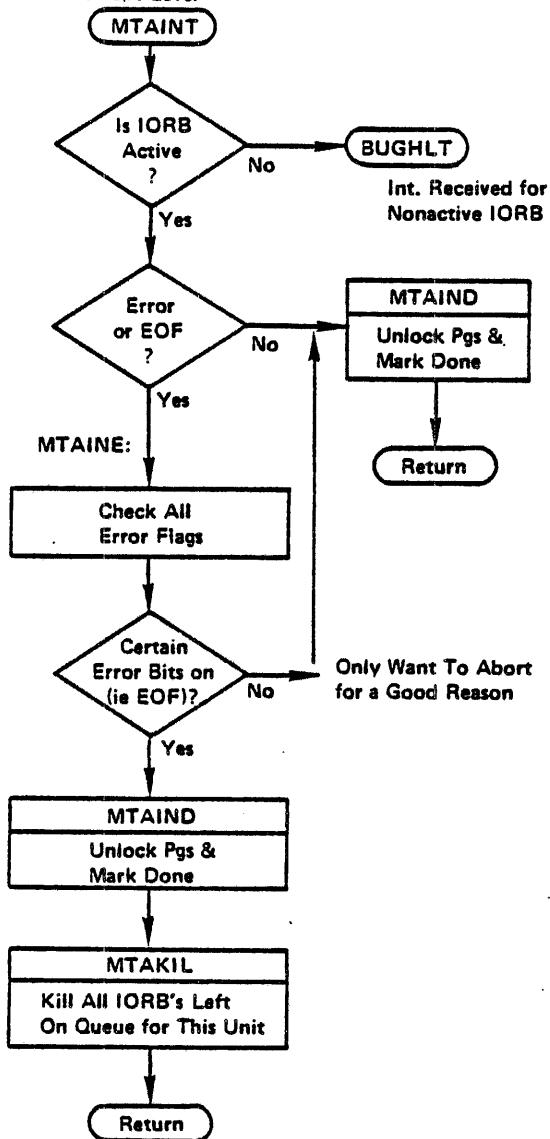
"INTERRUPT DONE" DSK/DRUM DEPENDENT CODE



PHY8

"INTERRUPT DONE" MAGTAPE DEPENDENT CODE

Called When Non-Dump Xfer Done For
MTA At Interrupt Level



PHY9

Requesting DISK/MTA I/O Comments

SI01

- (1) The algorithm for queuing up a MTA request is:

If the request requires positioning, append the request to the PWQ.

If the request requires no positioning (i.e., Read/Write Forward or Read Reverse) append the request to the TWQ only if the PWQ is empty. Otherwise, append it to the PWQ.

DSK/MTA Interrupt Handling Comments

PHYINT

- (1) The channel dependent routine (RH2INT for RH20s) is called to analyze the interrupt. Lower level routines called by RH2INT (i.e., Unit dependent routines) return an argument in AC, P4, to PHYINT to indicate whether to dismiss the interrupt ($P4 = 0$), to schedule another channel cycle right away ($P4 < 0$) or to housekeep the current request ($P4 > 0$) before scheduling another channel cycle. The channel dependent routine also records error information so that PHYINT can see if error recovery is in progress or should be started.

The request to dismiss ($P4 = 0$) is invoked for example when the done flag is on and the channel is not occupied. The request for an immediate channel cycle ($P4 < 0$) is made when a positioning done interrupt has occurred and there is no transfer in progress. Transfer Done requests will require further housekeeping ($P4 > 0$) by PHYINT before scheduling another channel cycle.

SCHSEK

- (2) The scan algorithm with read preference in effect performs as follows:

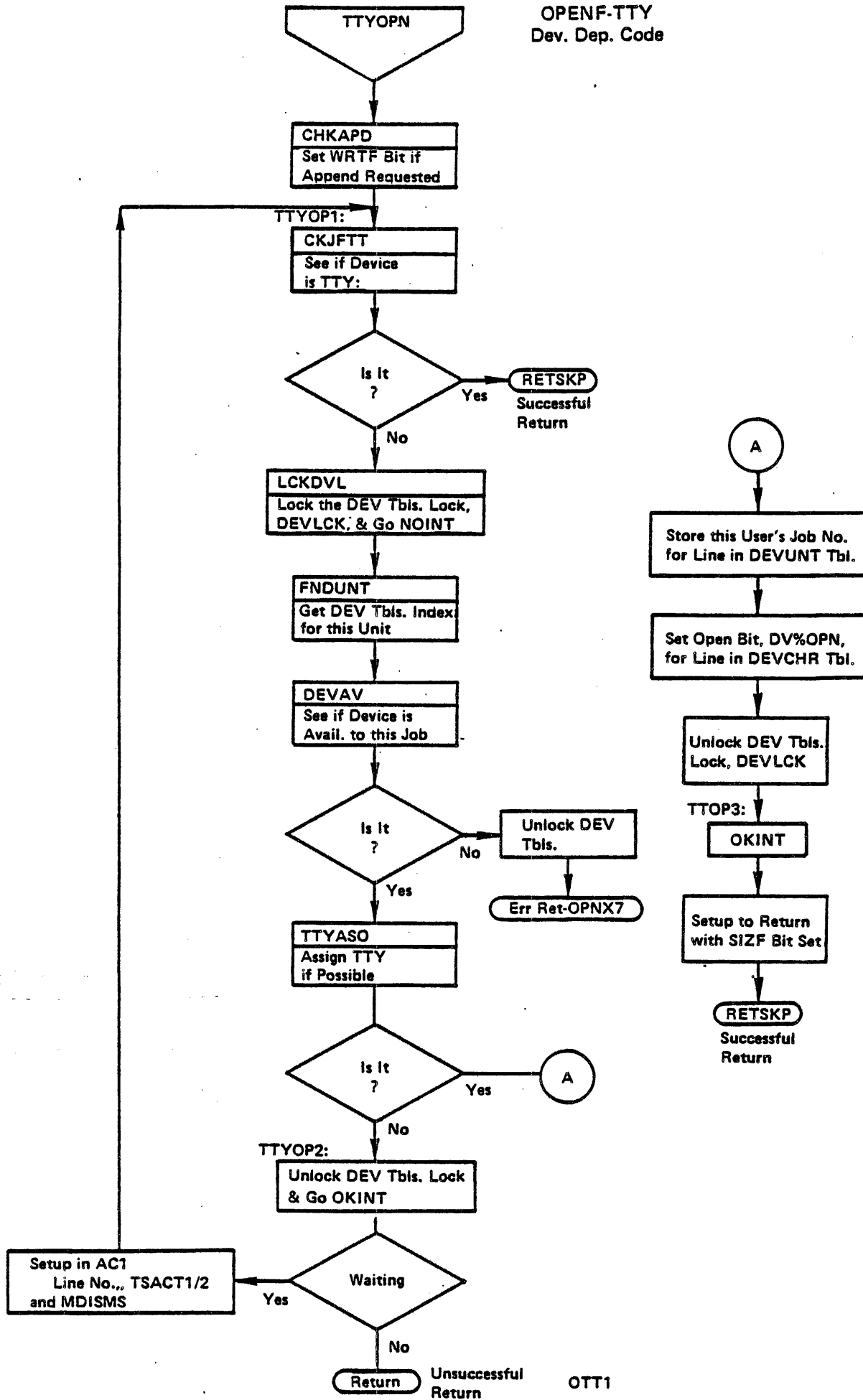
Take the next higher-numbered cylinder read request from the current cylinder. If none, take the next higher-numbered cylinder (write) request from the current cylinder.

If none, take the lowest numbered cylinder read request from the current cylinder. If none, take the lowest numbered cylinder (write) request from the current cylinder.

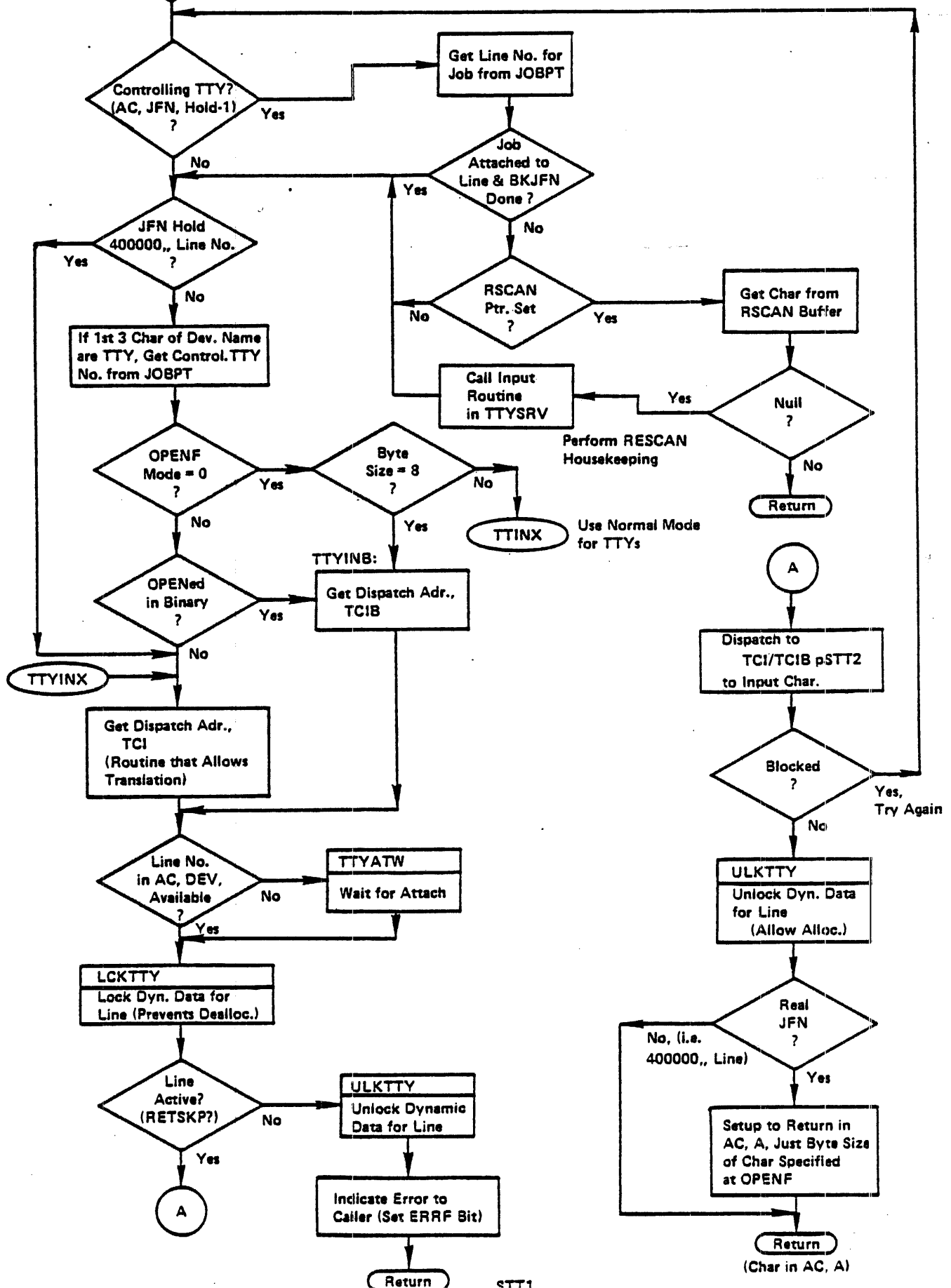
JSYS CALL FLOWCHARTS
TTY DEPENDENT LEVEL

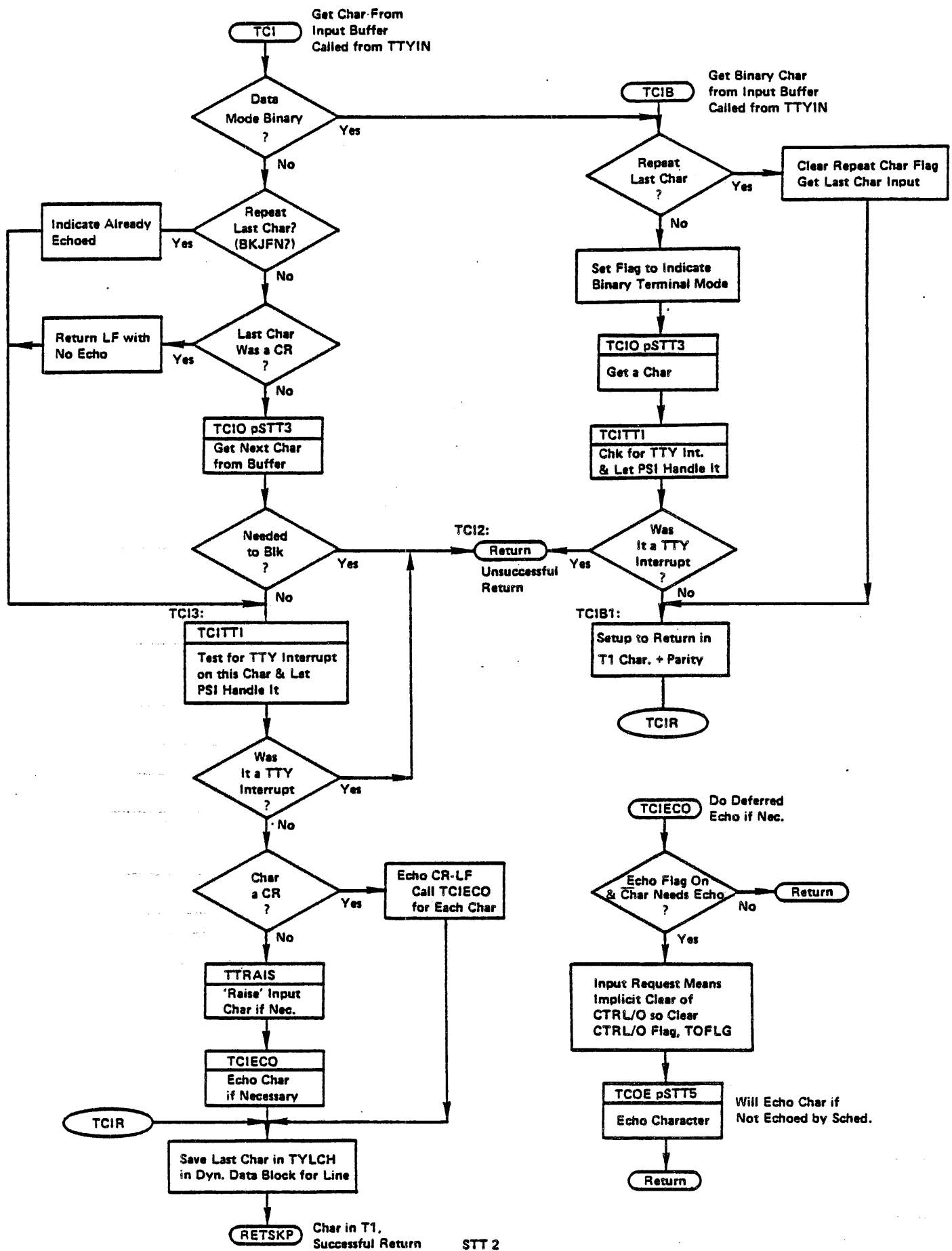
TTYOPN - Teletype Opening of a File	OTT1
TTYIN - Teletype Sequential Input	STT1
TCI/TCIB - Get Character from Line's Input Buffer	STT2
TCIO - Get a Character	STT3
TCOE - Echo Character	STT5
TTYOUT - Teletype Sequential Output	STT4
TCO/TCOB - 1st Level: Output a Single Character - Translate According to Fork's Specification	STT5
TCOY - 2nd Level: Do Links & Formats for a Particular Device	STT6
TCOUT - 3rd Level: Do Buffering and Output 1 Character	STT7
TTSND - Send Character to Line	STT8
TTYCLS - Teletype Closing of a File	CLTT1

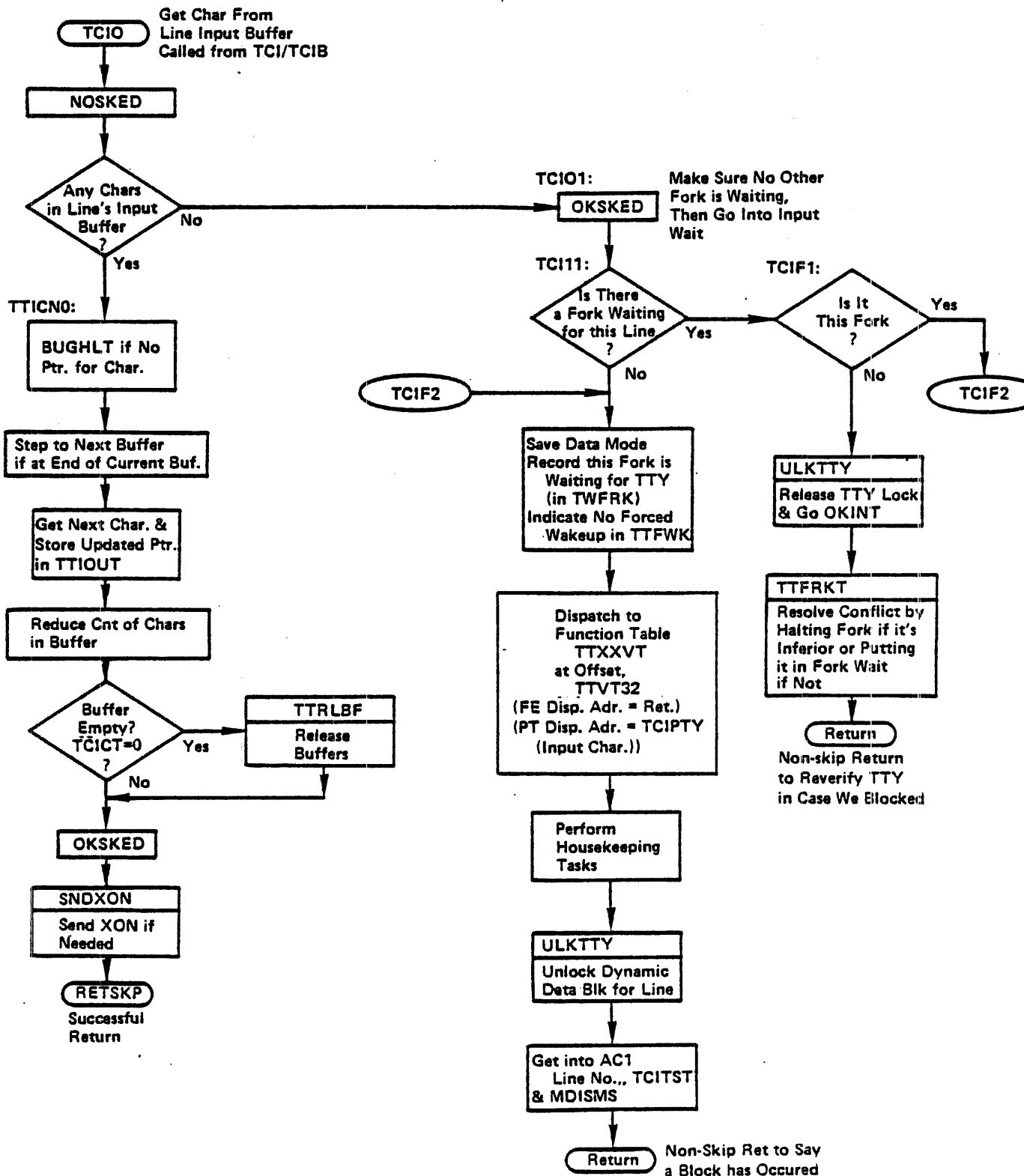
OPENF-TTY
Dev. Dep. Code

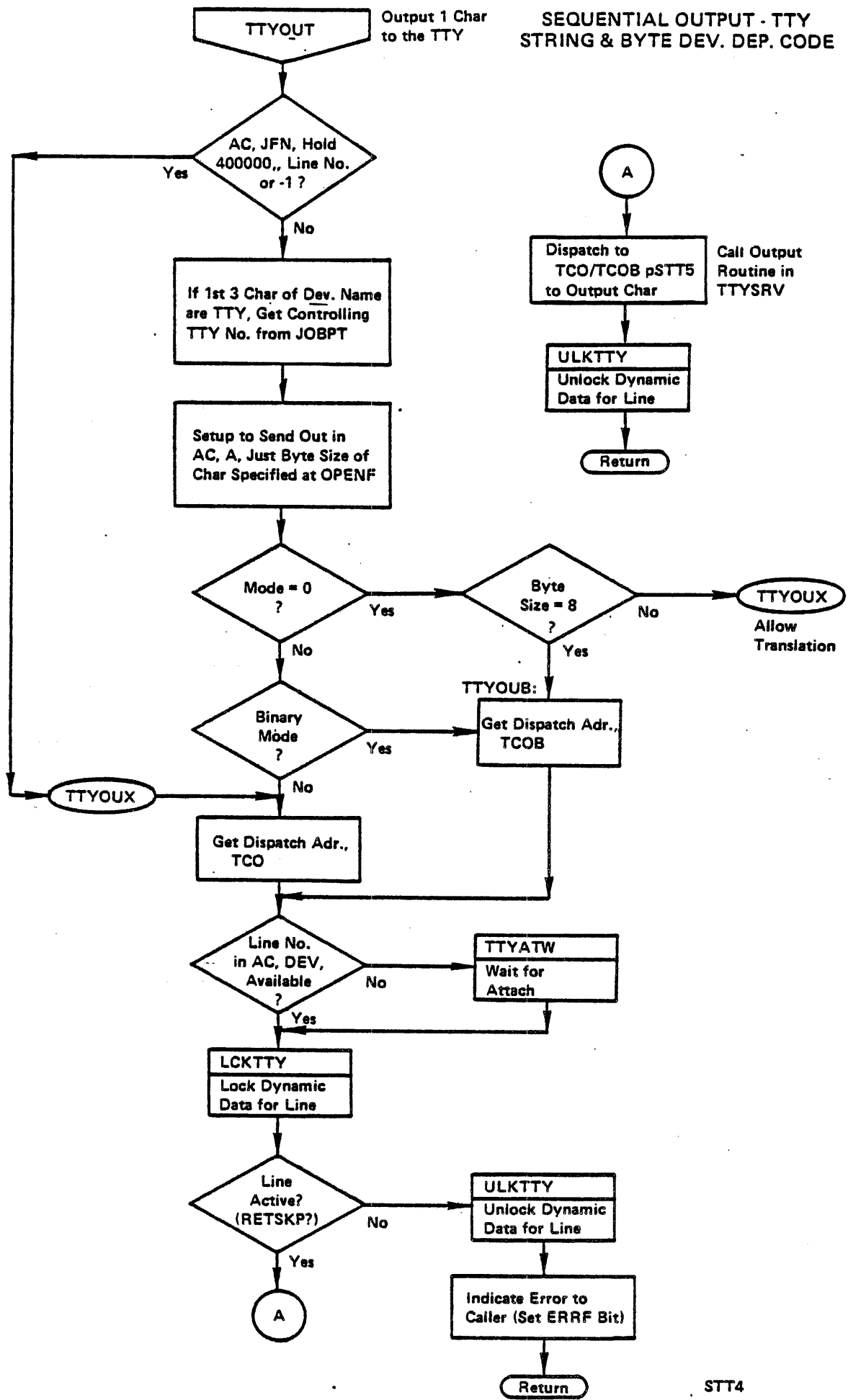


TTYIN Get 1 Input Char for the Line

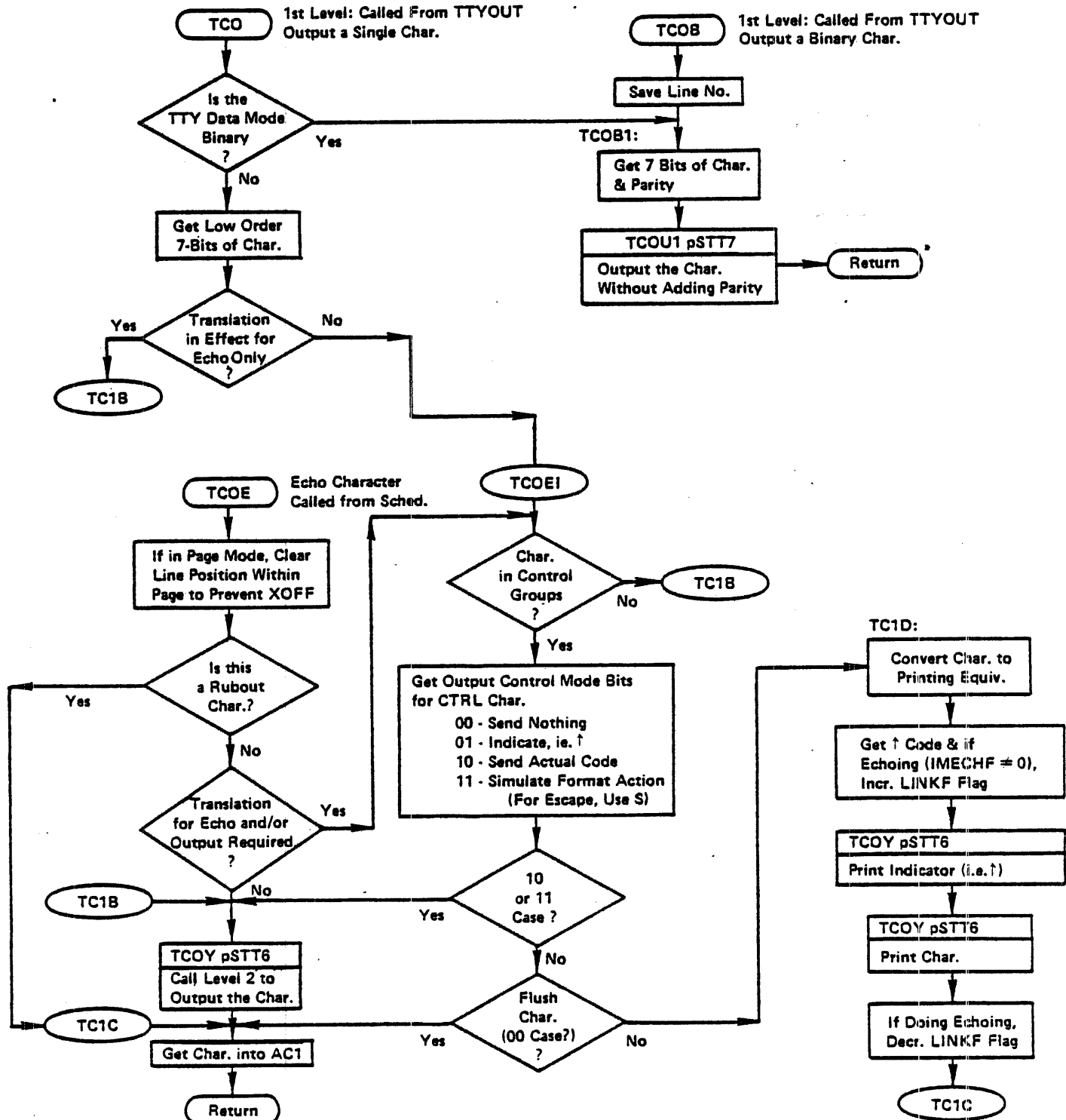




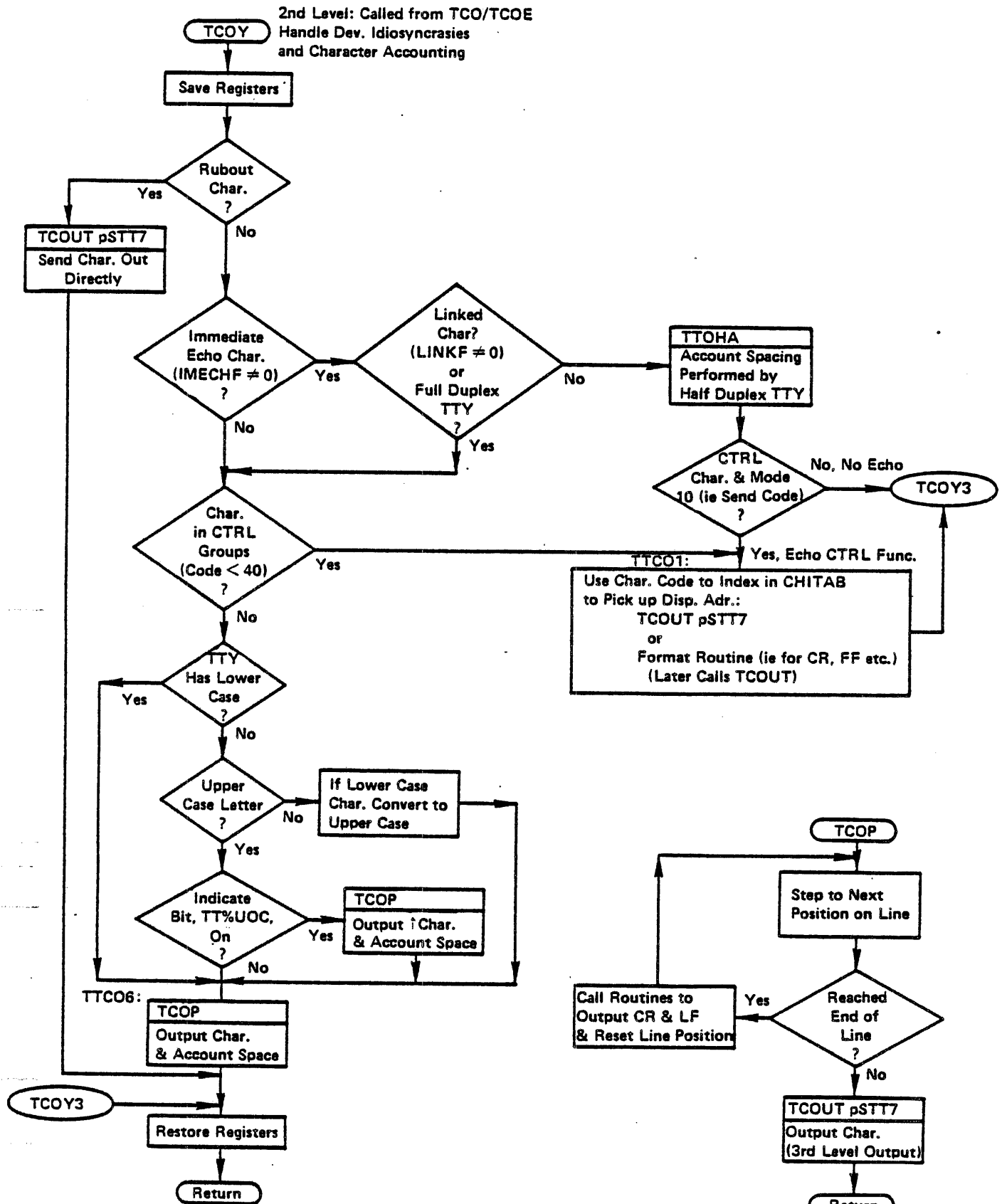




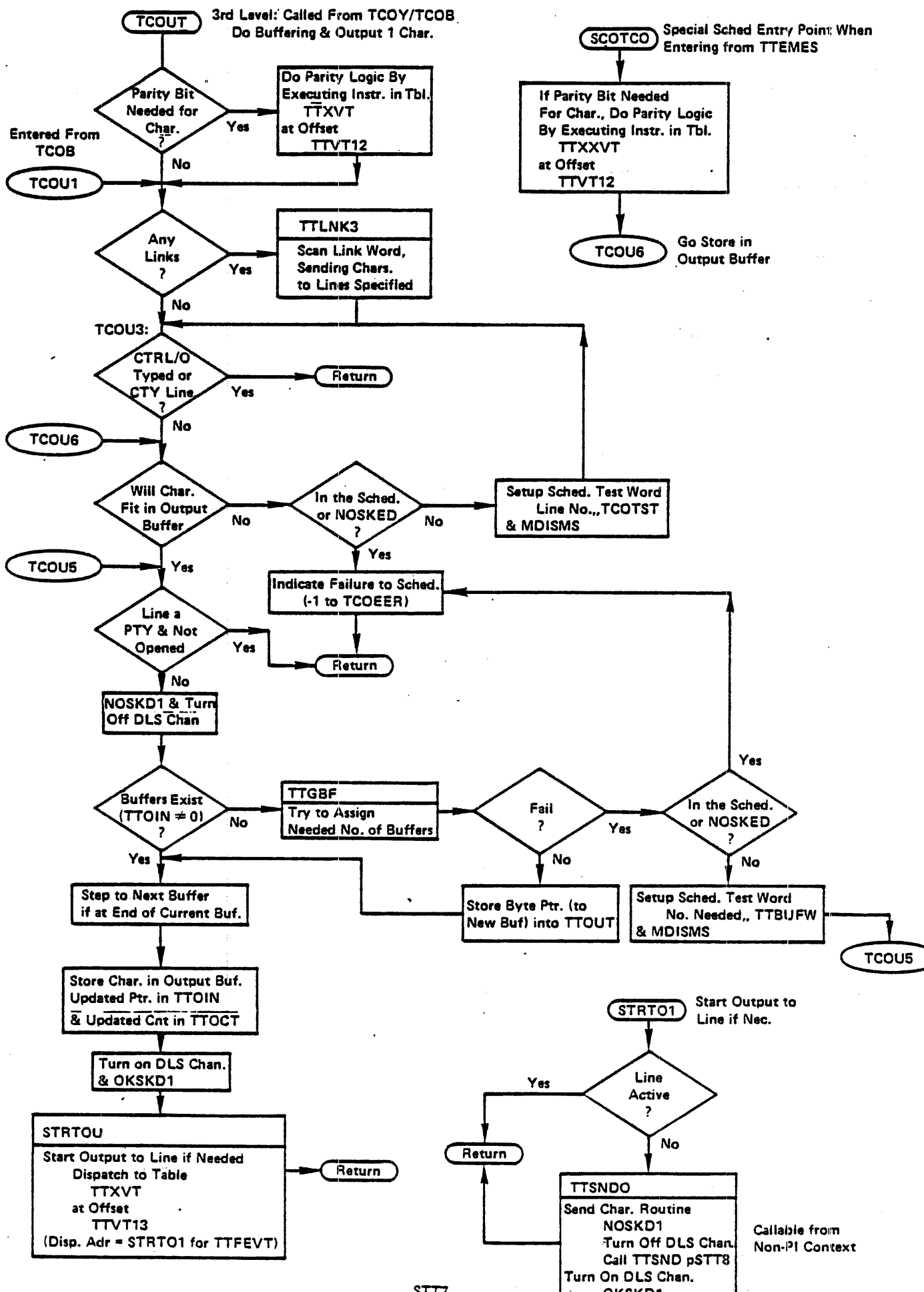
TCO - 1st Level - Translate According to Program's Desires
TCOY - 2nd Level - Do Links & Format for a Particular Device
TCOUT - 3rd Level - Do Buffering, etc.

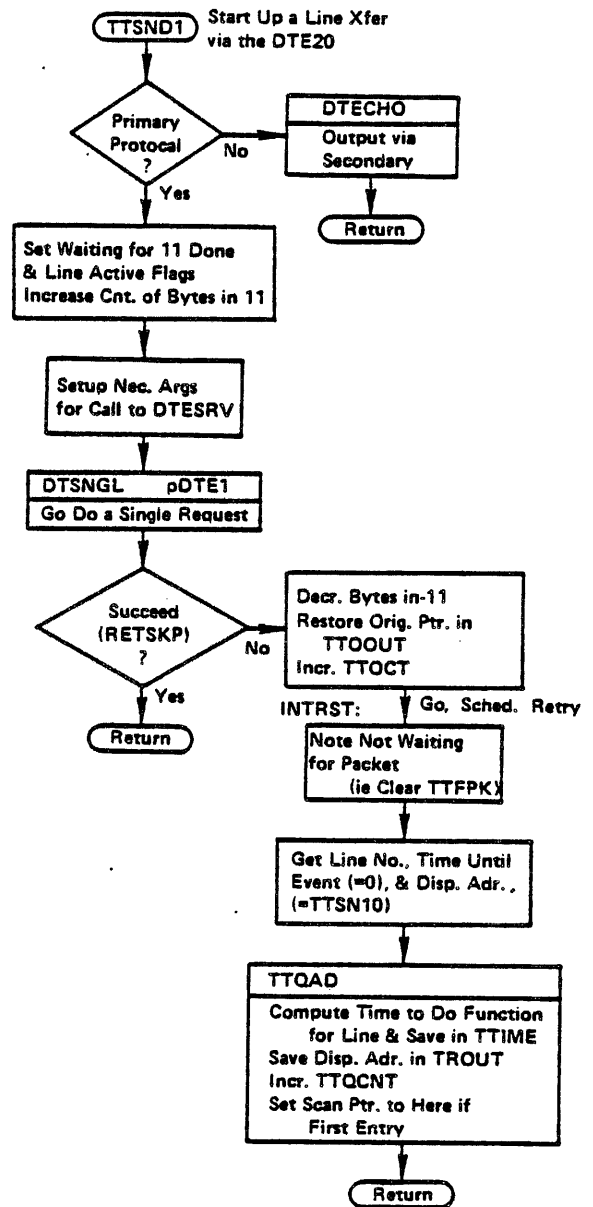
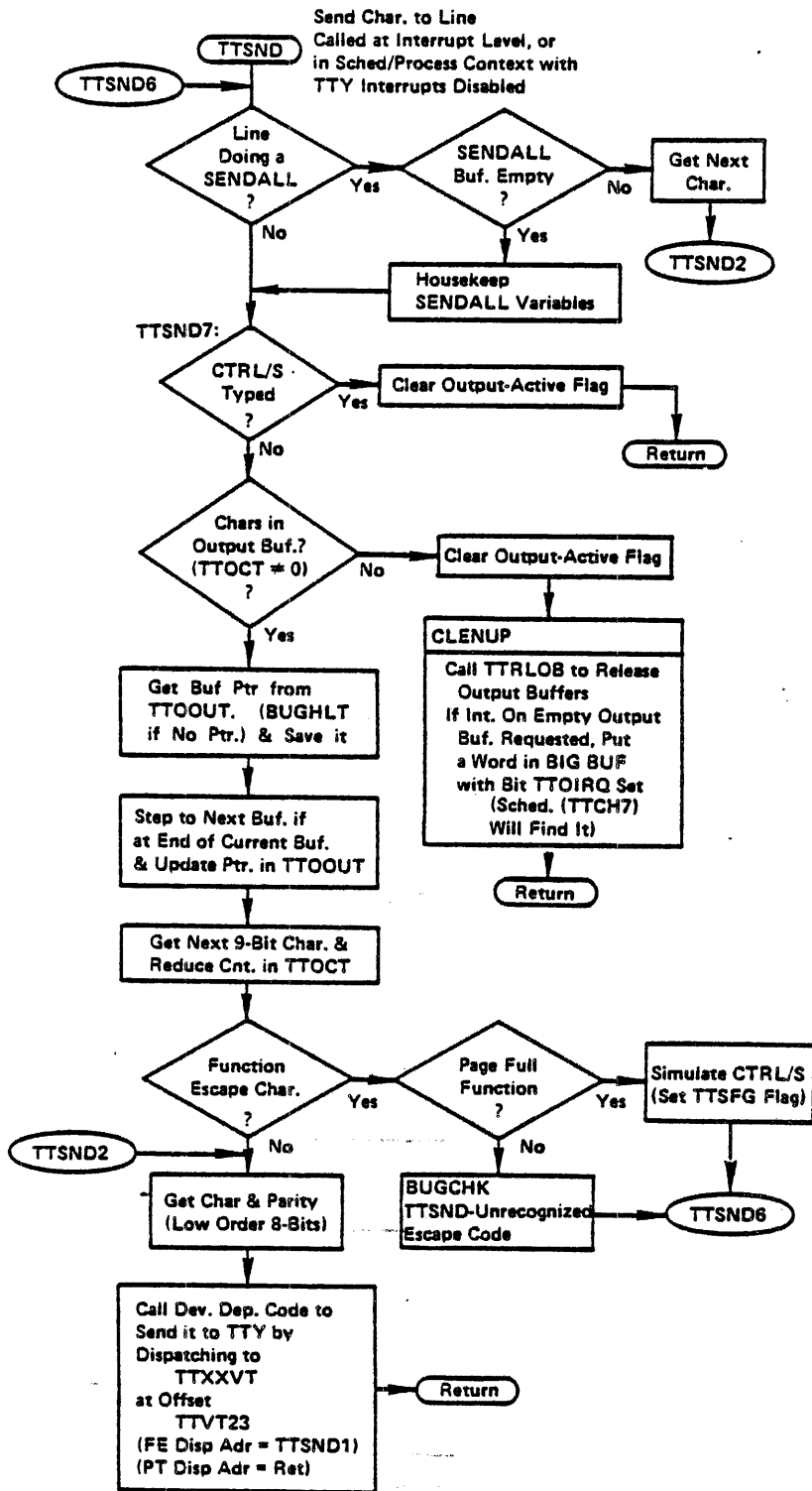


STT5



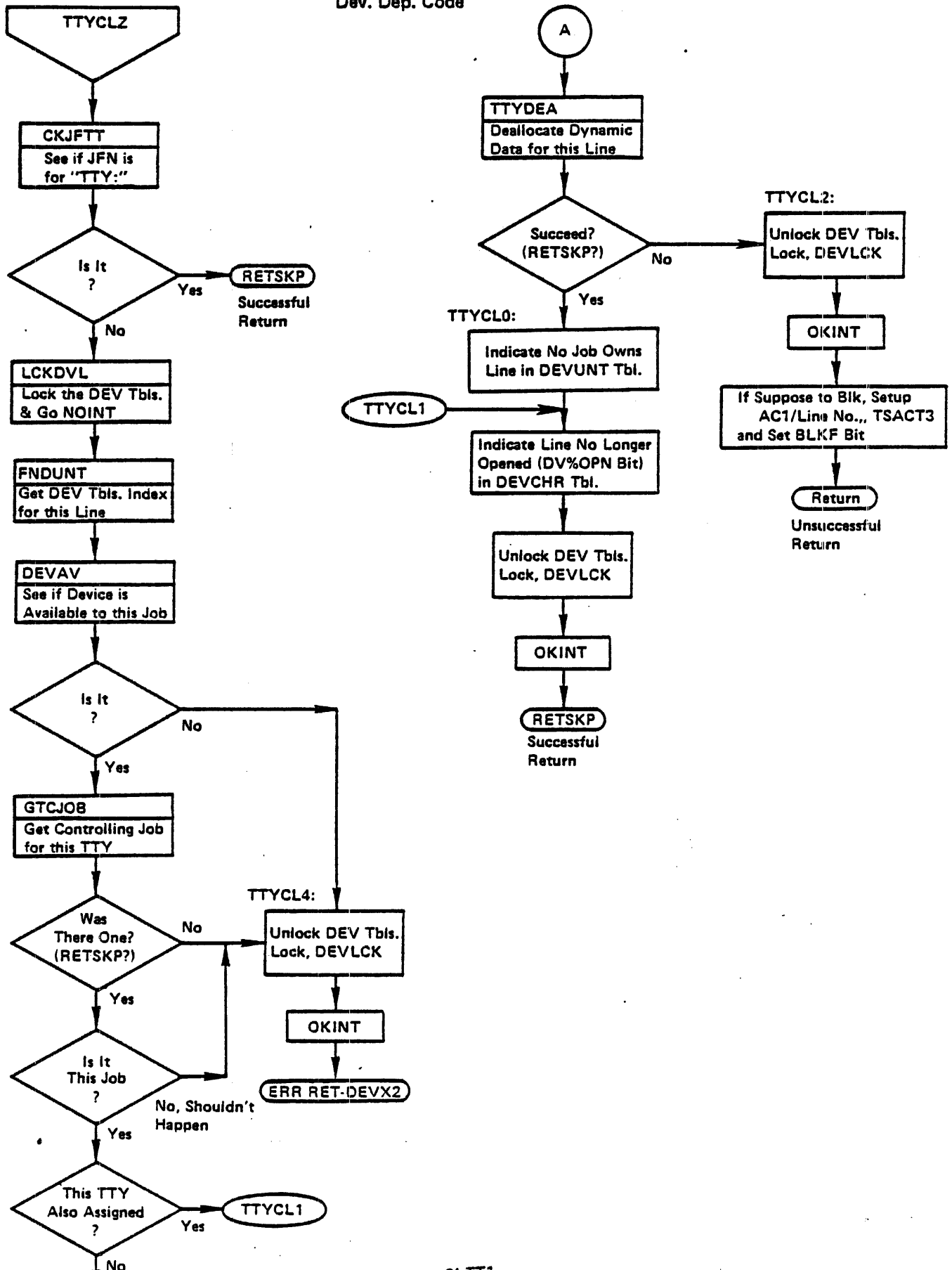
STT6





STT8

CLOSF-TTY
Dev. Dep. Code



SCHEDULER TTY INPUT ANALYSIS & STORAGE

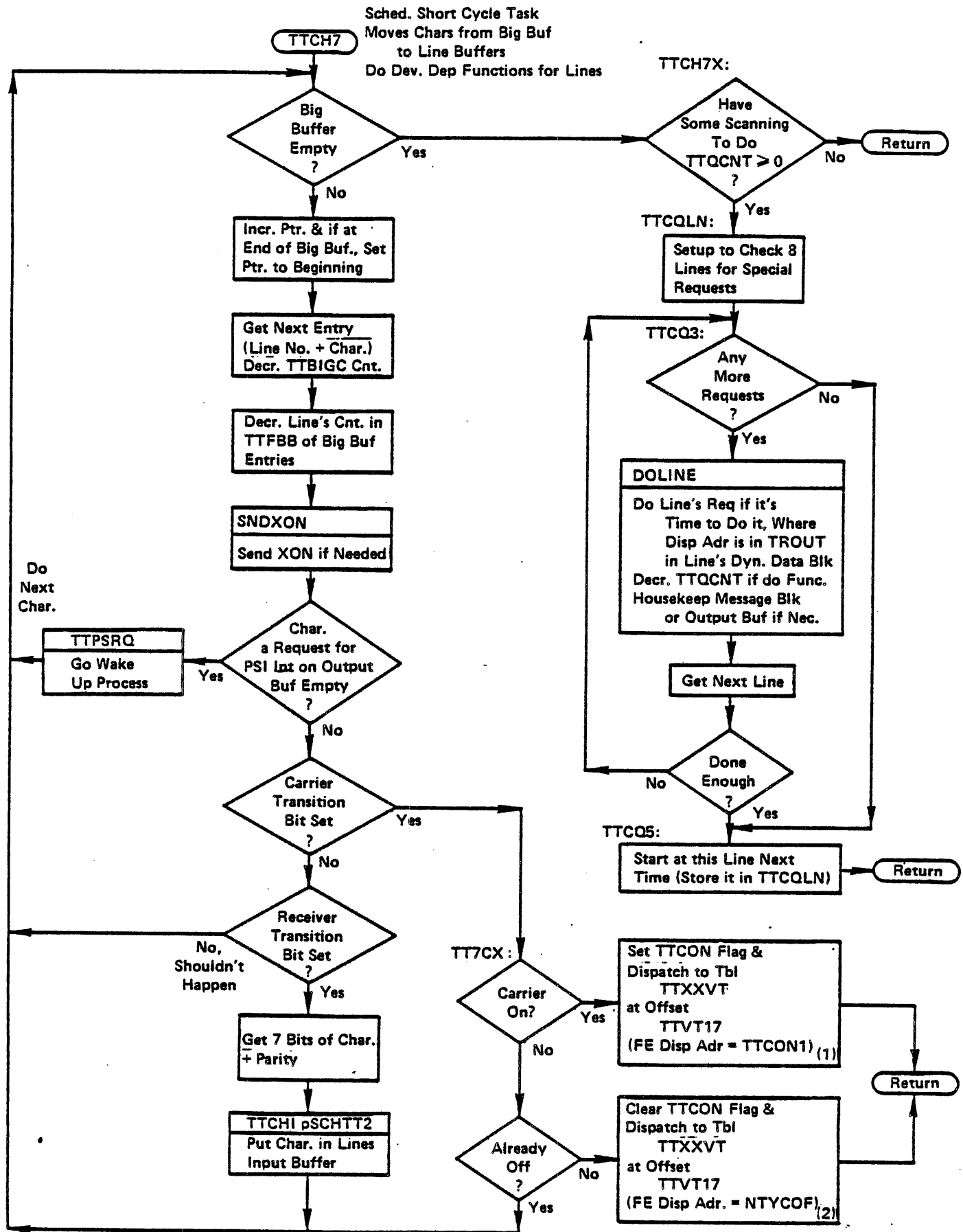
TTCH7 - Moves Characters from the Big Buffer
to Line Buffers

SCHTT1

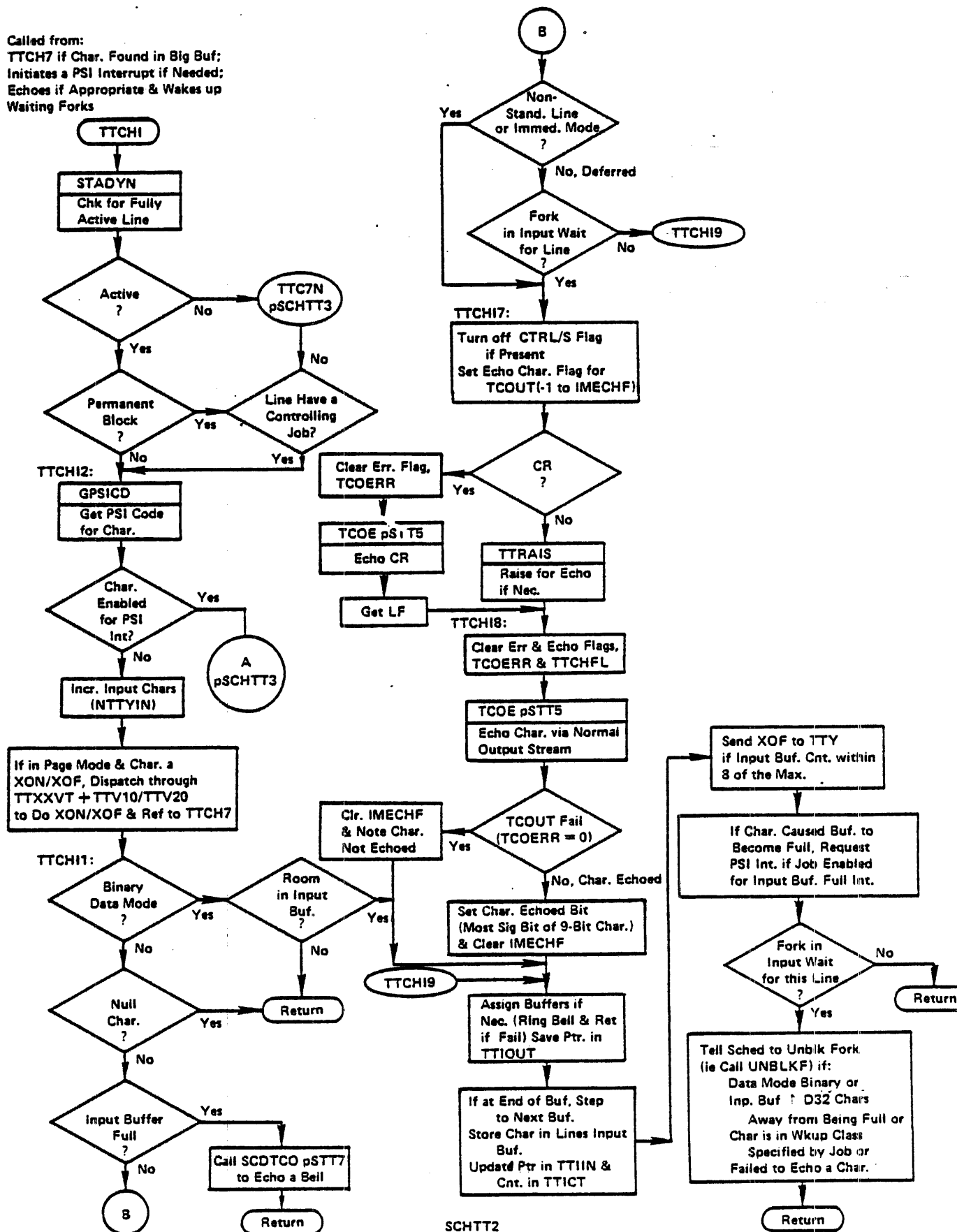
TTCHI - Initiates a PSI Interrupt if
Needed, Echoes if Appropriate
& Wakes Up Waiting Forks

SCHTT2

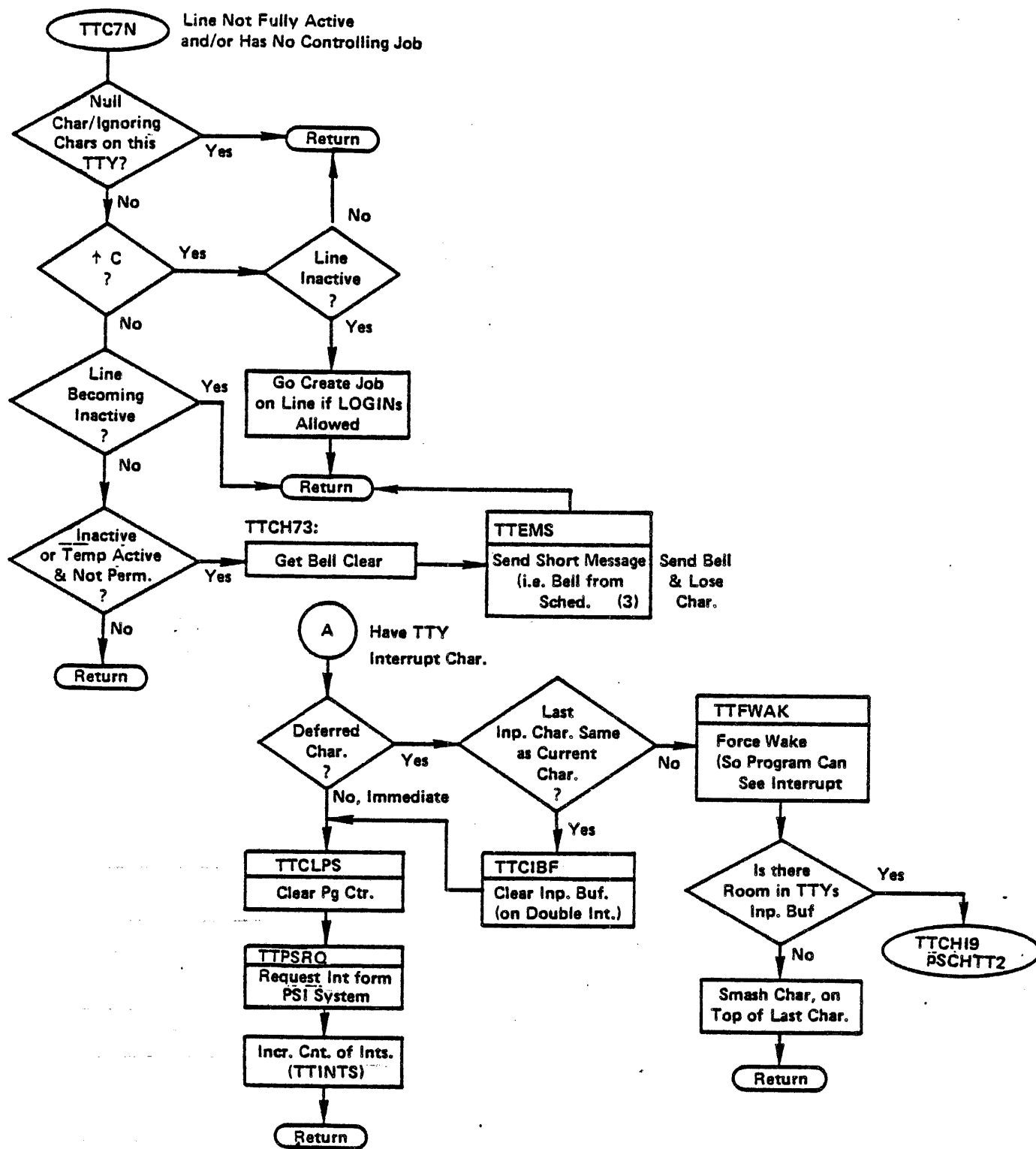
SCHEDULER TTY INPUT ANALYSIS & STORAGE



Called from:
 TTCH7 if Char. Found in Big Buf;
 Initiates a PSI Interrupt if Needed;
 Echoes if Appropriate & Wakes up
 Waiting Forks



SCHTT2



SCHTT 3

Scheduler TTY Input Comments

TTCH7

- (1) The carrier-on routine for the FE device is TICON1. If the line is in use or a job is being created, it just returns. Otherwise, it creates a job by the CTRL/C mechanism (i.e., putting a request in Scheduler's Request Queue, SCDRQB) before returning.
- (2) The carrier-off routine for the FE device is NTYCOF. It flushes outputs and issues an interrupt via the PSI system if process has enabled for carrier-off interrupt. It then issues a monitor-internal interrupt via routine, PSIR4, which causes the top fork to go to JOBCOF in MEXEC to cause the job to be detached.

TTC7N

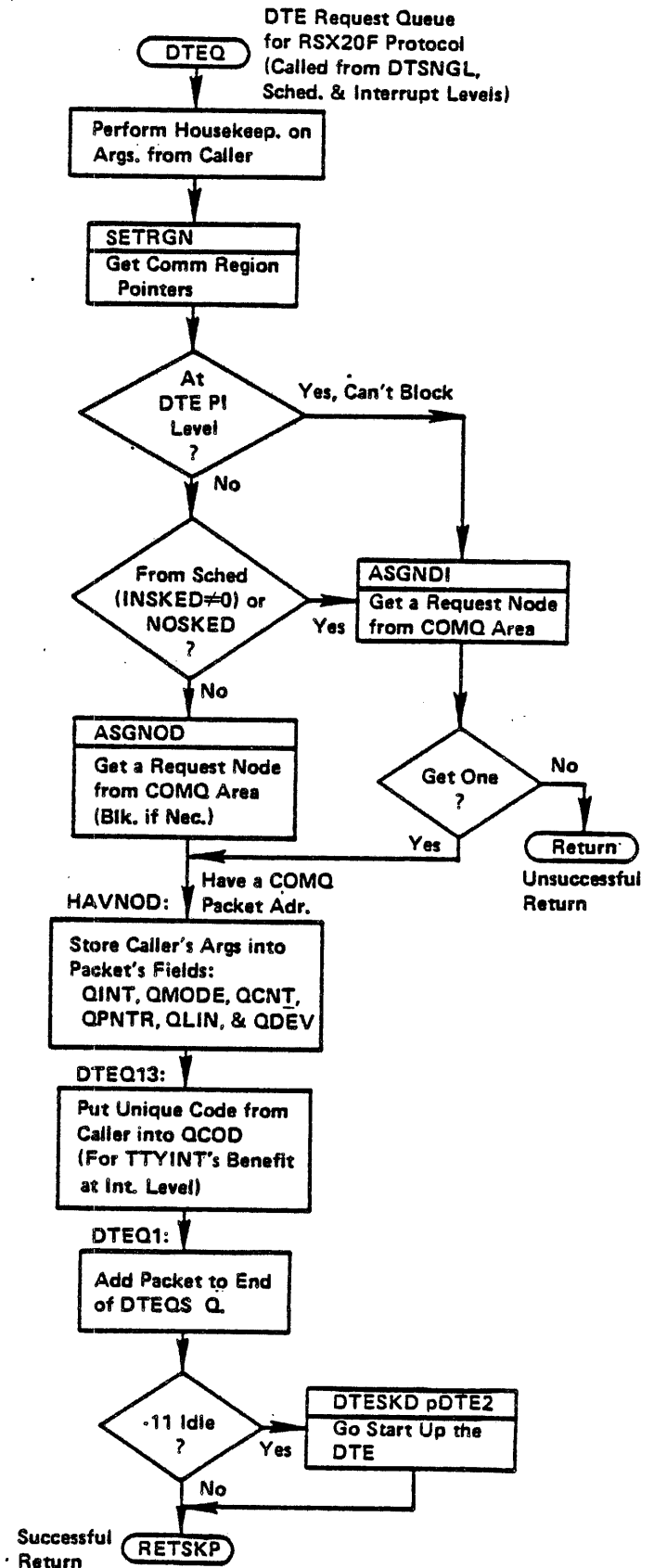
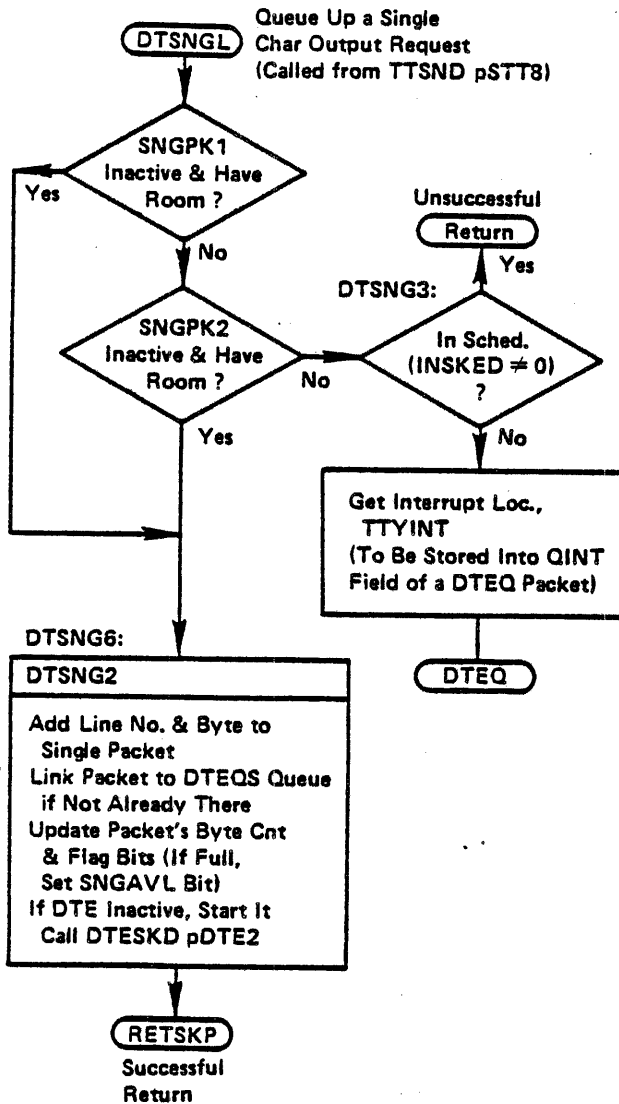
- (3) TTEMES is called at Scheduler Level to send a short message to a line. If the line is active, it appends characters to the line's output buffer. If the line is not active, it creates a message-length dynamic block for the line and puts the characters into this block.

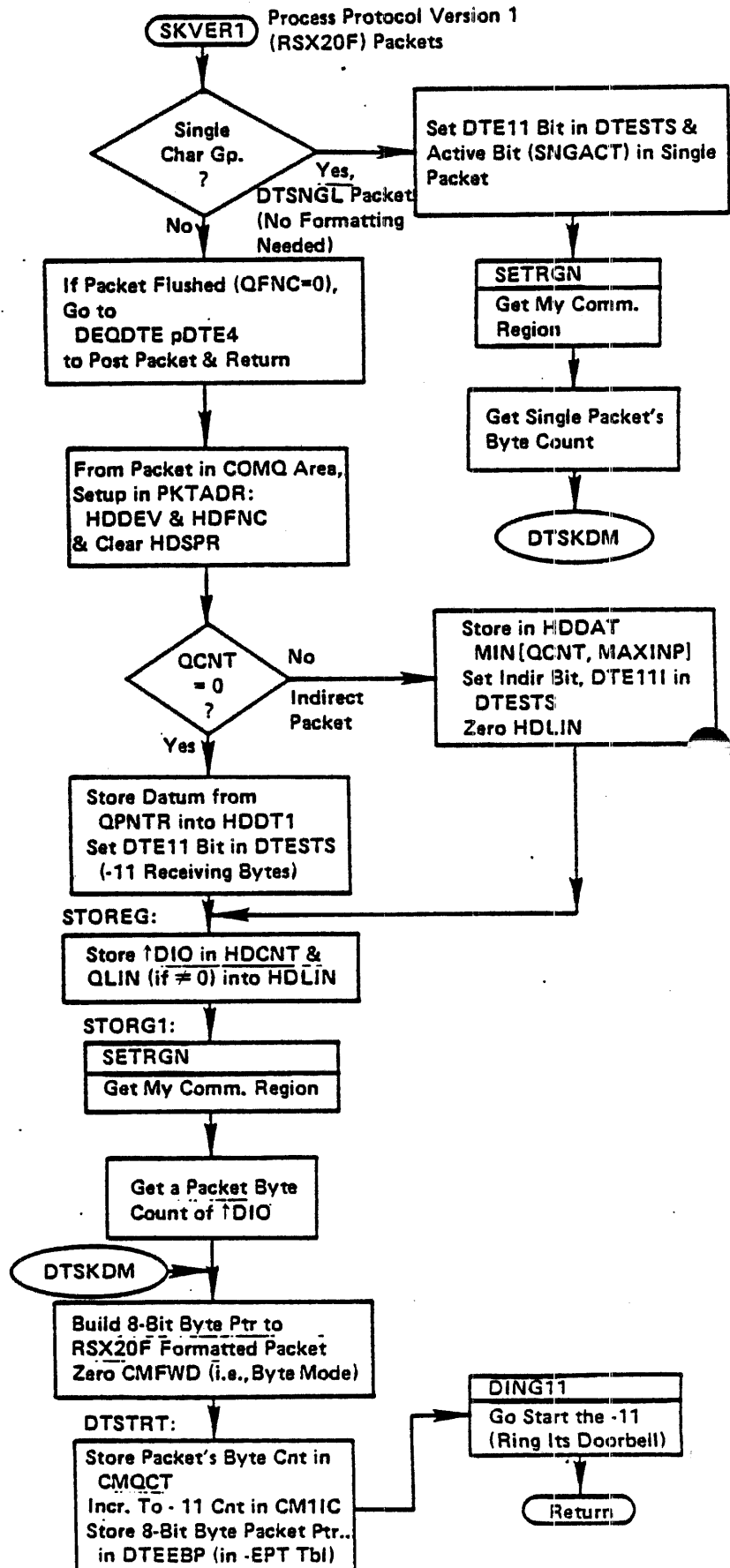
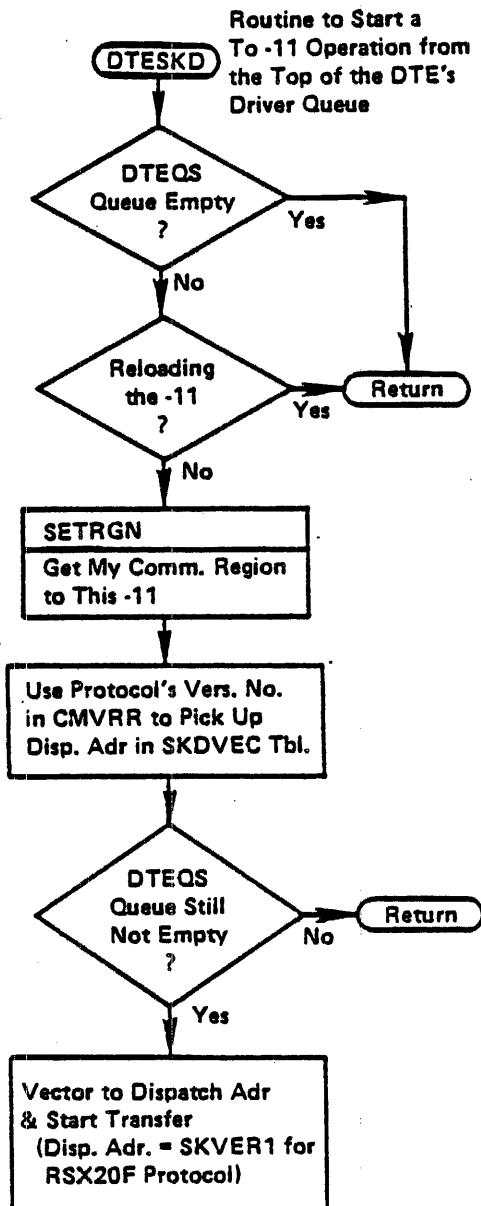
TTEMES calls SCDTCO (pSTT7) to output each character via TCOU to the buffer or message block.

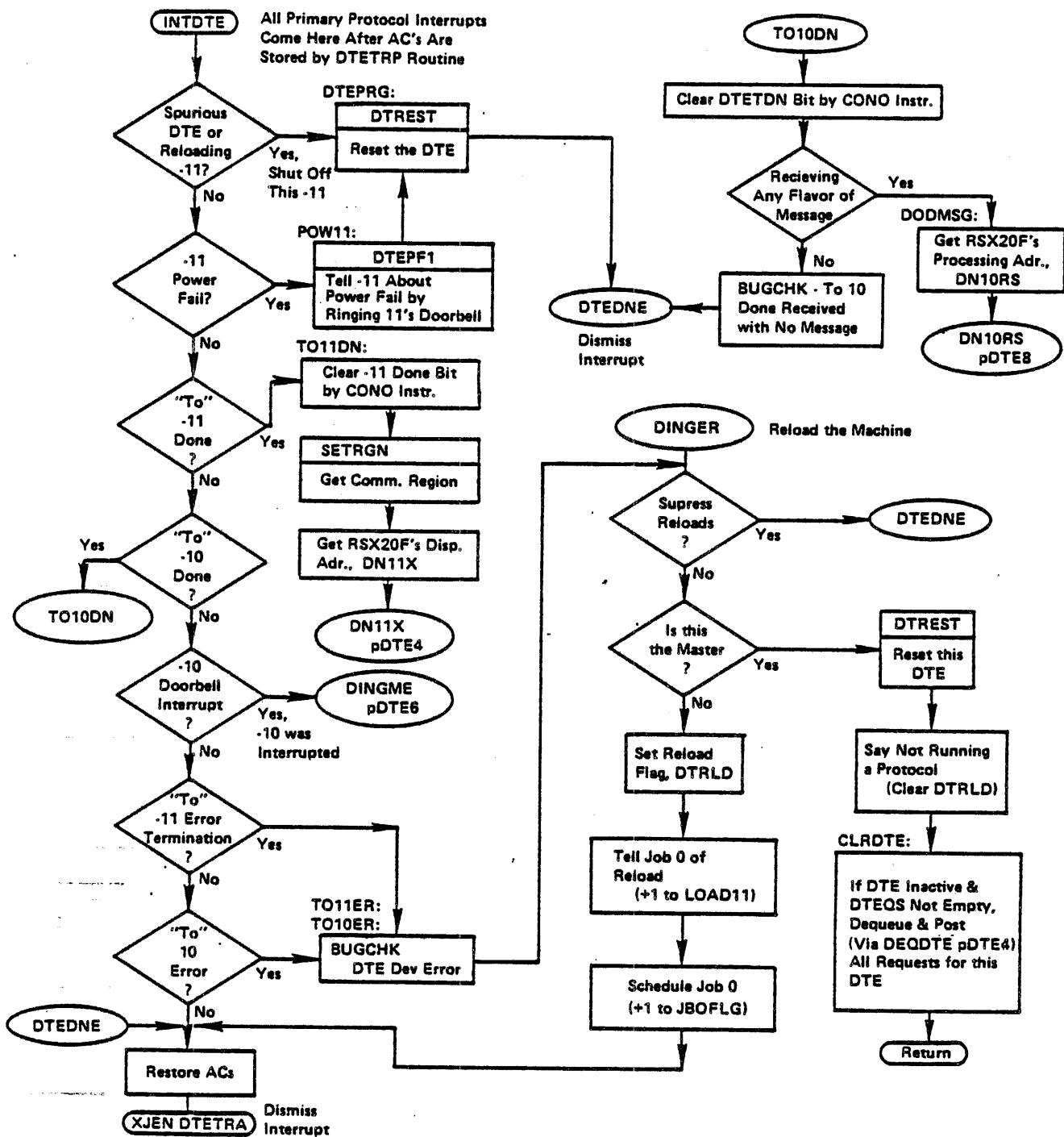
REQUESTING DTE OUTPUT & DTE INTERRUPT HANDLING FLOWCHARTS
(DTE PROTOCOL HANDLER)

DTSNGL - Queue Up a Single Character Output Request	DTE1
DTEQ - DTE Request Queues for RSX20F Protocol	DTE1
DTESKD - Start a To -11 Operation	DTE2
SKVER1 - Process RSX20F Packet	DTE2
INTDTE - DTE Interrupt Handler	DTE3
DN11X - To -11 Done	DTE4
DEQDTE - Dequeue Completed Request, Post it, and Schedule Next One	DTE4
TTYINT - Complete a TTY Output Request	DTE5
DNSNGL - Post Single Character Done	DTE4
DINGME - 10 Received a Doorbell Interrupt	DTE6
DOFRGM - Start a To -10 Transfer	DTE7
DN10RS - To -10 Done	DTE8
TAKLC2 - Process To -10 Done for RSX20F Protocol	DTE9
BIGST2 - Store Character into the Big Buffer	DTE10

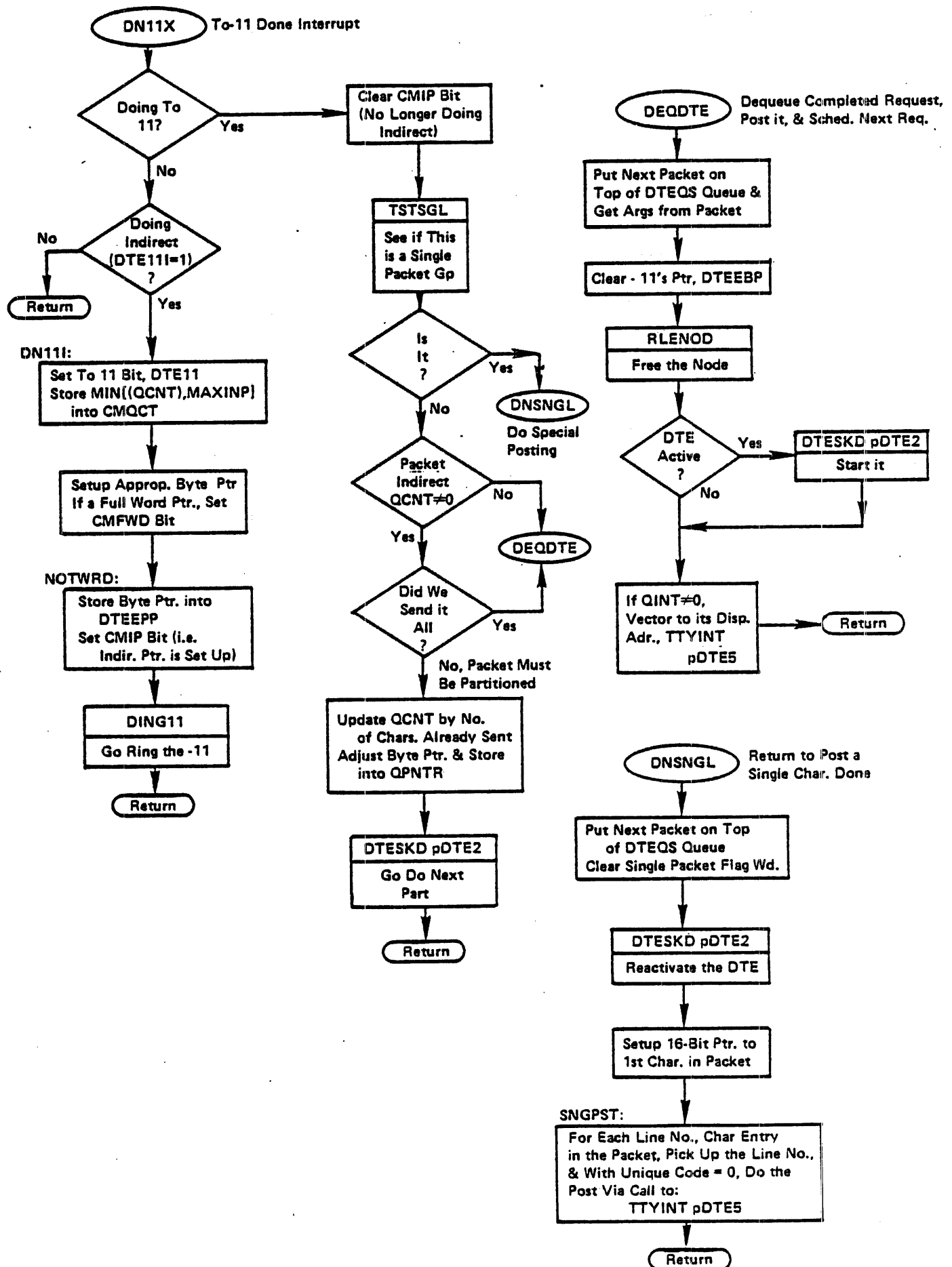
REQUESTING DTE OUTPUT
Called From Interrupt, JSYS
& Scheduler Levels



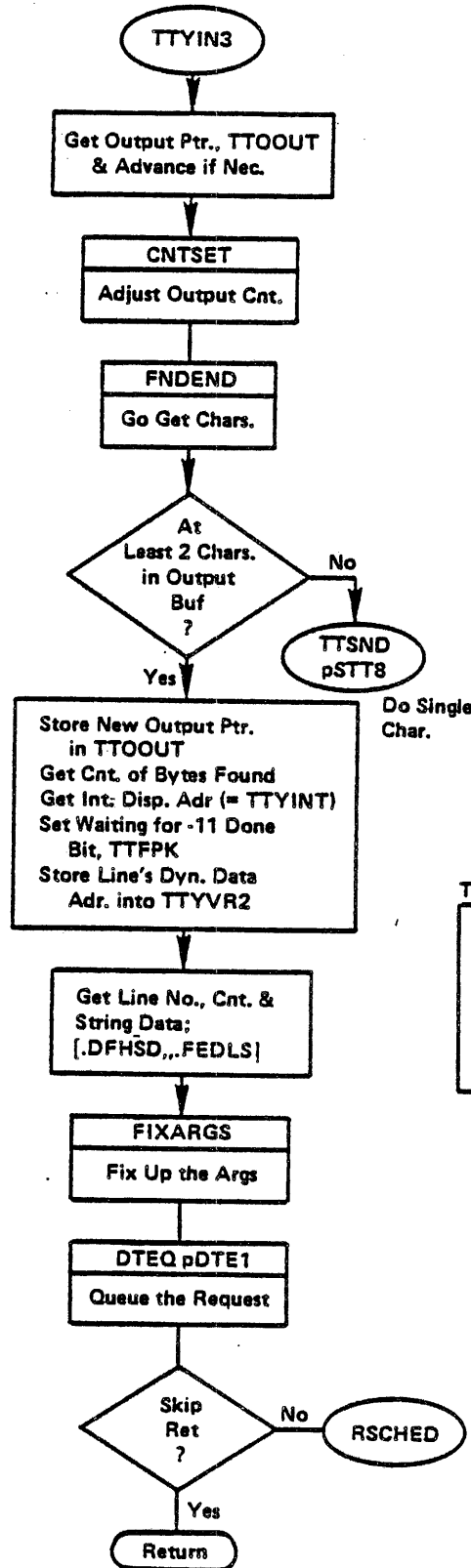
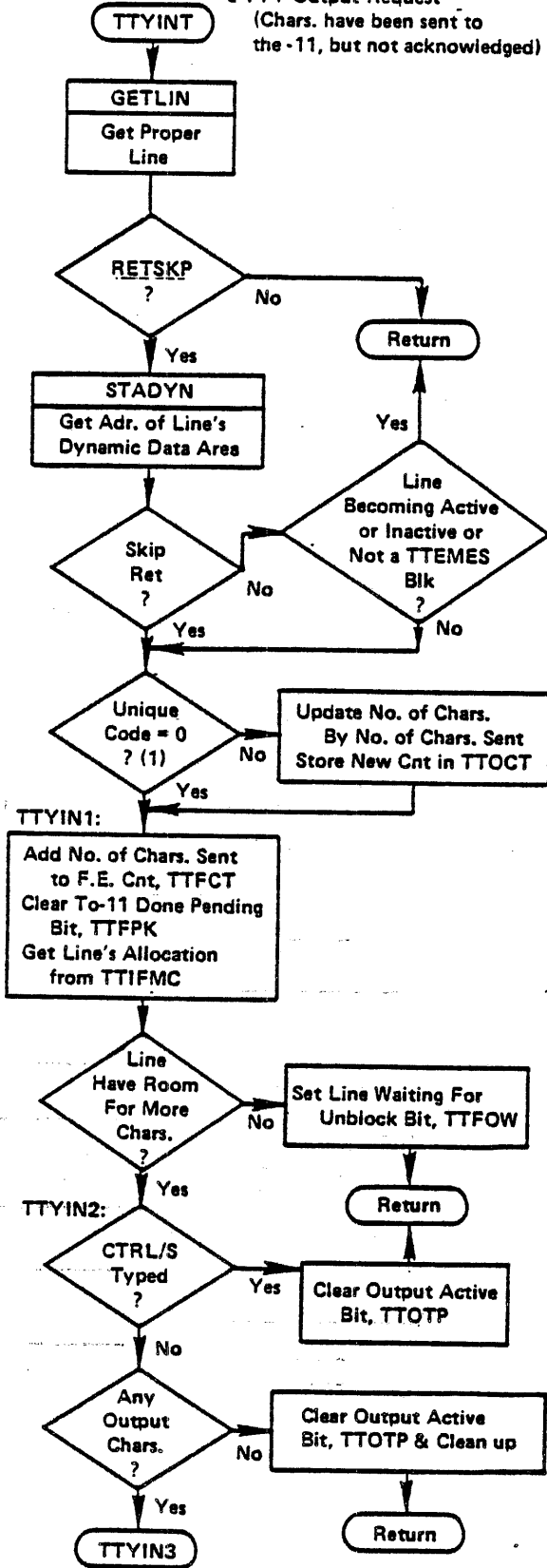




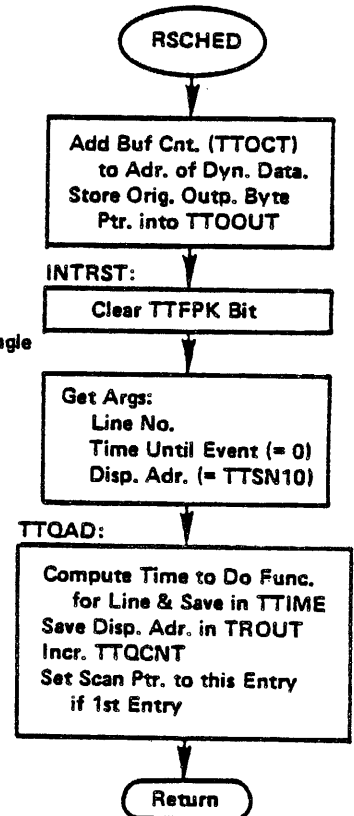
DTE3



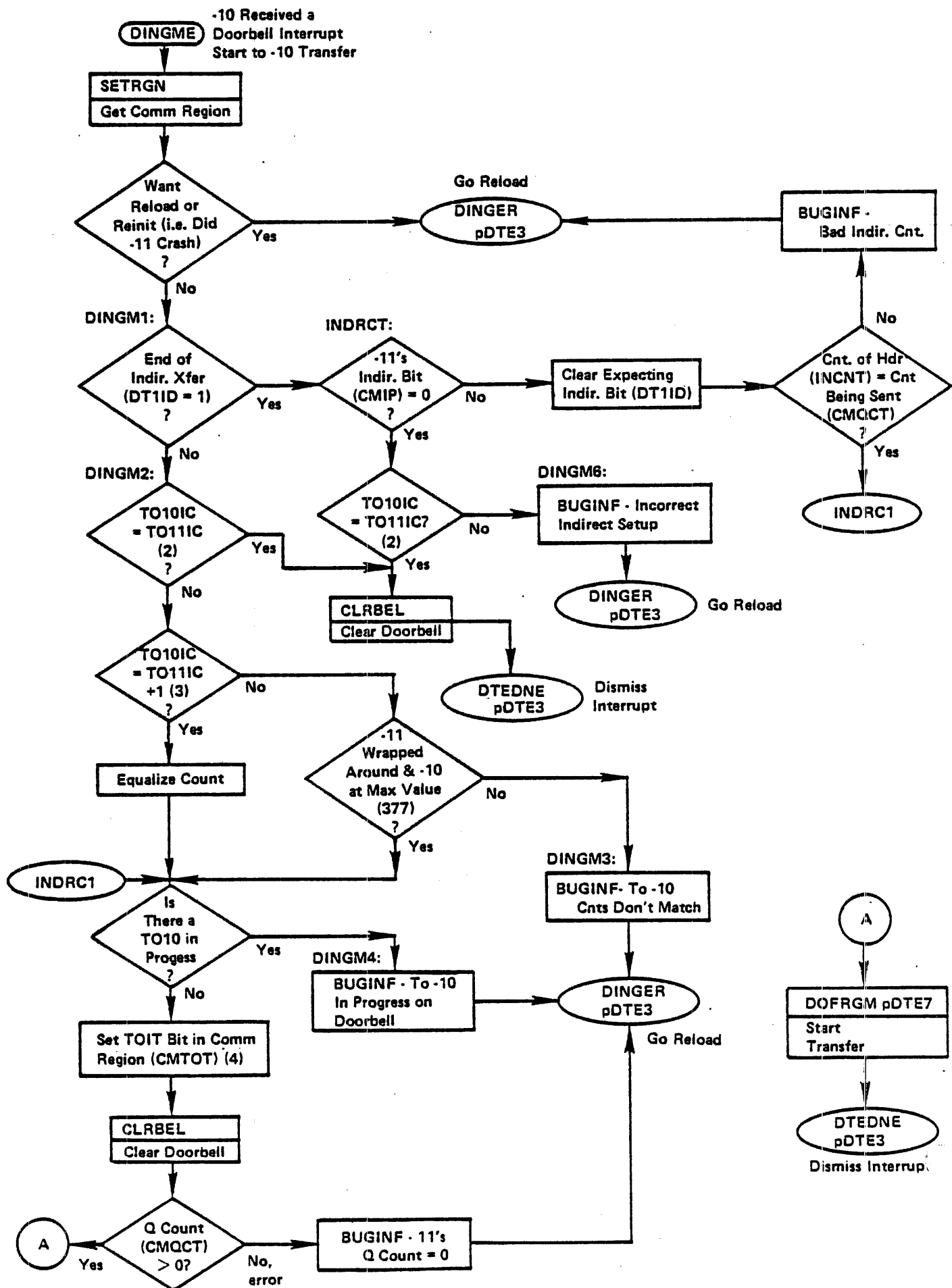
Called at Int. Level to Complete
a TTY Output Request
(Chars. have been sent to
the -11, but not acknowledged)

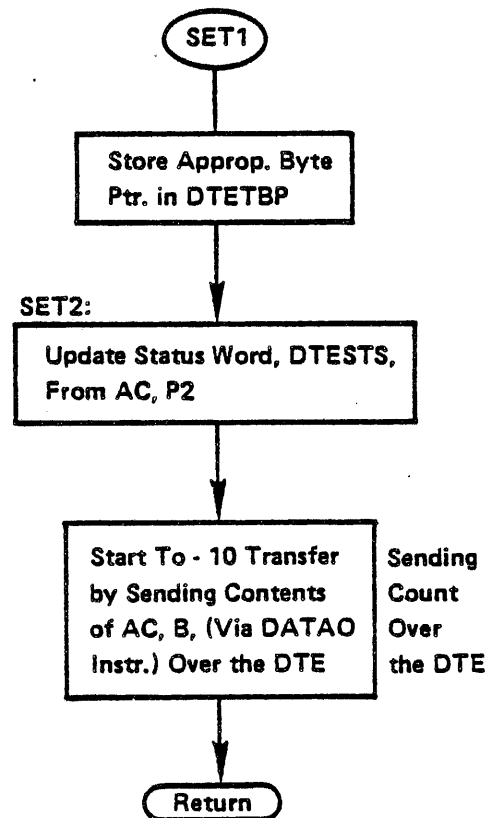
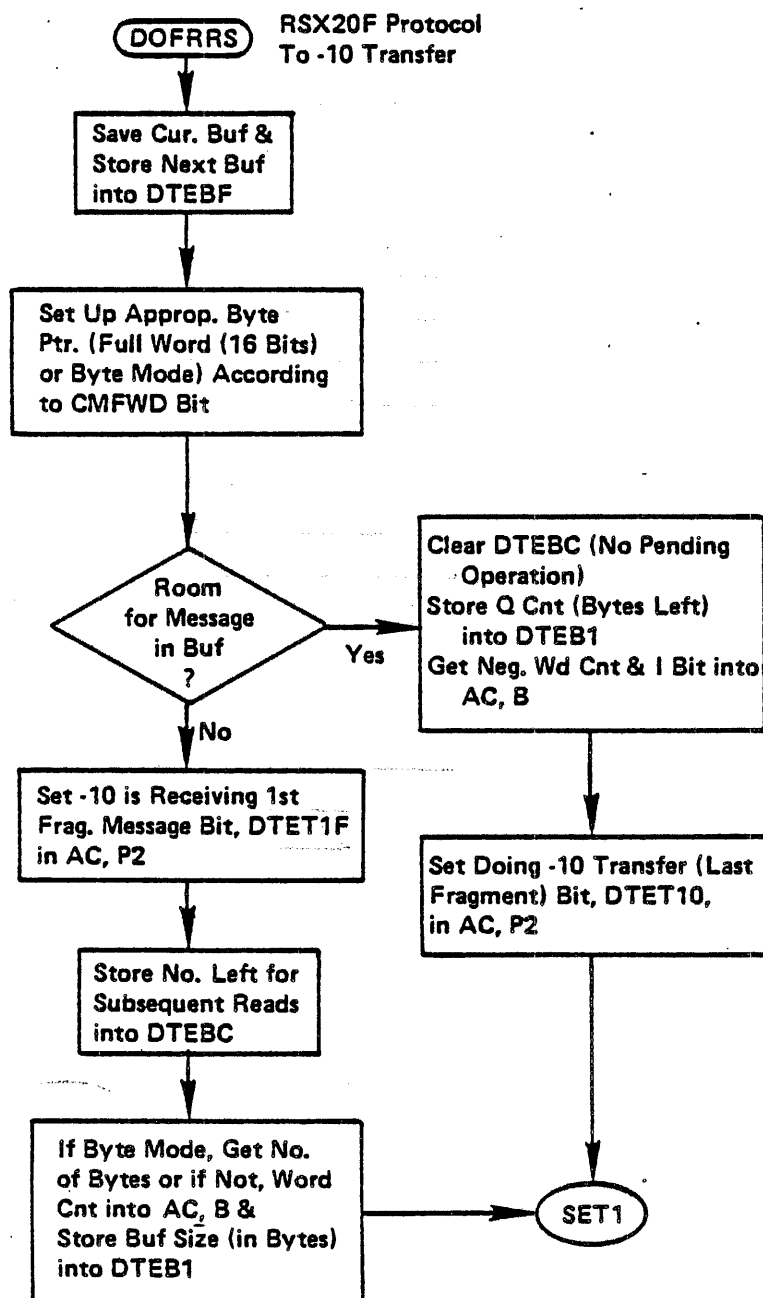
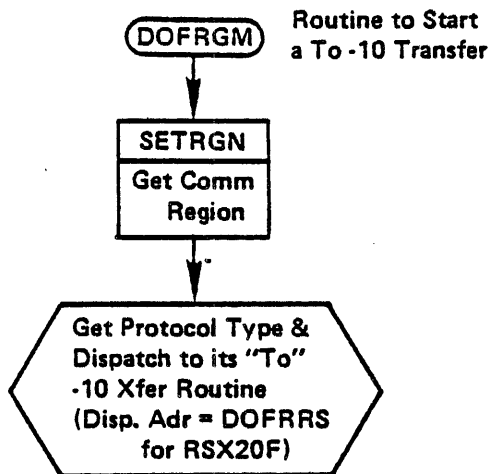


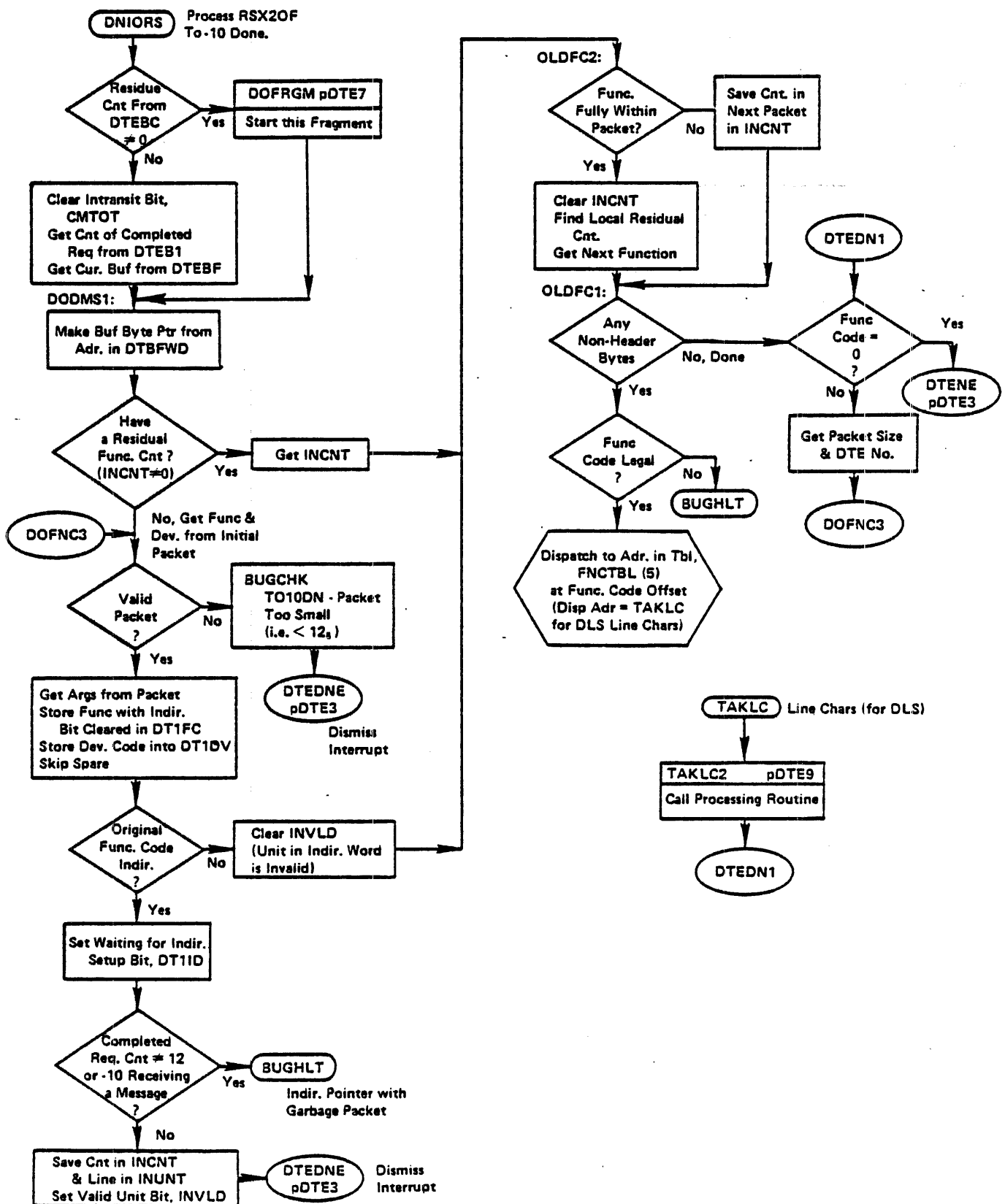
DTEQ Failed;
Restore Cnts. &
Arrange Sched. to
Restart Output



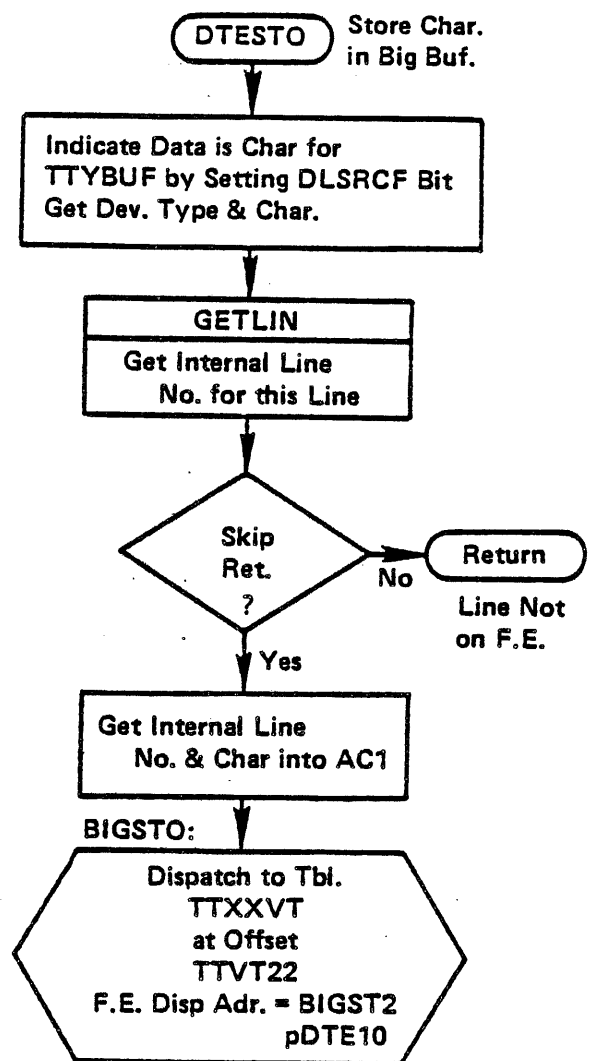
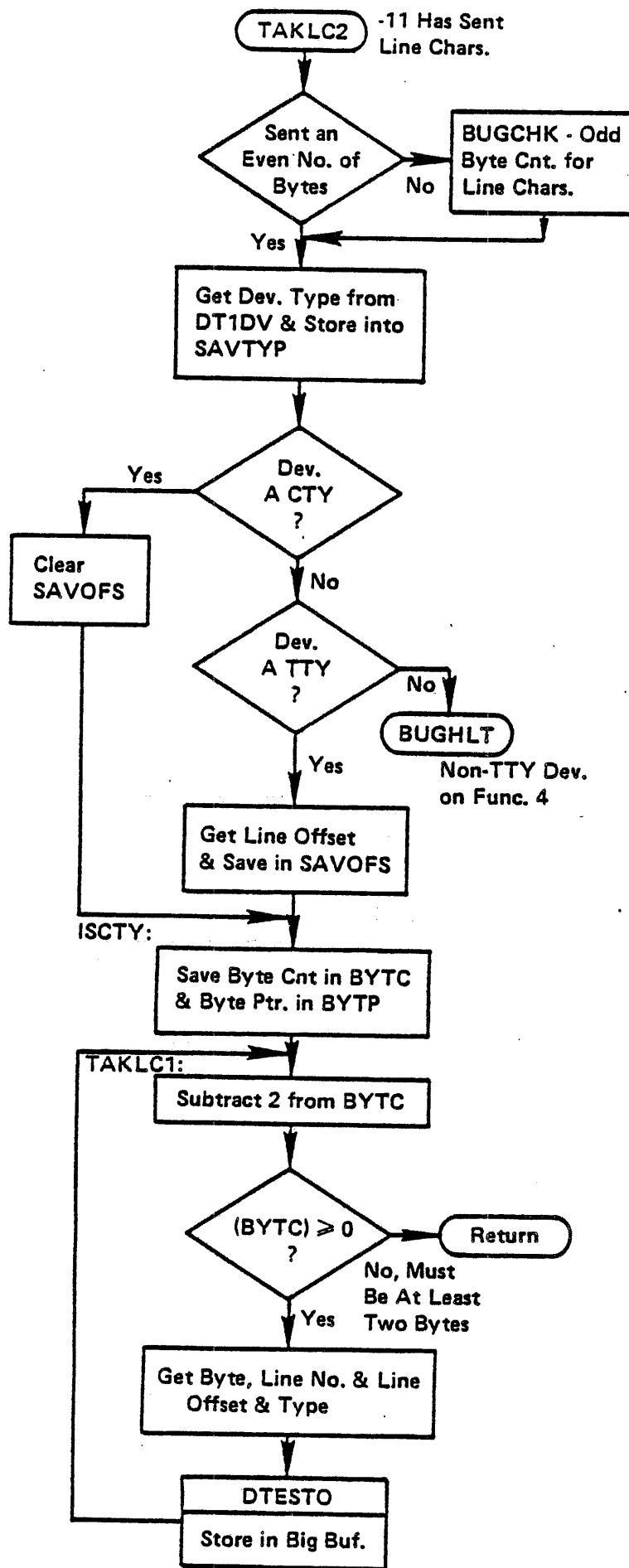
DTE5

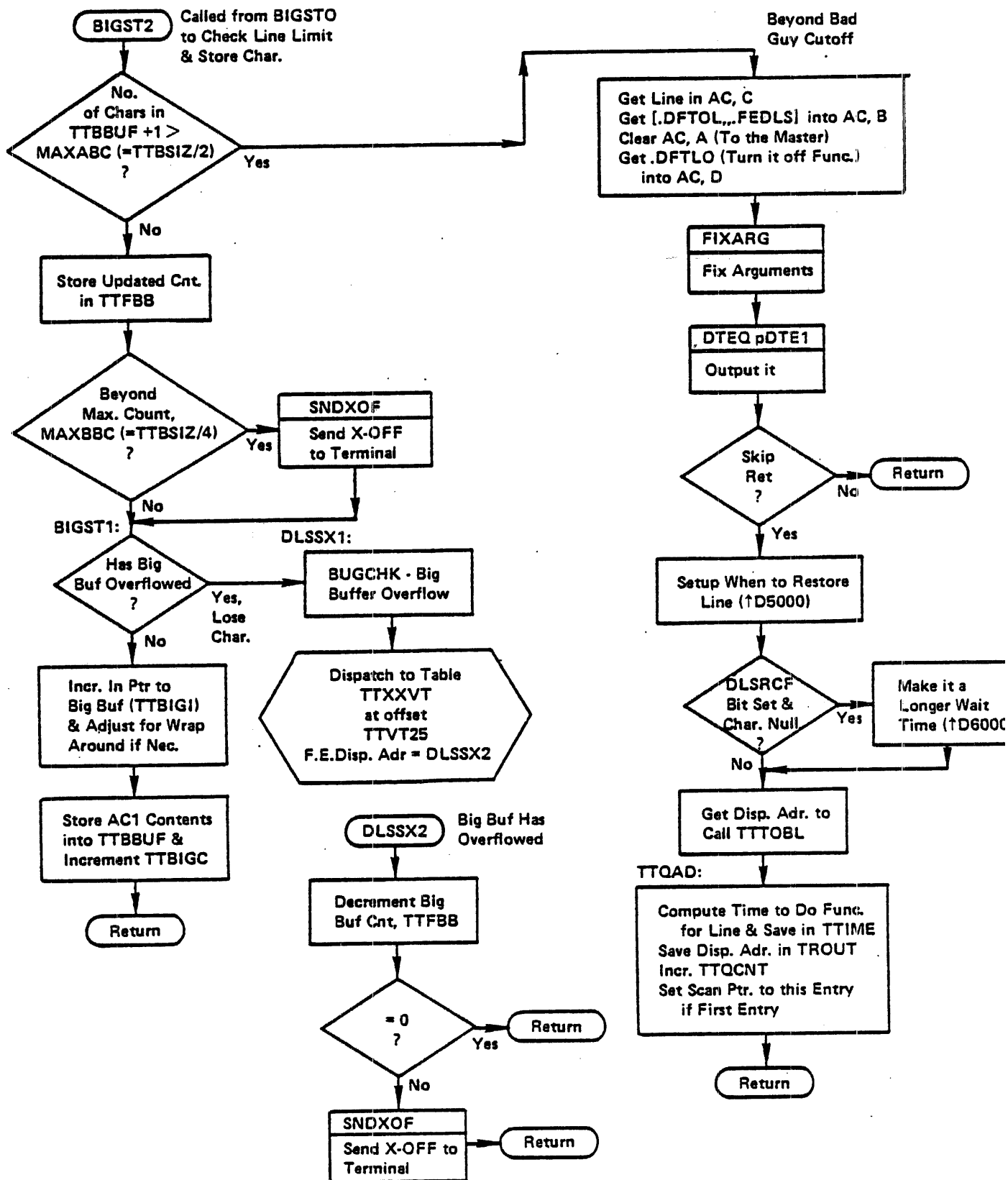






DTE8





DTE10

DTE Interrupt Handling Comments

TTYINT

- (1) The Unique Code argument of form (0, count) tells TTYINT the number of characters that have been sent to the -11 in some call to DTEQ that specified TTYINT as its return address.

Count = 0 implies this was a single character (DTSNGL was called) and buffer counts have already been updated.

Count $\neq$ 0 implies this was multiple characters and the count must be updated.

DINGME

- (2) TO10IC and TO11IC are wrap-around counters of Indirect Transfers where TO10IC is maintained by the -11 and TO11IC is maintained by the -10. If the two wrap-around counters are equal, it means the transfer finished correctly.
- (3) If the difference between the wrap-around counters is greater than 1, the -11 has tried to send a direct transfer before the last indirect transfer finished or a doorbell has been lost in a previous transaction.
- (4) Receiver sets TOIT equal to 1 in Sender's section of Receiver's communication region after Sender sets @ or increments Q count and rings the doorbell; Receiver clears TOIT upon getting To-Receiver Done (This assures that the Receiver doesn't lose an interrupt).

DN1ORS

(5) The function table has dispatches for such features as:

- F.E. telling about the CTY
- String data for the CDR
- Line characters (for DLS)
- -11 Sending error information
- -11 wants or is sending time of day
- Line dialed up, hung up or line buffer empty
- Set line speed or allocation
- Take -11 reload information
- Acknowledge all devices and units
- Take KLINIK data.

CHAPTER 1

EXTENDED ADDRESSING IN TOPS-20

1.1 INTRODUCTION

The information in this section is of a preliminary nature, and should not be construed as a commitment by Digital Equipment Corp. TOPS-20 does NOT support user-mode extended addressing at this time. The Jsys described herein is available but not documented nor supported. The purpose of this information is to inform the student of the eventual availability of this feature to user mode programs, as we have already learned about the monitor's use of this hardware functionality of the Model B KL processor.

Reference:

DECsystem-10/DECSYSTEM-20 Hardware Reference Manual, Vol.
1 Chapter 3, section 4, page 3-23: Tops-20
Paging and Process Tables

Release 4 of TOPS-20 does NOT support user mode extended addressing. Extended addressing is a feature of the current KL (sometimes referred to as "Model B"). With this feature, TOPS-20 itself will use extended addressing to expand its own address space. This is true to different extents depending on whether the system is running version 3A or version 4, and also whether the machine is running a 2050 or 2060 monitor.

The main purpose of extended addressing is to allow expansion of the previous 18 bit virtual address on the KI, KS and Model A KL processors to 23 bits. Actually, the software is capable of supporting a processor with 27 bit virtual addresses. The design as described herein is intended to support a 30 bit virtual address space, making our high-end products support larger address spaces than VAX/VMS. (VAX supports a 30 bit address space as well, but VAX is byte-addressed). Since the code to handle the extended capabilities exists within the monitor (for the KL), the monitor is the first program built to take advantage of the firmware features.

The implementation of extended addressing calls for the "sectioning" of the large virtual address space. Each section contains 512 pages, and corresponds to the traditional 18 bit address spaces we are used to. The number of sections available depends on the software and firmware capabilities. Current support is for the KL, meaning 5 more bits for the section number. Therefore, each process can have 32 address spaces each containing 512 pages. All sections work the same, with the exception of section 0. Section 0 references are always in the traditional 18 bit format. Non-section 0 references can be local or global. With local addressing, instructions executed in that section can reference other locations within the section, using the same old 18 bit addresses and instruction format as always. Global references need to supply a full virtual address, including a section number.

1.1.1 Current Implementation Objectives

The current implementation of user mode extended addressing (available but not supported by Digital) intends to provide the capability for programs to create and use multiple sections for data storage. Also, to allow programs to be run in any section, or in multiple sections, with certain restrictions. These restrictions are that addresses passed to most Jsyz's as arguments must specify arguments in the same section as the call itself.

1.1.2 JSYS Interface

Each section has a page map, and the page maps for a given process are pointed to by the process' section map, which exists in the User Process Table starting at location USECTB (540). To find the page map for section n, we simply look at USECTB plus n, where we find a section pointer to the page map. The Jsyz's which deal with user mode extended addressing manipulate these maps. The SMAP% Jsyz sets the map for a section or a virtually contiguous series of sections. The SMAP% Jsyz does with sections what the PMAP% Jsyz does with pages. This includes the possibility of specifying whether the section is private, shared with a file (files don't really have sections, so this specifies any group of 512 contiguous pages in the file), or shared with some other fork, or some other section within the current fork. This indirect relationship is analogous to the indirect mapping available on a page basis with the PMAP% call. This means that when the source mapping is changed, the destination will see the change in the section's map.

The RSMAP% Jsyz (currently unavailable) will be used to obtain information about the current mapping of sections. This corresponds to the RMAP% Jsyz used to obtain information on pages

within a section. Some other Jsys's will be added or changed to allow certain functions to occur within the extended environment, and usually these have the letter X prefixed to the name of the corresponding Jsyes. For example, XSIR and XRIR are used to specify channel and level tables with full 30 bit addresses, so the software interrupt system can be section independent. Note that most Jsyes calls which allow the program to supply an argument block address will currently not work if executed in a non-zero section.

1.2 PROGRAMMING IMPLICATIONS AND METHODOLOGY

The basic architecture of the DECsystem-10 family of processors up through the KL10 provided an 18-bit, 256K-word addressing space. This means in particular that:

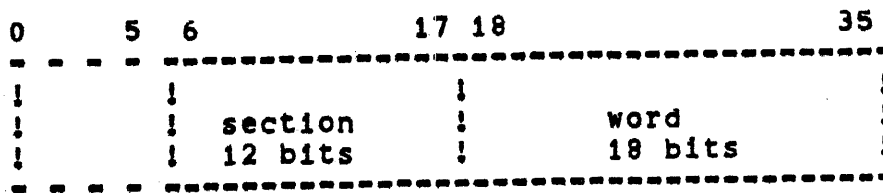
1. The Program Counter register is 18 bits;
2. Each and every instruction executed computes an 18-bit effective address. The contents of this address may or may not be referenced depending on the actual instruction, but the algorithm for calculating it (including indexing and indirecting rules) is the same for all instructions.

The above is true regardless of the size of the physical core memory available on any particular configuration. Note also that paging (or relocation on earlier processors) will determine if a particular virtual address corresponds to an existing physical word in memory, but the fundamental size of the virtual address space is a constant.

During the design of the KL10 processor, the need for a larger virtual address space was recognized. A design was developed which provides an extended address space to new programs while still allowing existing unmodified programs to execute correctly on the same processor. Although most of the essential data paths were provided in the original KL10 implementation, various design changes caused actual availability of extended addressing operation as described here to be deferred to the "model B" KL10 MPU.

1.2.1 Virtual Address Space

Under the extended addressing design, the virtual address space of the machine is now 30 bits, or 1,073,741,824 words. Although one can think of this as a single homogeneous space, it is generally more useful to consider an address as consisting of two components, the section number, and the word number.



The word (more precisely word-within-section) field consists of 18 bits and thus represents a 256K-word address space similar to the single address space on earlier machines. The section number field is 12 bits and thus provides 4096 separate sections, each of 256K words.

Each section is further divided into pages of 512 words each just as on earlier machines. The paging facilities allow the monitor to independently establish the existence and protection of each section as a unit.

In order to implement this extended 30-bit address space, the PC must be extended to hold a full address. The PC is always considered to consist of a section field and a word field, and the incrementing of the PC never allows a carry from the word field into the section field. That is, if a program allows flow of control to reach the last word of a section and the instruction in that location is not a jump, then the PC will "wrap-around", and the next instruction fetched will be word 0 of the same section. This will in fact be AC 0 as described below. The consequence of this is that flow of control of a program cannot implicitly cross section boundaries. In general, it would be a programming error to allow the PC to reach the last location of a section and execute a non-jump instruction or to execute a skipping instruction in either of the last two locations of a section.

1.2.2 Compatibility

In order to allow efficient use of the extended address space, it was necessary to modify the operation of various machine instructions and to change the algorithm for the calculation of

effective addresses. Because these changes have a high probability of causing any existing program to malfunction, the following convention was adopted:

If a program is executing in section 0, all instructions are executed exactly as they would be on a non-extended machine. If a program is executing in any section other than 0, the extended addressing algorithms are used for effective address calculation and instruction execution.

A program is said to be executing in section 0 when the section field of the PC contains 0. Effective address calculations and instruction executions are performed exactly as on a non-extended machine; hence any existing program will work correctly if run in section 0. Note however, that this also implies that no addresses outside of section 0 can be generated, either for data references or for jumps. That is, a program executing in section 0 cannot leave section 0 except via a monitor call. The only exception to this is the XJRSTF instruction.

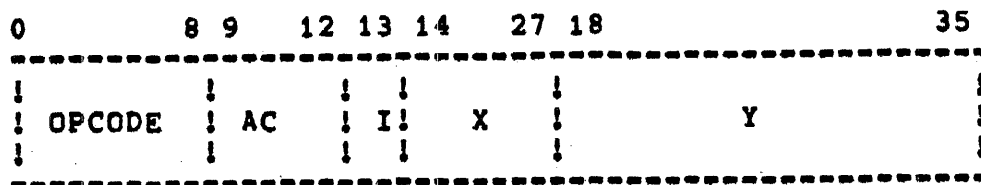
It is easy however to write or generate code which is compatible with both extended and non-extended execution. This was a specific goal of the design, and in general requires only that certain precautions must be taken regarding previously unused bits. Since these precautions must be taken however, one cannot generally assume that any existing program will have observed them and so execute correctly in an extended section.

2.3 Effective Address Calculation

Unless explicitly stated otherwise, everything in the following discussion refers to execution of instructions with the PC in a non-0 section; nothing applies to instructions executed from section 0.

1. Instruction format

The format of a machine instruction is the same as on a non-extended machine. In particular, the effective address computation is dependent on three quantities from the instruction, the Y (address) field, the X (index) field, and the I (indirect) field. These are 18 bits, 4 bits, and 1 bit respectively.



Depending on the format of the index and indirect words, the effective address algorithm will perform either 18-bit or 30-bit address computations. When a 30-bit quantity is indicated, an explicit section number is being specified and the address is called a global address. When an 18-bit quantity is indicated, the section field is defaulted from some other quantity (e.g., the PC), and the address is thus local to the default section and is called a local address.

In the simplest case, consider an instruction which specifies no indexing or indirection. E.g.,

3,,400/ MOVE T,1000

Here the effective address computation yields a local address 1000, and the section used for the reference is 3, i.e., the section from which the instruction itself was fetched. Precisely stated:

Whenever an instruction or indirect word specifies a local address, the default section is the section from which the word containing that address was taken.

This means that the default section will change during the course of an effective address calculation which uses indirection. The default section will always be the section from which the last indirect word was fetched.

2. Indexing

The first step in the effective address calculation is to perform indexing if specified by the instruction. The calculation performed depends on the contents of the specified index register:

1. If the left half of the contents of X is negative or 0, the right half of X is added to Y (from the instruction word) to yield an 18-bit local address.

This is consistent with indexing on a non-extended machine, and means, for example, that the usual AOBJN and stack pointer formats can be used for

2. If the left half of the contents of X is positive and non-0, the entire contents of X are added to Y (sign extended) to yield a 30-bit global address.

3. Indirection

1. Instruction format indirect word (IFIW): This word contains Y, X, and I fields of the same size and in the same position as instructions, i.e., in bits 13-35. Bit 1 must be 0 (its use is reserved for future hardware); bits 2-12 are not used. The effective address computation continues with the quantities in this word just as for the original instruction. That is, indexing may be specified and may be local or global depending on the left half of the index, and further indirection may be specified. Note that the default section for any local addresses produced from this indirect word will be the section from which the word itself was fetched.

0	1	2	5	6	17	18	35
!	!	!		!		!	!
!	!	!		!	<----- ----->		!
!	!	!		!	(section)	!	!
!	!	!		!		!	!
!	!	!		!		!	!
!	!	!		!		!	!

If indexing is specified in this indirect word, the entire contents of X are added to the 30-bit Y to produce a global address. A local address is never produced by this operation, and the type of operation is not dependent on the contents of X. Hence, either Y or C(X) may be used as an address or an offset within the extended address space just as is done in the 18-bit address space.

If further indirection is specified, the next indirect word is fetched from Y as modified by indexing if any. The next indirect word may be instruction format or extended format, and its interpretation does not depend on the format of the previous indirect word.

4. Some examples

1. Simple instruction reference within the current PC section:

```
3,,400/ MOVE T,1000 ;fetches from 3001000
        JRST 2000 ;jumps to 3002000
```

2. Local tables may be scanned with standard AOBJN loops:

```
        MOVSI X,-SIZ
LP:     CAMN T,TABL(X) ;TABL in current section
        JRST FOUND
        AOBJN X,LP
```

3. Global tables may be scanned with full address and index:

```
        MOVEI X,0
LP:     CAMN T,@[EFIW TABL,X] ;TABL(X) in ext
fmt
        JRST FOUND
        CAIGE X,SIZ-1
        AOJA X,LP
```

4. Subroutine argument pointer may be passed to subroutine in another section:

word in arglist:

```
        IFIW @VAR(X) ;indexing and indirecting
                        ;if specified will be relative
                        ;to the section in which this
                        ;pointer resides, i.e., the
```


,section of the caller

1.2.4 Immediate Instructions

All effective address computations yield a 30-bit address defaulting the section if necessary, as described above. Immediate instructions use only the low-order 18-bits of this as their operand however, setting the high-order 18 bits to 0. Hence, instructions such as MOVEI, CAI, etc. produce identical results regardless of the section in which they are executed.

Two immediate instructions are implemented which do retain the section field of their effective addresses.

1. XMOVEI (opcode 415) Extended Move Immediate:

This instruction loads the entire 30-bit effective address into the designated AC, setting bits 0-5 to 0. If no indexing or indirection is specified, the current PC section will appear in the section field of the result. This instruction would replace MOVEI in those cases where an address (rather than a small constant) is being loaded, and the full address is needed.

Example: calling a subroutine in another section (assuming arglist in same section as caller):

```
XMOVEI AP,ARGLIST
```

```
PUSHJ P,@(EFIW SUBR)
```

The subroutine could reference arguments as:

```
MOVE T,@1(AP)
```

or could construct argument addresses by:

```
XMOVEI T,@2(AP)
```

In both cases, the arglist pointer would be found in the caller's section because of the global address in AP. The actual section of the effective address is determined by the caller, and is implicitly the same as the caller if an IFIW is used as the arglist pointer, or is explicitly given if an EFIW is used.

2. XHLLI (opcode 501)

This instruction replaces the left half of the designated accumulator with the section number of its

effective address. It is convenient where global addresses must be constructed.

1.2.5 AC References

Any reference to a local address in the range 0-17(8) will be made to the hardware ACs. Also, any global reference to an address in section 1 in the range 0-17(8) (i.e., 1000000-1000017) will be made to the hardware ACs. Global references to locations 0-17(8) in any section other than section 1 will reference memory. Thus:

1. Simple addresses in the usual AC range will be reference ACs as expected, e.g., MOVE 2,3 will fetch from hardware AC 3 regardless of the current section.
2. To pass a global pointer to an AC, a section number of 1 must be included.
3. Very large arrays may lie across section boundaries; they will be referenced with global addresses which will always go to memory, never to the hardware ACs.
4. PC references are always considered local references; hence a jump instruction which yields an effective address of 0-17 in any section will cause code to be executed from the ACs.

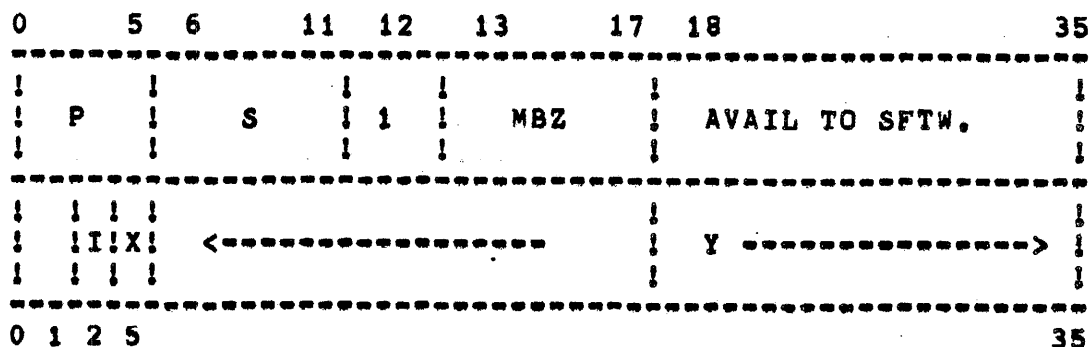
1.2.6 Special Case Instructions

In addition to the differences in effective address calculation, certain instructions are affected in other ways by extended addressing.

1. Instructions which store/load the PC; PUSHJ, POPJ, JSR, JSP. These instructions store a 30-bit PC without flags and with bits 0-5 of the destination word set to 0. POPJ restores the entire 30-bit PC from the stack word. JSA and JRA are not affected by extended addressing and store/load only 18 bits of PC. Hence they are not useful for inter-section calls.
2. Stack instructions (PUSHJ, POPJ, PUSH, POP, ADJSP) use a local or global address for the stack according to the contents of the stack register following the same convention as for indexing. That is, if the left half of the stack pointer is 0 or negative (prior to

incrementing or decrementing), a local address using the right half of the stack pointer is computed and used for the stack reference. If the left half of the stack pointer is positive and non-0, the entire word is taken as a global address. In the latter case, incrementing and decrementing of the stack pointer is done by adding or subtracting 1, not 1000001. Hence, a global stack for routines in many sections may be used in a similar manner to present stacks. Stack overflow and underflow protection would be done by making the pages before and after stack inaccessible since a space count field is not present in a global stack pointer.

3. Byte instructions. Two formats of byte pointer are recognized by the byte instructions. The non-extended format is identical to the present standard byte pointer format and is recognized if bit 12 (previously unused) is 0. If bit 12 is 1, a two-word extended byte pointer format is recognized which contains the fields as shown:



Note that the second word is identical to the Extended Format Indirect Word (EFIW). The right half of the first word is specifically reserved to software for byte counts, etc. Incrementing of this format of byte pointer is consistent with non-extended format; the P field is reduced until the end of the word is reached, whereupon the address in the second word is incremented. Incrementing may cause a carry from the word field to the section field of the address; hence extended byte arrays may lie across section boundaries.

4. Other new or modified instructions are LUUOs, BLT, XBLT, XCT, XJRSTF, XJEN, XPCW, XSFM. Some of these are valid only in exec mode. Consult the System Reference Manual or chapter 2.2 of the KL10 Functional Specifications for details.

1.2.7 Compatible Programming

It is possible to generate code which works correctly in both extended and non-extended environments. Such code may even utilize inter-section references when running in an extended environment; it is not limited to local addressing.

The basic rule is to observe the extended addressing rules in the construction or computation of:

1. Index words: be sure the left half is cleared or contains a negative quantity (e.g., a count) when setting up and using an index register. This will cause a local reference in the extended environment which will produce the same result as on the non-extended machine.
2. Indirect words: always set bit 0 of indirect words used in argument lists, etc. so as to produce local addresses.
3. Stack pointers: the most common present format (negative count in left half) works consistently under extended addressing; modifying a program to use an extended stack should require no change to stack instructions except if the code expects to find the processor flags in the stacked PC words.

When generating addresses to be passed to subroutines or used by other code, XMOVEI and XLLI instructions may be used. In the non-extended environment, these opcodes are SETMI (equivalent to MOVEI) and HLLI respectively, which always load 0 into the left half.

1.2.8 Program Architecture And Facilities

Ultimately, the extended addressing hardware in conjunction with the monitor should provide some of the power of general segmentation and some other useful conventions and facilities. The following are some of the ideas which have been advanced.

1. The fact that instructions are generally executed relative to the current sections suggests that subroutine libraries and facilities packages can be written which can be dynamically loaded into any available section and used by many programs. An important point to observe in connection with this and most of the following conventions is that programs must not be compiled with fixed section numbers built into

the code. Programs should be built so as to be loadable into any section and to request additional section allocations from the monitor as necessary. This is the only reasonable way to ensure that conflicting use of particular sections is avoided.

2. An entire section can be mapped by the monitor as quickly as a single page. Hence, an entire file (up to 256K) can be mapped into a section, and the data therein manipulated with ordinary instructions.
3. Programs can greatly reduce memory management logic by merely assuming very large sizes for all data bases. It is not however, very efficient to use many sections, each with only a small amount of data. A reasonable middle ground should be chosen.

BIAS CONTROL

1.1 BIAS CONTROL

The bias control knob provides for administrative control over scheduling algorithms. The knob allows a system administrator to bias the system according to the needs of the installation, to establish degrees of "fairness" among interactive and compute-bound users. This logical knob has settings from 1 to 20. The lower the setting of the knob, the more interactive jobs are favored. The higher the setting of the knob, the more compute-bound jobs are favored.

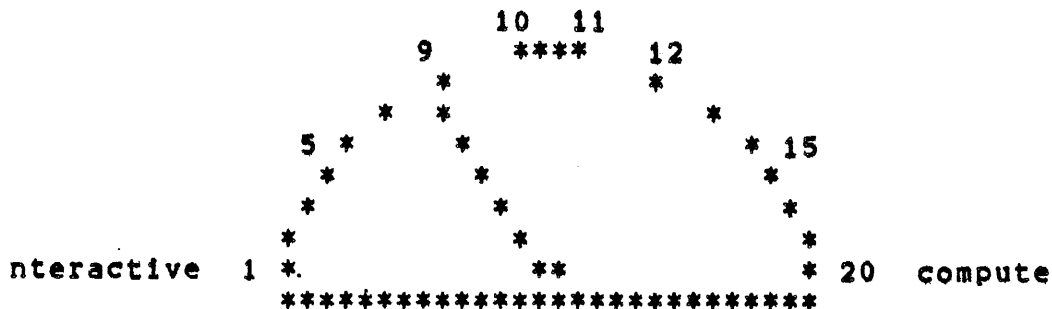


Figure of the bias control knob

Figure

2.1.1 USER INFORMATION

The bias control knob can be set through the use of a command to OPR, the operator interface. The command has the form:

```
OPR>SET SCHEDULER BIAS-CONTROL (to) n
```

where n is a decimal number from 1 to 20.

The bias knob can also be set through the use of a CONFIG command. The following command can be placed in 4-CONFIG.CMD:

```
BIAS n
```

where n is a decimal number from 1 to 20.

2.1.2 PROGRAMMING INFORMATION

Bias control can also be changed under program control through the use of the new SKED% JSYS. The .SKRBC function of this JSYS reads the bias control knob setting; the .SAKNB function sets the bias control knob setting.

2.1.3 IMPLEMENTATION

The value of the bias control knob is a displacement into a table of twenty entries. Each table entry represents a set of binary switches controlling a scheduler algorithm. Any binary value that causes an "on" setting (1) causes a bias in favor of interactive jobs while any binary switch that is "off" (0) is biased in favor of compute-bound jobs.

2.1.4 SWITCH SPECIFICS

The following is a graph of the current table (which starts at location SKFLGV) being used for the bias control knob.

Graph of Bias Settings

The following describes the current switches and their actions:

SK%CYT Bit 18, default setting is on.

Used for adjusting the length of the cycle times for the short and long cycle clocks. If 1, standard values for the cycle times are used (20 ms and 100 ms). If 0, the cycle times used are four times the standard cycle times (80 ms and 400 ms). This switch favors compute-bound jobs if off, since longer cycle times should reduce overhead, interactive jobs may not be allowed in during the longer cycles (they wait longer to be chosen), and a compute-bound job can hold the processor longer during longer cycles (which reduces context switching).

SK%IOC Bit 19, default setting is off; does not change.

Used to decide if a fork's quantum time should receive a charge for I/O. This is not really a bias between interactive and compute-bound processes in the strict sense. Jobs that do a large amount of disk file I/O are affected. Off indicates the charge is to be made; on eliminates this charge.

SK%HT1 Bit 20, default setting is on.

The SK%HT1 switch controls the number of processes with balance set hold time. If the switch is on, there is no more than one other process with balance set hold time, and less than half of memory is allocated, the process under consideration is given hold time. If the switch is off or the other conditions do not hold, the process is given hold time. The switch favors interactive processes if on by controlling the amount of hold time for a process. This keeps the balance set membership more dynamic which helps interactive processes back into the balance set.

SK%HT2 Bit 21, default setting is off.

If on, a process that enters the balance set after a process of higher priority has been skipped receives no balance set hold time. This favors interactive processes by not allowing processes of lower priority to run for long periods when processes of higher priority cannot fit into the balance set.

SK%HQR Bit 22, default setting is off.

Used to control "HQ disaster avoidance." If, during system

operation, the load average reaches a certain level (a load average between 4 and 6) low queue jobs (which tend to be compute-bound jobs) are no longer allowed into the balance set. If the load average goes higher (to between 6 and 9) low queue jobs are forced out of the balance set. If this switch is off, the code that causes the above to happen is skipped. Having this switch off favors compute-bound jobs since they can still get machine time when the load average is high.

Bits 23 and 24 are currently not used.

SK%RSQ Bit 25, default setting is on.

The SK%RSQ switch determines whether an overhead cycle is initiated when a process with higher priority than the running process becomes runnable as the result of disk I/O completion. If the switch is on and the conditions are met, the current process is dismissed and the higher priority one selected to run during the overhead cycle. This favors interactive processes since they can be run immediately after unblocking. If the switch is off, the dismissal occurs in the next normal overhead cycle.

K%RQ1 Bit 26, default setting is on.

Used for controlling the use of the interactive queues. If this switch is off when setting the queue level for an unblocking process, the interactive queues are not used. This favors compute-bound jobs if this switch is off since the interactive jobs do not get the increased priority and the extra quantum of the interactive queues.

K%TTP Bit 27, default setting is on.

Used to control the moving of processes to the interactive queues for TTY wait. If on, the added boost of moving the process to Q1 is given, which favors interactive processes with an increased priority.

K%WCF Bit 28, default setting is off.

Used to decide whether to decrease the wait credit of a previously blocked process based on the load average in determining the new queue. When off, the wait time is divided by the short-term load average to maintain fairness (that is, decrease the advantage gained for a wait if the load average is high). Other processes on the system have not fared that well while the process was blocked. When on,

interactive jobs are favored since they are not penalized for a high load average.

Bits 29 to 35 are not used.

It is interesting to see how values change from one setting of the knob to the next. Some changes of the knob may cause no change in the scheduler. For example, changing the knob from 1 to 4 currently causes no change in the scheduler. Changing from 16 to 17, however, causes three specific switches to be changed. Also, notice that the design of the table lends itself to changes in the future.

CHAPTER 3

CLASS SCHEDULING

.1 CLASS SCHEDULING

Class scheduling enables the administrator of a system to allocate the system according to classes of users.

In a particular system there may be classes of users that need large amounts of resources at a given time. There may also be a class of users that should not be given a large amount of resources. Class scheduling permits the administrator of a system to allocate system resources (specifically, CPU resources, which implies other resources such as I/O and memory) on the basis of class.

NOTE

Class scheduling is not required.
If it is not enabled, its
parameters are ignored.

Consider the following example:

DECSYSTEM-20 is used for computer courses at a university. This system has three definite classes of users:

1. Students that are taking the computer science courses
2. Faculty members that give the courses
3. Administrative users that do some processing during free time on the machine

They wish to divide up the machine as shown below:

Figure 2-3

Using class scheduling, the administrator would define three classes on the system. Each class would be defined to receive the indicated percentage and each user would be assigned to a class.

An important point should be realized concerning the divisions made in the example above. The machine is divided using class scheduling. If there is one student, three faculty members, and five administrators on the machine at a given time, the system would try to give the one student 75% of the machine, each of the faculty would receive 5% of the machine, and each of the administrators would receive only 2% of the machine. Or, another situation might have 25 students and only one member of the faculty and administrative groups at a given time. In the latter case, even though the students were allocated a larger percentage of the machine, each student would receive only 3% while the faculty user and the administrative user would receive 15% and 10%, respectively. It is therefore important to consider the size of a class of users when defining the percentage of the machine for that class.

As in the case where one student was logged on the system, it is possible that a particular class may sometimes not be able to use all of its allocated percentage. The amount of unused CPU resource in such a case is known as windfall. Another possible case that creates windfall is when all of the machine is not

allocated with class percentages. In either case, windfall can be either withheld or allocated. If withheld, the system runs idle rather than give a class more than its share. If windfall is allocated, the system distributes the excess CPU resource proportionately among the active processes.

NOTE

Withholding of windfall is not recommended. Its use should be restricted for at least two reasons:

1. If withholding windfall is enabled before all of the class percentages are defined correctly, a condition could exist where no one could do any useful work.
2. It wastes system resources.

1.1.1 USER CHARACTERISTICS

The first step in implementing class scheduling is to determine whether it is really needed. Class scheduling should not be used only because it is there. There should be a definite need for dividing up the machine. It is important to remember that while class scheduling may improve throughput for a particular class, it probably decreases throughput for the system as a whole. There is overhead involved with class scheduling and for this reason its use is not recommended with small machines (2020s and 4050s with less than 256K). It is not that class scheduling does not work with these machines -- the expense in overhead of using class scheduling on small machines probably outweighs the possible benefits.

1.1.2 CLASS ASSIGNMENT

When a need is established to use the class scheduler, the class members must be identified and assigned. Identifying the members requires grouping them according to characteristics. Assigning users to a class on the system requires using one of two

available methods.

To identify the classes requires some considerations of the mechanisms that the system uses. The system only supports eight classes, each specified on the system with a number from 0 to 7.

The characteristics of a class can be determined in a number of ways. In the previous example, the groupings were based on the obvious distinctions between students, faculty, and administrators. In the case of a timesharing bureau, the divisions could be made based on specific customers, with the end result being the sale of specified portions of the machine.

After the classes are defined, they need to be assigned on the system. There are two methods for accomplishing this.

The recommended method of assigning class members is to use the accounting system. Using this method, class membership becomes an attribute of a particular account. As the user logs in and is assigned an account, the user is also placed in the class that is identified as an attribute of that account. If the user is able to change accounts with the SET ACCOUNT command, the action of changing accounts may also change the class of the user.

NOTE

Using the accounting method
requires that account validation be
enabled.

An alternate method of assigning class membership is to use the access control job. By using this method, the user's class can be determined dynamically by a user-written policy program. This enables class assignments to be based on the current state of the system.

If the class is not assigned by one of these methods, the system assigns a user to a default class as the user logs in.

3.1.3 CLASS PERCENTAGES

The next step in implementing class scheduling is to determine the class percentages. The number of members in the class needs to be considered. As seen previously, the more members of a class with active jobs on the system, the less the percentage each member of the class receives. Factors unique to the installation, such as time of day, need to be considered. In the

In a university example, students may receive a very large percentage of the machine during the day, when they are doing a majority of their work. At night, however, the administration may need a large percentage of the machine for administrative work.

Finally, the mechanics of the implementation need to be considered. In assigning percentages to classes, the total percentage can be less than 100 % of the machine but not more than 100 %.

The percentages for classes can be defined in one of three ways: by using CONFIG commands, by OPR commands or by a policy program.

To define the class percentages using CONFIG commands, a command of the following form must be placed in 4-CONFIG.CMD:

```
CREATE n frac
```

where: n is the class number (0-7).

frac is the class percentage expressed as a fraction of 1 (0.00-.99).

There should be as many CREATE commands as there are classes on the system.

To define the class percentages using OPR commands, a command of the following form must be used:

```
OPR>SET SCHEDULER CLASS (number) n (to percent) per
```

where: n is the class number (0-7).

per is the class percentage (0-99)
(note that per is actual percentage in this case).

This command enables the operator to change class percentages during normal system operations.

The class percentages can also be changed using a policy program. This enables the percentages to be changed dynamically as based on administrative policy.

3.1.4 Batch Class

For added control of batch jobs, the administrator can require that batch jobs be placed in a specific class on the system. Using this option, batch jobs are placed in the specified batch class. This overrides the class assignment that would be

received if the user were to log in normally. This option is indicated either through a CONFIG command, an OPR command, or through the use of a policy program.

NOTE

This option requires class scheduling to be implemented using the accounting method. If the classes are assigned with the policy program, the batch class commands have no effect.

The CONFIG command has the form:

BATCH-CLASS n

where n is the class to be used for batch jobs (0-7).

The OPR command has the form:

OPR>SET SCHEDULER BATCH-CLASS n

where n is the class to be used for batch jobs (0-7).

3.1.5 TURNING CLASS SCHEDULING ON/OFF

Class scheduling can be turned on and off using either CONFIG commands, OPR commands, or a policy program. The CONFIG and OPR commands are described below.

Starting class scheduling with a CONFIG command requires two additional pieces of information. First, the disposition of the windfall must be determined by the requirements of the system. For example, if the requirements are such that no user is to receive more than his allocated share, windfall should be withheld.

NOTE

If windfall is withheld and there are errors in the CONFIG file, a situation could occur where no one has processor time allocated to

them.

Second, the method for determining class assignments, either by accounting or by policy program, should be identified. The CONFIG command has the form:

```
ENABLE CLASS-SCHEDULING ACCOUNTS      WITHHELD  
                                POLICY-PROGRAM  ALLOCATED
```

NOTE

This command must be the last class scheduling command in the CONFIG file. That is, it must follow all of the CREATE and BATCH commands. If it does not, all CREATE or BATCH commands that follow the ENABLE command will have no effect. If windfall is withheld and all of the CREATE commands follow the ENABLE command, no class will have percentages and no one will be able to do any useful work.

The OPR command for starting the class scheduler requires the same information that the CONFIG command requires. Both the method for assigning classes and the disposition of the windfall should be indicated. The OPR command has the form:

```
OPR>ENABLE CLASS-SCHEDULER  
      /CLASS-ASSIGNMENTS: ACCOUNTS  
                                POLICY-PROGRAM  
      /WINDFALL: ALLOCATED  
                   WITHHELD
```

OPR has an additional command for disabling the class scheduler:

```
OPR>DISABLE CLASS-SCHEDULER
```

1.1.6 IMPLEMENTATION

Class scheduling is used to influence the priority of a process. A process that is ahead of its percentage will tend to have its priority reduced and a process that is behind its percentage will tend to have its priority increased. Therefore, a process'

relation to its "target" percentage is a new metric that becomes part of the heuristic that dictates the transition rules among the queues. The following is a description of the new metrics and how they are employed.

A job's "utilization" is periodically computed by the process controller as a decaying average of the CPU time used by the processes of the job. The computation performed is

$$U(I+1) = U(I) * e^{(-t/C)} + F(1 - e^{(-t/C)})$$

where: $U(I)$ is the current process utilization.

F is the fraction of the CPU used by the process in the recent interval " t ".

t is the time since the last computation of U .

C is the "decay" interval. After time C , the utilization decays by e . This number represents the amount of "history" that the monitor uses in determining a process' behavior. The number is chosen to insure as uniform a behavior as possible.

The utilization function for each of the classes is the sum of the utilizations of the jobs in the class. The class utilization, CU , is used to compute the class "distance" (CD) from its ideal (target) utilization as follows:

$$CD = \frac{CP - CU}{CP}$$

where: CP is the class' ideal utilization. CP is the class percentage assigned when the class is defined.

Each job also has a job distance (JD) calculated as follows:

$$JD = \frac{\frac{CP}{N} - JU}{\frac{CP}{N}}$$

where: JU is the job utilization.

N is the "population" of the class of which this job is a member. N is currently the number of logged-in jobs belonging to the class.

The class distance and the job distance serve as a two-key sort for the GOLST for assigning priorities. CD is the primary key

and JD is the secondary key. In addition to this sorting, "specials" in the assigning of priorities ensure that certain critical and interactive processes are scheduled promptly (for example, NOSKED, CRSKED, high priority queue processes).

CHAPTER 4

EXECUTE-ONLY

.1 EXECUTE-ONLY

Execute-only provides the capability of protecting files from being changed or examined, while allowing the same files to be executed.

.1.1 DEFINITION

An execute-only file is one that cannot be copied or read in a normal manner, but can be run as a program. In order to provide this in TOPS-20, the following constraints must be placed on a file in order for it to be considered an execute-only file:

1. The file must be protected with EXECUTE access allowed but without READ access allowed.
2. The file cannot be read or written using any of the file-oriented monitor calls (i.e., SIN, SOUT, BIN, PMAP in referencing a file, etc.).
3. The file can be mapped into a process (via GET), but only in its entirety and only into a virgin process.

An execute-only process is a process that is created by performing a GET on an execute-only file. To insure security for the execute-only file and process, the execute-only process must also be restricted:

1. No other process can read from an execute-only process' address space or accumulators.

2. No other process can change any part of an execute-only process' context in such a way as to cause the execute-only process to reveal any part of its address space unintentionally.
3. An execute-only process must start at either its START or REENTER entry point (ENTRY VECTOR). Allowing a process to start elsewhere could cause it to reveal itself.
4. An inferior fork that is created from an execute-only process with the same map should be execute-only also.

Some related definitions are

Virgin process -- A process that has just been created (using CFORK) with none of its pages having been mapped and no operations having yet changed its context.

Context of a process -- The context of a process includes its address space, PC, ACs, interrupt system, traps, etc.

4.1.2 USER CHARACTERISTICS

An execute-only file is created by modifying the protection attribute of the file. This can be done by setting the protection field for the the desired class of users (owner, group or world) to FP%EX+FP%DIR, or 12 (octal). For example, to make a file execute-only for everyone except the owner of the file, set the protection to 771212. This indicates that the file can only be seen with the directory command and executed; the file cannot be read or written.

NOTE

Any file can be made execute-only. This can include data files, for example, but the results of such an exercise prevents any use of the file. It does not indicate that the file can only be read by an executing process.

An execute-only process is created by performing a GET on an execute-only file -- using a virgin process' address space. This prevents merging the execute-only file into an address space

which may already contain code that could reveal the address space of the execute-only file.

1.3 WHAT CAN AND CANNOT BE EXECUTE-ONLY

Due to the characteristics of execute-only, there are some restrictions as to where it can and cannot be used.

Most programs (.EXE files) can be protected execute-only. For example, saved COBOL, FORTRAN, and BASIC programs can be execute-only. Some utilities that can be execute-only include: UMPER, TV, EDIT, etc.

NOTE

Although COBOL and FORTRAN programs can be made execute-only, care must be exercised. The object time system (OTS) must be included in the EXE file. Otherwise, users can make their own "OTS" and define SYS: to cause that OTS to be used and to cause the execute-only process to reveal itself.

Here are some programs that cannot be execute-only. Some of the major ones are listed here along with an explanation of why they cannot be execute-only:

1. Any object time system such as FOROTS -- These are merged into an address space. This violates the restriction that confines the reading of an execute-only file into a virgin address space. Note: This does not imply that an execute-only process cannot bring in an OTS, but only that the OTS cannot be execute-only.
2. The TOPS-10 Compatability Package (PA1050) -- This cannot be execute-only for the same reason that an OTS cannot.
3. Any program that is brought in with the TOPS-10 UUO's RUN and GETSEG -- These UUOs require the program to be mapped into nonvirgin address space.

4. Any program that needs to be started at any location except its entry vector (START or REENTER address) -- To start elsewhere may cause an execute-only process to reveal itself.
5. Any program that uses TOPS-10-style "CCL starts" (starting at the start address plus one) -- Again, the program cannot start at a location other than that specified in the entry vector.
6. A compiler or linker invoked through the COMPILE/LOAD/EXECUTE/DEBUG commands -- These use the CCL start.

4.1.4 SOME OTHER RESTRICTIONS ON THE USER

Some other restrictions on using an execute-only file or process are extensions of the previous definitions.

EXAMINE, DEPOSIT, MERGE, DDT, SET ENTRY-VECTOR, SET PAGE-ACCESS, SET ADDRESS-BREAK commands do not work for execute-only processes. These commands would either cause the process to reveal itself or allow a modification to the process that could eventually cause the process to reveal itself.

The START command cannot be used with a start address argument for an execute-only process. The process must be started using its defined entry points (entry vector).

The INFORMATION (ABOUT) VERSION will only return the name of the program. The version information is part of the process address space that is not to be revealed.

It is important to note that the use of execute-only protection does not guarantee the security of the file. It is the programmer's responsibility to insure that the program does not reveal itself through programming mistakes that would, for example, allow the program to map itself to unsecured forks or use noncertified libraries.

WHEEL privileges can affect the use of execute-only files. An enabled WHEEL cannot create an execute-only process. It is when the process is initialized that it becomes execute-only. If the user has WHEEL privileges and they are enabled, the execute-only code in the initialization is bypassed since the enabled WHEEL has read access as well as execute access. For a user with WHEEL privileges enabled to create an execute-only process, privileges must first be disabled, the GET performed, and the privileges reenabled if needed.

1.1.5 IMPLEMENTATION

The implementation of execute-only involves some data in the JSB, checks in a number of JSYSs to see if the user is allowed to examine a particular process, and steps in a few JSYSs to set up execute-only processes.

1.1.5.1 DATA STRUCTURE -

In the JSB is a SYSFK area that has an entry for each process of the job. Three flags contain information about the process needed for execute-only. An entry appears as follows:

```

0 1 2 3 4      8 9      17 18      35
-----+-----+-----+-----+
! ! ! !      !      !      !
-----+-----+-----+-----+

```

bits	Register	Contents
		If set, indicates entry not in use, fork has been deleted
	SFEXO	If set, fork is execute-only
	SFNVG	If set, fork is not virgin
	SFGXO	If set, indicates fork can PMAP into execute-only forks because it is doing an execute-only GET
-8		Not used
-17	FKHCNT	Count of handles on a given fork
8-35		System fork number

1.1.5.2 ROUTINES -

Most of the use of the flags in the SYSFK area described above takes place in general routines. These routines are described below.

CHKNXS - CHECK if Not execute-only or Self -- Routine CHKNXS is a general test routine to determine if the process specified is either SELF or not an execute-only process. Otherwise, an illegal instruction trap occurs, returning the error FRKHXS -- "Illegal to manipulate an execute-only process". If the specified process is not execute-only, it will be declared nonvirgin by clearing the virgin process bit SFNVG.

SETEXO - SET process EXecute-Only -- This routine in FORK will

cause the selected process to become execute-only. If the process is not virgin, this does not succeed.

SETGXO/CLRGXO - SET/CLEAR GET execute-Only status -- These routines set and clear the execute-only GET bit (SFGXO) in the current process. These routines are called by GET to allow mapping into an execute-only process.

CLRVGN - CLEAR Virgin flag -- This routine sets SFNVG to indicate the process is not virgin.

SREADF - Set READF, read access and restricted-access -- This routine will set the read access bit (READF) and the restricted-access bit (FRKF) in the status word for the selected JFN. Also, the previous state of the FRKF flag is returned. This routine is required to allow GET to use BIN, SIN, PMAP, etc. to a file opened for execute-only access.

CREADF - Clear READF -- This routine undoes what SREADF did.

4.1.5.3 Major JSYSSs -

Certain JSYSSs have a major role in implementing execute-only. These JSYSSs are presented below.

CFORK

CFORK creates a virgin process if CR%ST (start process) and CR%MAP (give process same map as creating process) are not set. Note that loading parameters in the ACS using CR%ACS does not make this a nonvirgin process. Setting CR%ST and either CR%ACS or CR%MAP allows the process to execute code and, therefore, makes the process nonvirgin. Setting CR%ST without CR%MAP or CR%ACS seems rather useless.

CFORK creates an execute-only process if bit CR%MAP is set and the creating process is an execute-only process. This is the only way (besides GET) to create an execute-only process.

SFORK

SFORK has a switch to indicate that a process is to be continued, ignoring any PC change that may be specified in the right half of AC2. This is to insure that an execute-only process is not halted and continued from another location that could cause the process to reveal itself.

PMAP

It is illegal to specify an execute-only process as either the source or the destination in a PMAP call unless that execute-only process is executing the PMAP. If the executing process is doing a GET of an execute-only file (that is, if SFGXO is set), the process may map pages into any execute-only process.

SET

A GET call that addresses an execute-only process is illegal unless the calling process is the same execute-only process (SELF).

If the JFN specified in the GET call refers to a file for which the user only has execute access, the process specified must be a virgin process. GET must overcome two protection features to GET an execute-only file:

1. Reading the file without READ access (and not allowing others access at the same time).
2. Mapping pages from the file into an execute-only process.

GET must then perform the following steps:

1. Perform OPENF on the file for READ and EXECUTE (as in the past).
2. If the OPENF succeeds, proceed as usual since the file is not execute-only.
3. If the OPENF for READ and EXECUTE fails and either the specified process is not virgin or GT%ADR (address limits) was specified, return the error from the OPENF.
4. Perform OPENF for only EXECUTE access.
5. If Step 4 fails, return the error from the OPENF.
6. Lock the process structure.
7. Set the execute-only bit (SFEXO) in the destination process by calling SETEXO. If the destination process is not virgin, the execute-only bit will not be set and:
 - a. Unlock the process structure.
 - b. Mark the process as not virgin.
 - c. Return to Step 1. Since the process is not virgin, Step 3 will fail.

8. Unlock the process structure.
9. Remember that this will be an execute-only GET.
10. Disable interrupts within this process (NOINT). This is to protect the use of READ access to the file and the use of the execute-only GET bit (SFGXO).
11. Set READ access and restricted access in the JFN status for the selected JFN by calling SREADF.
12. Set the execute-only GET bit (SFGXO) in the executing process by calling SETGXO.
13. Perform the normal operations required to GET the file into the process.
14. If any errors occur, clean up but return error.
15. If this was an execute-only GET, clear the READ access and the restricted access in the JFN status by calling CREADF, clear the execute-only GET bit (SFGXO) in the executing process by calling CLRGXO, and enable process interrupts (OKINT).
16. Close the file (if possible) using CLOSF. Note that if pages are mapped from the file, it will not be closed, but will be left open with only EXECUTE access.

4.1.5.4 Minor JSYSSs -

There are a number of JSYSSs that either change a process' context or allow access to a process' address space. Since many of these functions are disallowed if the process involved is execute-only, calls to CHKNXS are contained in these JSYSSs. Routines that use CHKNXS either directly or indirectly and return the error "FRKH8: Illegal to manipulate an execute-only process" include:

ADBRK	AIC	DIC	DIR	EIR	IIC
RFACS	SAVE	SCVEC	SDVEC	SETER	SEVEC
SFACS	SFRKV	SIR	SIRCM	SPACS	SSAVE
STIW	TFORK	UTFRK			

OPENF returns fail if a attempt is made to open an execute-only file with illegal access specified.

CRJOB creates a virgin process as the top-level process. Thus, an execute-only program can be run as the top-level fork.

1.1.6 RESTRICTIONS AND LIMITATIONS

This section summarizes some restrictions and limitations of the current implementation of execute-only:

1. There is no hardware concealment of process pages. The KL10 hardware has the capability to conceal pages within a process from other parts of the same process. This feature, if used, would provide an additional feature that would allow non-execute-only programs to load execute-only programs (such as an execute-only object time system) into their address space.
2. There is no protection from a process revealing itself through its own carelessness.
3. Compilers and object time systems cannot be execute-only.
4. There is no page-by-page protection. Only an entire file can be execute-only.
5. Only disk files can be execute-only.
6. PA1050 must exist on physical disk for execute-only programs. A user cannot use his own version of PA1050 with execute-only programs.
7. An enabled WHEEL cannot create an execute-only process.
8. The version of an execute-only program cannot be read, since that information is stored only in the program's address space.

CHAPTER 5

MONITOR ADDRESS SPACE

5.1 MONITOR ADDRESS SPACE

During the development of Release 4, the TOPS-20 monitor ran out of address space. Extended addressing could not be used to solve the problem. Even though 2060 and ARPA monitors require extended addressing, there are DECSYSTEM-20s that do not support extended addressing (2020s and older 2040s and 2050s).

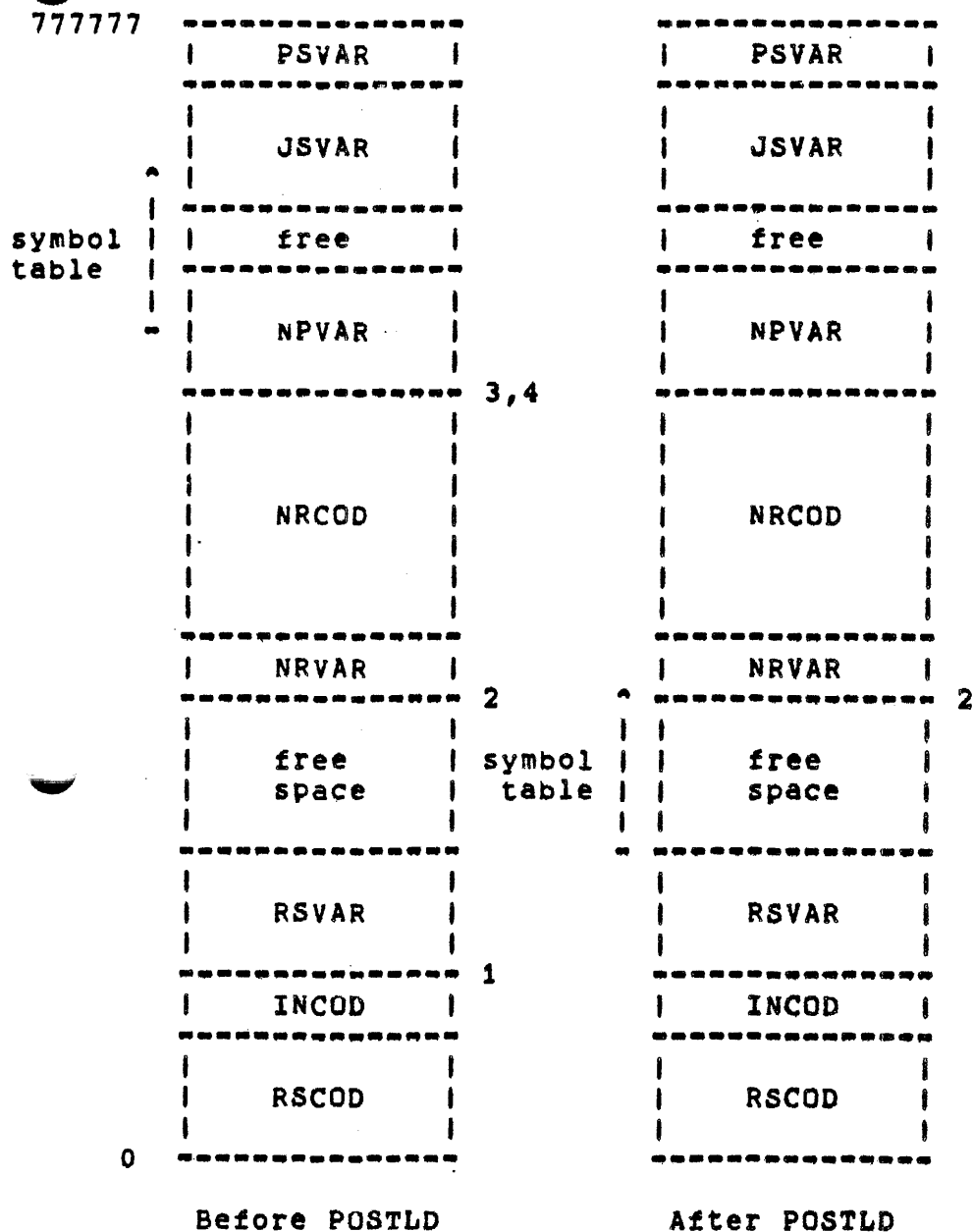
5.1.1 ADDRESS SPACE REARRANGEMENT

To solve the shortage of address space, the symbol table was moved out of the monitor address space and placed in an alternate address space. The symbol table was selected since its removal frees a large amount of space and it is used only infrequently under normal conditions. Making the symbol table harder to access does not degrade normal operations. In addition to the removal of the symbol table, other aspects of the monitor have been reorganized.

In order to make the changes easier to understand, the following short description of the address space layout for Release 3A is presented.

5.1.1.1 ADDRESS SPACE IN RELEASE 3A -

The address space in Release 3A of TOPS-20 had two layouts (see Figure 1) because of two different phases that occurred in the loading of the monitor. The difference between the two layouts was the location of the symbol table. The symbol table was moved by POSTLD from its original location to immediately after RSVAR.



Notes:

1. POSTCD
2. PPVAR
3. PPVAR
4. BGPTR

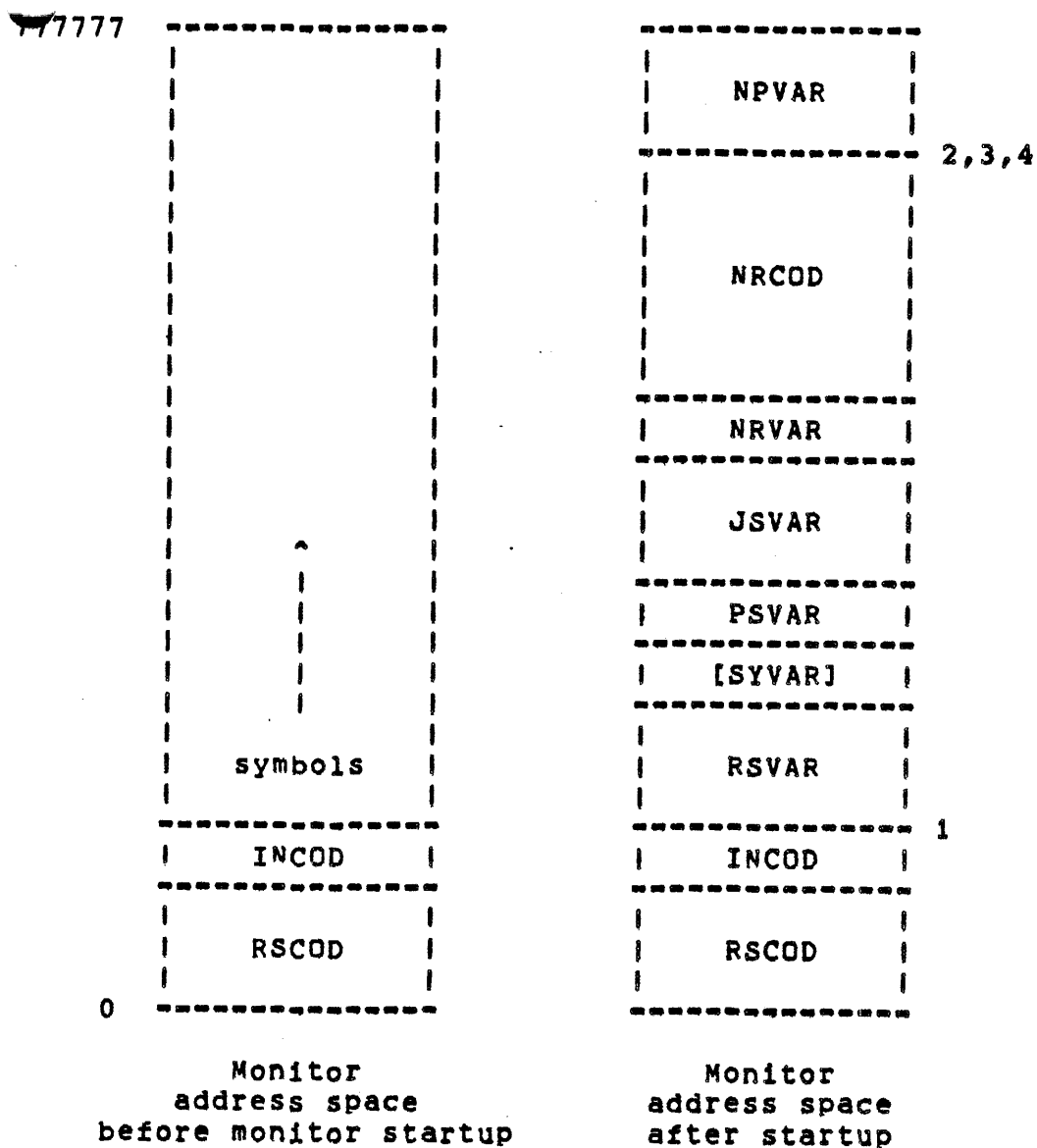
Figure 1 Release 3A Address Space

5.1.1.2 ADDRESS SPACE IN RELEASE 4 -

In order for the symbols to be available from monitor startup time, they must reside in the MONITR.EXE file and be read in by BOOT. Any scheme to put them into a different file would mean that they would not be available until after all of the monitor's disk mounting and other startup code had been executed. This would mean that EDDT would have no symbols for debugging. To put the symbols into the EXE file without overwriting code or data, an all-zero area big enough to hold them must be used.

Based on this, the monitor's address space is laid out as indicated in Figure 2. BOOT reads in RSCOD (resident monitor) and INCOD (which contains initialization code and EDDT) and the symbols. The symbols are initially read into the area where the psects PPVAR and RSVAR will ultimately reside. STG then moves the symbol table up to immediately following the SYVAR psect in physical memory. STG does this to free up the RSVAR and PPVAR psects so that MMAP and the CSTs can be set up to turn on paging.

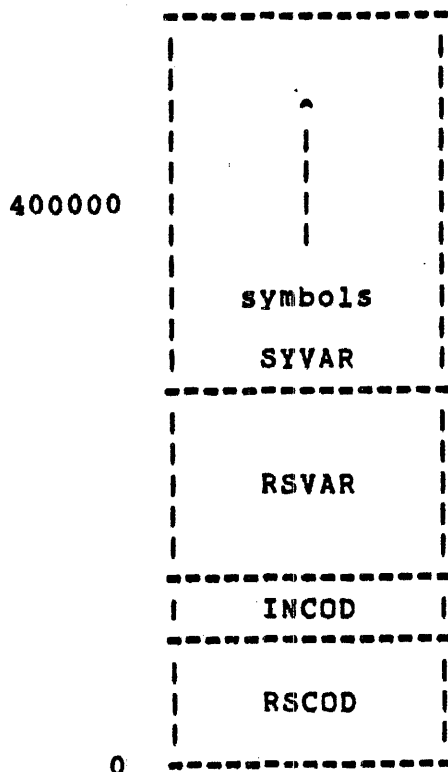
Later, PGRINI sets up SUMMAP to point to an alternate address space containing EDDT, the symbols, and parts of the monitor. EDDT and the symbols live at the same addresses in this virtual address space as they do in physical memory. (See Figure 3.) The symbols do not appear in the monitor's main address space at all.



Notes:

1. PPVAR 2. BGSTR 3. BGPTR 4. POSTCD

Figure 2 Release 4 Address Space



Notes:

1. PPVAR
2. BGSTR
3. BGPTR
4. POSTCD

Figure 3 Alternate Address Space

5.1.2 MODULE CHANGES

With the changes to implement the alternate address space (which is also often referred to as hidden symbol processing) some modules now function differently.

BOOT

BOOT is read in by the front end at a predetermined physical address (currently on top of the RSCOD psect) and started. The first thing BOOT does is find the highest 20 pages in section 0 of physical memory, and move itself there. It then sets up a mapping that is straight physical to virtual, except that it always maps itself at the end of virtual memory (pages 760000 and beyond).

EDDT

With Release 4 EDDT now has two conditions. In the first condition, when the monitor is loaded by BOOT but not started, the symbols are in the monitor's virtual address space and EDDT accesses them much as it did in past releases. In the second condition, the monitor has been started and EDDT must reference the symbols using the alternate mapping.

The alternate mapping is accomplished using an alternate page table. To access the symbols, EDDT changes the page table pointers at MSECTB and MSECTB+1 (for sections 0 and 1). The old page table pointers are placed in OSECTB and OSECTB+1. New page table pointers are retrieved from SSECTB and SSECTB+1.

DDT

DDT accesses symbols when it needs them by mapping needed pages of the alternate address space. This is accomplished through the use of a new monitor subroutine called .IMOPR (Internal Monitor Operation).

SNOOP JSYS

The SNOOP JSYS accesses symbols in the alternate address space by using .IMOPR in the same fashion as MDDT.

1.3 LIST OF MONITOR PSECTS

The following is a list of all of the TOPS-20 monitor's psects, along with a short description of each. Note that each psect whose name ends in VAR is all-zero, and is not filled in until the monitor starts running.

1. Page 0 (and 1 on 2020s) -- Not really a psect; these pages are loaded with LOC statements and are full of miscellaneous communication areas, flags, etc.
2. RSCOD (Resident code) -- This psect contains the code and data that can never be swapped out. This psect is hard-wired into the monitor as the first one, and the location MONCOR contains the last page number in it.
3. INCOD (Initialization code) -- This psect contains some routines used only during monitor initialization (including the 143SG dialog). It also contains Exec DDT. This psect is locked when the monitor is started, but gets unlocked at GETSWM unless EDDT is needed.

4. PPVAR (Per-processor variables) -- This psect contains nothing. It is used to reserve a few slots in MMAP for use in setting up temporary mappings to memory pages. APRSRV uses these map slots to recover from parity errors, for example.
5. RSVAR (Resident variables) -- This psect contains the monitor's variables that must always remain resident, including the EPT, MMAP, and the CSTs.
6. SYVAR (Symbol variables) -- This psect contains everything that should appear only in EDDT's alternate address space (for example, symbols), and not in the monitor's normal address space. The symbols are appended to this psect early in the monitor's initialization, and then, both the symbols and the psect are "hidden." This psect is of zero size if the HIDSYM conditional is not set.
7. PSVAR (PSB variables) -- This psect contains the PSB.
8. JSVAR (JSB variables) -- This psect contains the JSB.
9. NRVAR (Nonresident variables) -- This psect contains monitor data locations that can be swapped.
10. NRCOD (Nonresident code) -- This psect contains the monitor code that can be swapped, including the processing routines for all the JSYSSs. This psect is usually write-locked.
11. BGSTR (Bugstrings) -- This psect contains ASCIZ strings that describe each BUGINF, BUGCHK and BUGHLT. Like NRCOD, this psect is swappable and write-locked.
12. BGPTR (Bugpointers) -- This psect contains a few words for each BUGxxx, including the additional arguments and pointer to the corresponding Bugstring. This psect is also swappable and write-locked.
13. NPVAR (Nonresident page variables) -- This psect contains monitor variables that are allocated a page at a time and can be swapped. One of the main features of this psect is the resident free pool (RESFRP), whose pages are locked in memory one at a time as they are allocated.

5.1.4 HINTS ON MONITOR BUILDING

The following hints appear below as an aid to those who must rebuild monitors.

WHY THE PSECTS ARE WHERE THEY ARE

Several rules must be followed when rearranging the psects:

1. BOOT reads the monitor in around itself in virtual memory. Since BOOT needs to remain mapped and functioning until the swappable monitor has been read in and started, it must not lie in any part of the monitor's virtual address space that will be used by the monitor's initialization code. For this reason, the only three areas that can be used are a gap between psects, the NRVAR psect, or the last part of the NPVAR psect (the first part contains the resident free pool that is used by the

disk mounting code and the swapper). The last area is the one currently used.

2. RSCOD must be first; it checks to see if an address lies in RSCOD are just CAMLE MONCOR, which will not work if any psect falls below RSCOD.
3. All of the psects in the part of the monitor that is started first and that reads in the swappable monitor must be first. In addition, the last item in that part of the monitor must be the symbol table, which includes the RSCOD, INCOD, PPVAR, RSVAR, and SYVAR psects. The code in these psects must work before any swapping or paging can occur; therefore, they must all be low enough to fit in physical memory on the smallest configuration supported. Also, BOOT stops reading in the resident monitor when it hits the end of the symbol table, so the symbol table must be the last psect in this group.
4. The group of nonzero psects that swap are treated as a unit by certain parts of the monitor, and should therefore be together. These psects are NRCOD, BGSTR, and BGPTR.
5. PSVAR, JSVAR, POSTCD, NRVAR, and NPVAR can generally be put anywhere. They have been moved in the past with success.

5.1.5 WHICH PSECTS CAN OVERLAP

The POSTCD psect can overlap any xxVAR psect, since it will be gone by the time MONITR.EXE is generated. POSTCD is currently allocated its own three pages to avoid psect overlap warnings from LINK (a cosmetic precaution only).

With hidden symbol processing, the SYVAR psect and the symbol table can overlap any other xxVAR psects. The SYVAR psect is currently allocated its own page to avoid psect overlap warning messages from LINK.

If BUGSTF is not set and the bugstrings and bugpointers are not present in the running monitor, the BGSTR and BGPTR psects can overlap any xxVAR psect. In the current monitor, they would probably be overlapped with the NPVAR psect, which immediately follows them.

No other psect overlaps can be allowed without breaking the monitor.

5.1.6 HOW TO CONTROL THE SYMBOL TABLE ORIGIN

The psect origins of all the real psects are controlled by /SET switches in the LINK .CCL file. The symbol table, however, must be controlled in a more indirect way, via the LINK switches /SYMSEG and /UPTO.

The /SYMSEG:PSECT:name switch directs LINK to append the symbol table to the named psect. If symbols are hidden, the symbols should be appended to INCOD; if not, they should be appended to SYVAR. The symbols normally start 200 words after the end of the preceding psect. The extra 200 words are the PAT.. area.

The /UPTO:address switch tells LINK the highest legal address for the symbol table. If there are enough symbols to make the symbol table attempt to exceed the specified address, LINK will output a warning message and truncate the symbol table. (The resulting monitor should still run if this occurs.) The address should be set to one less than the base of the first psect that the symbol table cannot overlap.

5.1.7 SAMPLE OUTPUT FROM LINKING A MONITOR

The following is part of the output from linking a 2020 monitor; it shows both a new format for the output and the new address layout.

Monitor address space:

Psect	Start	End	Length	Free	Limit
RSCOD	2000	63633	61634	6144	
INCOD	72000	112634	20635	1143	
PPVAR	114000	117777	4000	0	
RSVAR	120000	214226	74227	13551	
SYVAR	230000	230777	1000	7000	
PSVAR	240000	325777	66000	2000	
JSVAR	330000	417777	70000	0	
NRVAR	420000	441426	21427	2351	
NRCOD	444000	660347	214350	13430	
BGSTR	674000	702602	6603	1175	
BGPTR	704000	704714	715	1063	
POSTCD	706000	710217	2220	560	
NPVAR	711000	772777	62000	5000	777777
Loaded symbols	112735	251750	137014	172027	NRCOD
Runtime symbols	231000	370013	137014	320764	NPVAR

The symbols will be moved to their runtime area right after the SYVAR psect by STG early in the monitor's initialization.

There are 57 (octal) free pages, not counting symbols.

% Runtime symbols must end by 360000 in order to run on a 128K system.

Writing sorted bug list to file BUGSTRINGS.TXT.1
Saving monitor as SMONITR.EXE.1

CHAPTER 6

MONITOR MODULES

MONITOR MODULES

The source code for the TOPS-20 monitor is in the form of macro files. Each file contains code pertaining to a particular function or device. These files are assembled into monitor modules. Some of the modules are assembled from only one file, which may include searching certain universal files. But some of the modules are a combination of files. For example, consider the module TTYSRV. The file TTYSRV.MAC contains the terminal service functions. The module TTYSRV, however, is built of four files: TTYSRV.MAC, KLPRE.MAC which indicates the terminal service is for a KL, TTFEDV.MAC which contains information about the lines being connected to a front-end, and TTPTDV.MAC which contains the code required to support pseudo terminals.

All modules are not contained in every TOPS-20 monitor. Some modules are built for certain hardware and/or software. For example, the TTYSRV module described above is contained in a monitor built for a KL10, but the TTYSSM module contains the terminal information needed for a monitor that is to run on a 020.

The following describes the modules contained in the Release 4 monitor.

APRSRV This is the processor dependent service module for the KL10. It contains the initialization code for paging, MUUO handlers, and the priority interrupt system as well as for the clocks, APR, and DTE devices. Interrupt handling for these devices, pager control routines, and pre and post JSYS handling is also performed here.

KLSSM This is the processor dependent service module for the KS10. It performs the same functions as APRSRV (excluding service for devices that do not exist on the KS10 such as

DTE service).

CDESRV Card punch service.

CDESM Card reader service for the KS10 processor.

CDESRV Card reader service for the KL10 processor.

COMND Code for the COMND JSYS.

DATIME Code for the date and time conversion JSYSS.

DEVICE Device initialization and lookup code.

DIAG This module contains the code to support the DIAG JSYS for the KL10.

DIRECT Directory management code.

DISC This module contains the pre-PHYSIO disk dependent routines for I/O JSYSS and a dispatch table of vectored addresses, DSKDTB, which points to them.

DSKALC Drive type independent code for disk block allocation, including swapping space allocation, front-end file system definition, and structure definition.

DIESM Dummy replacement code for DTESRV for the KS10.

DIESRV DTE service driver; protocol handler for requests to and from the front-end.

EEILIM This module performs the same functions as FILINI using extended directory support.

ENO This module implements the ENQ/DEQ facility to control simultaneous access to user specified sharable resources.

EESRV Device code for FE devices. This code contains the device-dependent routines for the FE pseudo devices FE0-FE3.

EILINI This module contains code to initialize the file system at system startup.

EILMSC This module contains miscellaneous routines for the PTY, TTY string (includes break mask and field width support) and null I/O devices and also includes a device dispatch table for each of these devices.

EILNSR This module contains the filesystem interface to NSP. It includes the device dispatch tables for SRV: and DCN:.

EQRK Contains the fork controlling JSYSS and support code.

EREL Job storage free area management.

ENTILI Utility module which contains routines to copy strings to/from JSBs, routines to retrieve/change connected structure and directory information and routines to get a yes/no answer from CTY.

ETJEN Contains the code for GTJFN, and the JSYSS which support lookup, recognition, and creation of file names.

IMAN22 AN22 driver for the 2020.

IMRANX IMP driver for AN10.

IMPDY This module contains the Interface-Message-Protocol (IMP) device independent code. It runs cyclically as a separate fork (i.e., under JOB0) and handles the interface to the ARPA network by monitoring network activity and managing the message queues.

IMZPAR This is the parameter file for the IMP modules.

ID Contains most of the device-independent sequential, random, and dump input/output routines for BIN, BOUT, SIN, SOUT, DUMPI, and DUMPO.

IECE Code for the system interprocess communications facility.

ISYSA Random JSYSS for system and directory access, device allocation, job parameter settings, system accounting and file/fork mapping.

ISYSE Contains code which implements various file system JSYSSs.

IDESRV This module provides support for DUP11's with a KMC11 for NSPSRV on a 2020.

IDINIT This module at load time defines storage PCs for the JSYS dispatch table, JSTAB.

INEPR Lineprinter service for the KL10.

INESM Lineprinter service for the KS10.

OGNAM Contains the logical name definition and recognition JSYSSs and routines.

OOKUP Device independent file name lookup.

AGTAP This module contains the pre-PHYSD magtape-dependent routines for I/O JSYSSs and a dispatch table of vectored addresses, MTADTB, which points to them.

EXEC This module contains the MINI-EXEC (MX) which is a limited

command interpreter for certain system loading/maintenance functions, and swappable monitor bootstrap procedures. It is part of the swappable monitor and also contains many JSYS routines.

MEILN Floating point input and conversion JSYSSs.

MELOUT Floating point output and conversion JSYSSs.

MSTR Contains the code to implement the mountable structure JSYS, MSTR.

NETWORK This module contains the interface for all standard I/O JSYSSs that communicate with the ARPA-network. It also provides a finite state machine of various events associated with a connection for the network control program (NSP).

NSRINT Network Services Protocol Internal Interface.

NSRPAR This parameter module contains data structures and symbol definitions required by NSPSRV. In particular, NSPPAR contains the definitions of the logical link blocks.

NSRSRV This module contains the control routines and JSYS interfaces for the message level protocol of DECNET known as NSP (Network Services Protocol), which allows communication between processes on hosts by means of logical links.

PAGEM Page management code; core management routines, swapper routines, pager trap logic, OFN control, and CST and SPT initialization.

RHYH11 Channel dependent code for RH11 controller.

RHYH2 Channel dependent code for RH20 controller at direct I/O level.

RHYM2 Device dependent code for TM02/TM45 magtapes at direct I/O level.

RHYR4 Device dependent code for RP04/RP06 disks at direct I/O level.

RHYRPAR Universal file for PHYSIO and associated modules. It contains the definitions for the Channel Data Block, Channel Dispatch Table, Unit Data Block, Unit Dispatch Table, and the Input/Output Request Block.

PHYSIO This module handles the channel and driver I/O routines. It is responsible for queueing I/O requests into their proper queue, choosing the "best" request for seeking and/or transferring and starting I/O.

PHYX2 Device dependent code for DX20.

PLT Plotter service.

POSTLD This code runs immediately following the loading of the monitor, and performs functions outside the capabilities of LINK. It builds the MONITR.EXE file, writes a BUGSTR text file and deletes itself from core.

PROKL Parameter file indicating KL10.

PROKS Parameter file indicating KS10.

PROLOG This is a file of parameters, storage assignments, and macro definitions. The major regions of the monitor address space are defined as well as macros affecting PI bug strings, pseudo-interrupts, and scheduling. All PSB and JSB storage defined by the monitor at assembly time is specified here. All of the BUGXXX definitions are included in this module from the file BUGS.MAC.

PTP Paper tape punch service.

PTR Paper tape reader service.

SCHED This module contains the Channel 7 Interrupt routine (which performs context switching), the process controller, the working set manager, the job/fork initialization/dismiss routines, and the Program Software Interrupt (PSI) analysis and resolution routines.

SERCD This module contains the error codes and fields for SYSERR, a program which produces hardware performance reports for field service personnel.

IG The bulk of the monitor storage, both resident and non-resident, is defined in this module.

SWACL This is the swapping space allocator which handles a device of some number (SWPSEC) of sectors, and some number (DRMMXB) of tracks. It has a resident bit table which is used to allocate swapping storage.

YSERR Error reporting module for field service (not to be confused with the SYSERR program used to read the error information).

ARE This module contains the tape handler and record processor.

IMER This module implements the TIMER JSYS and all of its support.

TYM This is the terminal service module for a KS10 ARPA monitor. See TTYSRV.

TXSNV This is the terminal service module for a KL10 ARPA

monitor. See TTYSRV.

TTYSRV This is the terminal service module for the KL10. This module contains the TTY I/O drivers, special control character conversion routines, terminal JSYS routines and the interface to the primary and secondary protocols in DTESRV. Its device dispatch table is contained in FILMSC.

TTYSM This is the terminal service module for the KS10. See TTYSRV.

VERSIQ Version information for the monitor.

CHAPTER 7

WATCH

7.1 WATCH

WATCH is a TOPS-20 data collection tool that can be used to gather the information necessary to analyze both system and job performance. WATCH periodically samples many system variables, writing them in a format that is usable for analysis. Collecting WATCH statistics is beneficial whenever the system is running. Such statistics are often useful when usage trends are being analyzed in order to plan system growth. Any user can run WATCH and obtain most of the system information and some of the job information. These statistics are normally sufficient for determining overall system performance and for spotting short- and long-term usage trends. Expanded system and job information is available for users who are running with WHEEL or OPERATOR privileges enabled. This expanded set of statistics provides the much more detailed information that is often required to observe and tune the workload of an individual application.

7.1.1 OUTPUT

The WATCH output consists of nine different display sections:

1. Heading -- This section contains the date, time, number of jobs logged in, and the time interval over which the data sample was collected.
2. System Statistics -- This section contains system-wide statistics that reflect the resource utilization of the CPU, disk, and memory.

3. Load Averages -- Load averages indicate the number of runnable processes over specified intervals. This section indicates the load average for the system, for the interactive and computational queues, and for each class (when class scheduling is in use).
4. Directory Cache -- A cache of the most recently used directories is kept by the monitor. This section displays statistics that indicate the usefulness of this cache.
5. Normal Per-Job Information -- Per-job statistics that relate the amount of CPU resource distributed to each job are displayed along with statistics concerning class utilization (when the class scheduler is in use).
6. Expanded Per-Job Information -- In addition to the CPU information, this display presents many statistics that show the states in which the job spent time and how large the job is. The display also provides disk and swapping information.
7. System Utilization Statistics -- The system utilization statistics include a summary of the expanded per-job section, additional system statistics, and computations of several key variables.
8. Disk I/O -- Disk I/O statistics are displayed on a per-drive basis. Included in these statistics are the number of seeks, reads, and writes performed by each drive.
9. Tune Mode Display -- This display is a single line that contains some of the more revealing system statistics and summary statistics from the system utilization section. It is a useful "quick and dirty" display for users who are monitoring changes in the system load.

In addition to the above displays, data record output can also be requested. This form permits further computer analysis of data output from WATCH. Consider the following example of system statistics:

SUMMARY at 8-Oct-79 10:52:21
for an interval of 1:59.9 with 54 active jobs.

USED: 21.1	IDLE: 42.7	SWPW: 2.0	SKED: 2.9
SUSE: 20.7	TCOR: 0.1	FILW: 29.6	BGND: 1.6
VTRP: 10.2	NCOR: 2.00	AJBL: 21.03	NREM: 0
TRAP: 1.2	NRUN: 0.9	NBAL: 0.9	NWSM: 66.6
BSWT: 0.5	DSKR: 18.0	DSKW: 9.4	SWPR: 5.7
VL0D: 13.52	CTXS: 21.9	UPGS: 1966.	FPGS: 205.
DMRD: 1.1	DMWR: 0.7	DKRD: 3.0	DKWR: 2.5
FTIN: 12.8	TTOU: 242.	WAKE: 13.7	TTCC: 2.50
FDIO: 7.4	RPQS: 3.3	GCCW: 1.6	XGCW: 0

(NOB: 11

QUEUE DISTRIBUTION PERCENTAGE:

0.20 6.95 10.28 3.64 0.00 0.00

The output data record for this WATCH output would have the following form:

```
0100110/08/79 10:52:2100119.90540021.100042.700002.000002.90
020.700000.100029.600001.600010.200002.000021.030000.000001
200000.900000.900066.600000.500018.000009.400005.700013.520
021.901966.200205.400001.100000.700003.000002.500012.800242.
000013.700002.500007.400003.300001.600000.00000110000.200006
950010.280003.640000.000000.000000.000000.000000.000000.000
```

All of the variables and the format of the data records are described in the "WATCH VARIABLES" section that follows.

.1.2 RUNNING WATCH

WATCH can be run by all users, though users with OPERATOR or HEEL privileges enabled can obtain more information than others. WATCH is run by typing either "WATCH" or "R WATCH". When WATCH starts, it identifies itself with a message like:

WATCH 4(3), /H for help.

Information is then requested in the following order:

Output to file:

Would you like to output data records? (Y/N)

Output file for data records:

Print monitor statistics?

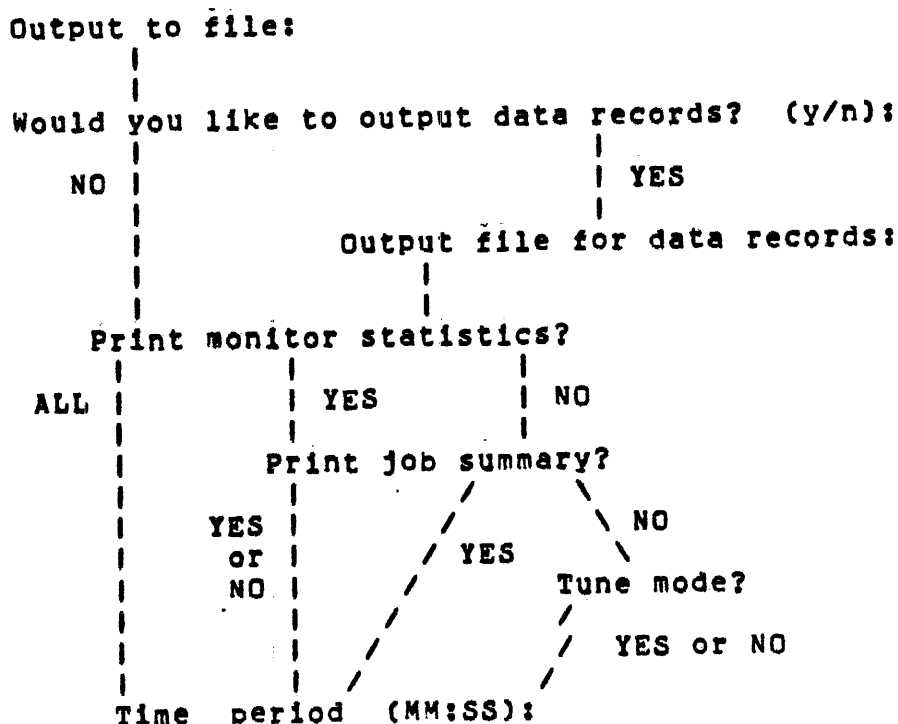
Print job summary?

Run mode?

Time period (MM:SS):

Some of these requests are not made if previous answers indicate

that the information is unnecessary. The following diagram outlines the order of the questions:



The output of data records is indicated with the second and third questions. The output display received is as follows:

1. Headings and load average displays are always received except in tune mode.
2. "Yes" to the monitor statistics question causes the system statistics display to be output.
3. "Yes" to the job summary question causes the normal per-job statistics display to be output.
4. "All" to the monitor statistics question causes all displays but the tune mode display to be output.
5. "Yes" to the tune mode question (which occurs only when "no" is given for both the monitor and job summary questions) causes the tune mode display to be output.

After the user enters the necessary information, WATCH displays the message:

WATCH IN OPERATION --

and then takes its first sample. This message is seen immediately after entering the interval time unless the user is

requesting displays that require privileges (extended and tune mode). Then, a 10-30 second pause occurs (depending on system load) while the SNOOP breakpoints necessary to collect the information are inserted into the TOPS-20 monitor.

When the user desires to stop collecting statistics, the following procedure should be followed:

```
^C
^C
CLOSE
RESET
```

At this point, the output can be printed.

7.1.3 SOME INDICATORS

When variables have the values indicated in the following list, the system is usually in "balance." Since it is quite possible for one variable to appear in balance, while others are not, this information is only a guideline.

1. NCOR = 30 PER MINUTE -- Expensive if higher.
2. NRUN/NBAL = 1 -- All processes wanting to run can fit in memory.
3. SWPR less than 20% -- Swap reads/writes are overhead and thus should be (ideally) a small component of disk usage.
4. SWPW close to 0 -- Since this variable represents processes waiting for memory when no others can run, utilization of the system can normally be increased by adding memory until this is a small value. However, if the load has a large I/O component, additional memory may merely shift the CPU idle time from SWPW to FILW.
5. SKED = 15% -- Scheduler overhead detracts from cycles going to user programs. Programs that do very little work each time they are scheduled generally drive this value up. If this value is high, it is important to determine if there is a set of applications which could be reprogrammed in order to do more work between interactions. Programs that become active as each character is typed (like some screen formatting software) should be viewed with suspicion.

6. FILW close to 0 -- This is CPU idle time caused by processes waiting for disk I/O to complete. More memory permits larger numbers of programs to be resident. At other times, reconfiguring the disk access patterns to spread the disk I/O more evenly across the disks and channels lowers this value.
7. BSWT/NBAL small -- If a large proportion of processes in memory are waiting on the disk, the CPU is not being utilized.
8. NREM = 0 -- Since this counts the times runnable processes are removed from the balance set, performance is best when it is zero. Performance degrades rapidly as the value increases.
9. FPGS large -- If the number of free pages is low, the system needs to expend resources to garbage collect more often. This statistic, along with NREM, can be used to indicate a system overload.
10. DMRD+DMWR less than 20 per second -- Because drum reads/writes utilize a percentage of the disk system's bandwidth, higher throughput is possible when swapping is low. Normally, swapping of less than 30 pages per second does not cause any visible effect. If the normal load contains a large amount of user disk I/O, swapping at rates higher than 20 will decrease the system throughput. If the normal load is mostly interactive or computational, higher swapping rates can be sustained.

7.1.4 WATCH VARIABLES

The following is a list of the displays produced by WATCH. Beside each display type is the output data record type used for that display's information when data records are written.

1. Heading (All record types)
2. System Statistics (01)
3. Load Average (02)
4. Directory Cache (05)
5. Normal Per-Job Information (04)
6. Expanded Per-Job Information (03)

7. System Utilization Statistics (05)
8. Disk I/O Statistics (05)
9. Tune Mode Statistics (06)

In the following sections, each display is described and the format of the output data records is given. In the formats of the data records, the length of the fields is measured in characters (all fields in the records are ASCII characters). The form of each field is shown as a COBOL PICTURE (9 represents a numeric character, X an alphanumeric character). The first field of each record is the record type; the second field is the record sequence number. For record types 01, 02, and 05, the record sequence number is always "001".

1.4.1 Heading -

The following is an example of the heading display. It shows the date and time, the length of the interval, and the number of sessions logged in. The interval shown is nearly equal to that specified by the user.

```
SUMMARY at 6-Aug-79 09:28:23
for an interval of 0:11.3 with 52 active jobs.
```

The heading information appears in fields 3, 4, and 5 of all output data records.

1.4.2 System Statistics -

The following example of the system statistics display may be referenced while reading the descriptions of the variables.

```
USED: 87.6 IDLE: 0.0 SWPW: 0.0 SKED: 10.0
SUSE: 80.0 TCOR: 0.1 FILW: 0.8 BGND: 1.8
NTRP: 22.3 NCOR: 2.02 AJBL: 58.44 NREM: 0
TRAP: 3.2 NRUN: 5.7 NBAL: 5.7 NWSM: 61.0
BSWT: 2.8 DSKR: 23.0 DSKW: 4.8 SWPR: 6.4
NL0D: 16.12 CTXS: 64.1 UPGS: 1992. FPGS: 147.
DMRD: 4.1 DMWR: 4.7 DKRD: 10.7 DKWR: 4.3
TY : 11.0 TTOU: 401. WAKE: 18.3 TTCC: 1.51
TDIO: 23.8 RPQS: 4.4 GCCW: 6.4 XGCW: 0
KNOB: 11
QUEUE DISTRIBUTION PERCENTAGE:
```

0.19 20.48 21.28 11.04 2.69 31.74

The statistics in this display are expressed either as percentages (%), as averages (AV), or as rates (Px). The rates are in units per minute (PM) or in units per second (PS).

The display includes "CPU usage statistics" from which the distribution of the CPU resource can be determined. The usage statistics are all percentages; their sum should be 100% (+/- roundoff error). The CPU Usage Statistics are USED, IDLE, SWPW, SKED, TCOR, FILW, BGND, and possibly TRAP.

Each of the variables in the example are described below; they are taken line by line in left-to-right order.

USED: (%) -- Percentage of the interval during which the CPU was executing instructions on behalf of some user. This includes user, JSYS, page fault, and interrupt processing.

IDLE: (%) -- Percentage of the interval during which the CPU was idle because there were no active processes on the system. If this number is nonzero, the system can accommodate some additional usage. If the CPU is idle when there are active processes, its idle time is accounted for in SWPW or FILW, not under IDLE time.

SWPW: (%) -- Percentage of the interval during which the CPU was idle and while one or more processes was waiting on the completion of a memory management service (normally a swap in). For time to be accounted in this variable, all active processes must be in wait states.

SKED: (%) -- Percentage of the interval during which the system was scheduling users for memory and the CPU. Other system overhead functions are measured by the variables TCOR and BGND.

SUSE: (%) -- Sum of the runtime percentages accumulated for each job running at the time of the report. This value differs from USED only by the skew that builds up during the time that it takes WATCH to collect all of the data about each job and by the loss of data from jobs that logged out during the interval.

TCOR: (%) -- Percentage of the interval spent garbage collecting memory. This represents part of the memory management overhead. The garbage collection process requires the monitor to look at the age of each page in memory to determine the ones that have not been referenced in a while. The least recently used pages become the prime candidates for being swapped out.

FILW: (%) -- Percentage of the interval during which the CPU was

idle, no processes were waiting for memory management services, and at least one active process was waiting for a short-term, user-initiated event (disk I/O) to complete.

BGND: (%) -- Percentage of the interval during which the monitor was performing background tasks. The primary background task is moving terminal input characters from a system-wide buffer to the individual terminal input buffers. This variable also includes the CPU overhead to echo terminal input characters.

INTRP: (PS) -- Number of page fault traps per second. Not all page faults require disk input to be resolved. Some page faults are resolved with pages that are currently in memory but not assigned to the process generating the page fault. Some of these pages are found in the replaceable queue (see RPQS below) while others may be shared by other processes.

ICOR: (PM) -- The average number of memory garbage collections per minute performed by the monitor during the last interval.

UJBL: (PM) -- The average number of times per minute the system was forced to adjust the balance set during the last interval. A rate of 60 times per minute is normal on a system with several users and no IDLE time.

IREM: (PM) -- The average number of times per minute the monitor had to remove a process from the balance set before the process came to a natural wait state (such as terminal input wait). This number becomes nonzero whenever there are more jobs to be run than can fit simultaneously in memory. Whenever this situation occurs, the monitor removes processes from the balance set, swapping them out to make room for other runnable processes. In general, whenever this number goes nonzero, response time gets longer.

RAP: (%) -- Percentage of the interval during which the CPU was responding to page faults. This time is normally charged to the user, and is therefore also part of "USED" time. If the monitor was built to remove this "TRAP" time from "USED" time, it will become part of the system overhead (like SKED) and cannot be billed. In such a case, the TRAP time must be added to the "CPU Usage Statistics" in order to account for 100% of the CPU time.

RUN: (AV) -- The average number of processes that were simultaneously active during the interval. This number represents the CPU load on the system during the interval. When NRUN is greater than 1.0, the user programs experience an average execution time at least "NRUN" times longer than if the system were stand-alone.

BAL: (AV) -- The average number of processes in the balance set

during the interval. If this number is less than NRUN by more than .5, the implication is usually that there is not enough memory to hold all active processes.

NWSM: (AV) -- The average number of working sets in memory during the interval. If this number is significantly larger than NRUN or NBAL, working sets are not being forced out of memory when processes go into a wait state (like terminal input wait) and, consequently, response times should not be greatly affected by paging.

BSWT: (AV) -- The average number of processes in the balance set that are waiting for the completion of some event. Normally this number reflects the number of processes waiting for a page to be read in from the disk. If NBAL - BSWT is less than one, there are not enough runnable processes in memory to keep the CPU busy 100% of the time. The idle time is included in SWPW or FILW.

DSKR: (%) -- Percentage of the processes in balance set wait (BSWT) that are waiting for a file page to be read into memory.

DSKW: (%) -- Percentage of the processes in balance set wait (BSWT) that are waiting for file pages to be written back to the disk.

SWPR: (%) -- Percentage of the processes in balance set wait (BSWT) that are waiting for a page to be swapped into memory from the swapping area of the disk.

NLOD: (PM) -- The average number of working sets loaded per minute into memory.

CTXS: (PS) -- The average number of context switches performed per second by the scheduler. A context switch happens when the running process voluntarily blocks, or faults on a page that is not in memory, or when a higher priority process is ready to run. Since it takes CPU time to perform a context switch, CTXS directly affects SKED.

UPGS: (AV) -- The average number of pages assigned to processes with loaded working sets. These processes may or may not be in the balance set, but they are allocated memory.

FPGS: (AV) -- The average number of physical memory pages that are currently available for swapping in user processes. The monitor normally keeps between 20 and 100 free pages. The monitor uses these pages (and the rest of memory not in use by balance set processes) as a page cache. For example, if a process reenters the balance set after waking up from a blocked state and it still has some of its pages in memory in the free page pool, those pages are used directly without requiring any disk I/O. It has been demonstrated that this

cache plays an important part in overall system performance. Therefore, if FPGS is very small, the system performance has most likely been degraded.

DMRD: (PS) -- The number of reads per second made to the swapping area.

DMWR: (PS) -- The number of writes per second made to the swapping area.

DKRD: (PS) -- The number of reads per second made to the file system.

DKWR: (PS) -- The number of writes per second made to the file system.

DTIN: (PS) -- The number of terminal input characters received per second from all terminals (real and pseudo) on the system.

DTOU: (PS) -- The number of terminal characters output per second by all jobs on the system. This includes real and pseudo-terminals.

AKW: (PS) -- The number of process wake-ups per second. Some of the types of wake-ups that fall into this category are

IPCF	Process Termination
ENQ	DISMS
Terminal Input	TIMER
Terminal Output	IIC

TCC: (PS) -- The number of terminal interrupt characters (e.g., CTRL/C) typed per second.

DIO: (PS) -- The aggregate number of disk pages read or written per second to both the file system area and the swapping area. Normally, 60 pages per second for a one-channel and 100 pages per second for a two-channel system are saturation levels. This variable is the summation of DMRD, DMWR, DKRD and DKWR.

PQS: (PS) -- The average number of pages per second that were retrieved from the replaceable queue in order to satisfy page faults. These page faults do not require disk I/O.

CCW: (PS) -- The average number of pages per second that were freed by global garbage collections.

GCW: (PS) -- The average number of pages per second that were freed by local garbage collections on specific processes. These garbage collections remove those pages from a process' working set that have not been used in a long time.

KNOB: (value) -- This is the setting of the bias control knob. The twenty possible bias control knob settings can only represent six switch settings in Release 4. Low settings favor interactive jobs; high settings favor compute-bound jobs. The default setting is 11.

The QUEUE DISTRIBUTION PERCENTAGE represents the portion of USED time allocated to processes in the various scheduling queues. The first queue is only used by Job 0 and jobs in the special high priority category. Normally the percentage of runtime accumulated in this queue is small. The second and third queues are the interactive queues. If the sum of these two values is high, there is a high interactive load on the system. The last three queues are the computational queues. Processes move onto these queues only if they have entered a compute-bound phase. If the sum of these three values is high, the system load is primarily computational. When the class scheduler is turned on, interactive users are scheduled in queue order while processes in the lower three (computational) queues are given priority on the basis of their class' distance from its target share.

The system statistics data record (record type 01) has the following form:

Item	Length	Picture
1. Record Type	2	99
2. Record Sequence Number	3	999
3. Date & Time (MM/DD/YY HH:MM:SS)	17	X(17)
4. Interval	7	9(5).9
5. Number of Jobs	3	999
6. USED	7	9(4).99
7. IDLE	7	9(4).99
8. SWPW	7	9(4).99
9. SKED	7	9(4).99
10. SUSE	7	9(4).99
11. TCOR	7	9(4).99
12. FILW	7	9(4).99
13. BGND	7	9(4).99
14. NTRP	7	9(4).99
15. NCOR	7	9(4).99
16. AJBL	7	9(4).99
17. NREM	7	9(4).99
18. TRAP	7	9(4).99
19. NRUN	7	9(4).99
20. NBAL	7	9(4).99
21. NWSM	7	9(4).99
22. BSWT	7	9(4).99
23. DSKR	7	9(4).99
24. DSKW	7	9(4).99
25. SWPR	7	9(4).99
26. NLOD	7	9(4).99
27. CTXS	7	9(4).99

18. UPGS	7	9(4).99
19. FPGS	7	9(4).99
20. DMRD	7	9(4).99
21. DMWR	7	9(4).99
22. DKRD	7	9(4).99
23. DKWR	7	9(4).99
24. TTIN	7	9(4).99
25. TTOU	7	9(4).99
26. WAKE	7	9(4).99
27. TTCC	7	9(4).99
28. TDIO	7	9(4).99
29. RPQS	7	9(4).99
30. GCCW	7	9(4).99
31. XGCW	7	9(4).99
32. KNOB	5	9(5)
3. Queue Distribution %'s	70	
Max 10 entries; each entry		9(4).99
4. Number of Queue Dist Entries	2	99

Total 361 Characters

3 Load Averages -

The term "Load Average" refers to the average number of processes simultaneously demanding service over some interval of time. The following is an example of the load average display:

LOAD AVERAGES:	5.29	4.06	3.39
HIGH QUEUE AVERAGES:	3.76	2.86	2.25
LOW QUEUE AVERAGES:	1.54	1.20	1.14

CLASS LOAD AVERAGES

CLA	SHR	UTIL

0	80.00	96.20	4.70	3.08	2.94
1	15.00	3.80	0.23	0.19	0.15
2	5.00	0.00	0.00	0.00	0.00

LOAD AVERAGES--The system keeps three exponential load averages. These values represent the average load over the last 1 minute, the last 5 minutes, and the last 15 minutes. These numbers can be used to estimate the expected elongation of the elapsed time required to run a program. If the system load average equals X, the approximate elapsed time required to run an additional program on the system is at least $(1+X)*Y$, where Y is the alone elapsed time required to run this program.

HIGH QUEUE AVERAGES--These values represent the load of interactive jobs (queues 1 and 2).

LOW QUEUE AVERAGES--These values represent the load of compute-bound jobs (queues 3, 4, and 5). The sum of the high queue average and the low queue average equals the load average.

CLASS LOAD AVERAGES--This display is of interest when class scheduling is being utilized. The following information is presented for all classes defined for the system.

CLA -- The class number.

SHR -- The class share of the processor. This share corresponds to the percentage of the CPU that the monitor will try to distribute among the jobs in this class.

UTIL -- The actual utilization achieved by each class. The value here should be less than the share unless the class has received some windfall.

The 1-minute, 5-minute, and 15-minute load averages are displayed for each of the classes. These load averages may appear to be very large because they are computed as follows:

Processes in class making demands
Class Load Average = -----
Maximum (Share, Utilization)

Thus, if there are 5 processes making demands in a class with a share of .20 and a utilization of .15, that class' load average is 25 (5/.20).

If the class scheduler is not running, the utilization and load averages are all zero.

The following is the form of the load averages data record (record type 02):

Item	Length	Picture
1. Record Type	2	99
2. Record Sequence Number	3	999
3. Date & Time (MM/DD/YY HH:MM:SS)	17	X(17)
4. Interval	7	9(5).9
5. Number of Jobs	3	999
6. Load Averages (3 values) Each Value	21	9(4).99
7. High Queue Avgs (3 values) Each Value	21	9(4).99


```

8. Low Queue Avgs (3 values)  21
   Each Value                  9(4).99
9. Class Load Averages
   Max 32 entries;
   each entry as follows:
   a. Class Number             5      9(5)
   b. Share                    7      9(4).99
   c. Utilization              7      9(4).99
   d. Load Avgs (3 values) 21
       Each Value              9(4).99
10. Number of Class Load Avg.  2      99
      ----
      Total 1377 Characters

```

7.1.4.4 Directory Cache Statistics -

The directory cache statistics are only available when an enabled user responds "ALL" to the question "Print monitor statistics?". The following is a sample display:

```

Directory Cache hits: 175
Directory Cache Misses - Cache Full: 0
Directory Cache Misses - New Entry Added: 321

```

Directory Cache hits: -- The number of times an accessed directory was found in the cache.

Directory Cache Misses - Cache Full: -- The number of times an accessed directory was not found in the cache while all the cache slots were filled with active directories. In this case the most recently accessed directory cannot be put into the cache.

Directory Cache Misses - New Entry Added: -- The number of times the accessed directory was not found in the cache, though room was available to add it (possibly as a replacement for an inactive entry).

The "hit ratio" is computed by $HITS/(HITS+MISSES)$ and provides a good indication of the cache's effectiveness. For instance, the hit ratio in the example is $175/(0+321)$ or 35% (not very good).

The directory cache portion of data record type 05 is shown in the "Disk I/O Statistics" section below.

7.1.4.5 Normal Per-job Information -

The normal per-job information which is available to all users running WATCH and consists of a line for each job that had an active process during the interval. The following example shows the statistics reported for each job:

JOB	TTY	USER	PROGRAM	DELTA	RT	%RT	JU	CSH
0	DET	OPERATOR	SYSJOB	7.81	4.3	23.94	3.64	
1	43	OPERATOR	PTYCON	1.71	0.9	1.86	3.64	
5	47	OPERATOR	OPR	0.35	0.2	0.28	3.64	
10	3	KELLEY	BASIC	5.31	3.0	9.85	3.64	
13	12	DENNING	EXEC	8.46	4.7	7.17	3.64	
14	24	KOVALCIN	WATCH	3.80	2.1	3.92	10.00	
16	2	GRAVES	BASIC	3.72	2.1	9.18	3.64	
18	21	BLIZARD	SYSTAT	4.46	2.5	13.55	3.64	
19	14	BOYACK	EDIT	10.74	6.0	20.71	3.64	
20	20	WOLFE	EDIT	0.27	0.1	0.34	3.64	

JOB -- The job number assigned by the system when the user logged in.

TTY -- The number of the terminal that is being used by the user running this job. "DET" means detached.

USER -- The name of the directory that the user logged into.

PROGRAM -- The name of the program being run or the name of the EXEC command being used. Please note that the program name is obtained at the time the sample is taken. It is not possible to tell if the program or command was running during the entire interval.

DELTA RT -- The incremental amount of CPU time (in seconds) that the job used during the interval.

%RT -- The percentage of the interval represented by the DELTA RT. The sum of all %RT values is used to compute the SUSE.

JU -- Job Utilization. When the class scheduler is running this will normally be a nonzero value. It represents the CPU utilization accumulated by this job and charged to the job's class share. Because the class scheduler tries to divide the class share equally among all active users in the class, computational jobs within the same class should normally receive nearly the same job utilization.

CSH -- Class Share. When the class scheduler is running, this value reflects the class' share divided by the number of active jobs in that class -- this is the target share for the job. Normally the job utilization (JU) is less than a job's class share.

There is one normal per-job data record (record type 04) for each job. The sequence numbers begin with "001" and increment by "001". The form of a record is

Item	Length	Picture
1. Record Type	2	99
2. Record Sequence Number	3	999
3. Date & Time (MM/DD/YY HH:MM:SS)	17	X(17)
4. Interval	7	9(5).9
5. Number of Jobs	3	999
6. Job Number	3	999
7. TTY Number (777 = DET)	3	999
8. User Name	20	X(20)
9. Program Name	6	X(6)
0. DELTA RT	7	9(4).99
1. %RT	4	99.9
2. JU	7	9(4).99
3. CSH	7	9(4).99
	--	
Total	89	Characters

1.4.6 Expanded Per-job Information -

Most of the information presented in this display is obtained by setting breakpoints in the monitor with the SNOOP JSYS. Thus, this information is only available to users who are running WATCH with either WHEEL or OPERATOR privileges enabled.

The presented information occupies a full 132-character line. For purposes of explanation, the columns are broken up as follows:

Job Identification Information:

OB TTY USER PROGRAM

These variables are the same as those listed in the normal per-job display.

Job Utilization Information:

RT DEMD USED GRDY BRDY SWPTR DSKR DSKW RPQW OTHR

Memory, Response, and Disk Information:

MF NLD NRSP RESP SP WSS UPGS SWPR DSKR TPF IFA

The job utilization information and memory, response, and disk

information are discussed in the next two sections below.

JOB UTILIZATION INFORMATION

The following is an example of the job utilization portion of the extended per-job display:

JOB		%RT	DEMD	USED	GRDY	BRDY	SWPR	DSKR	DSKW	RPQW	OTHR
0		1.0	14.3	8.6		72.9	5.7	4.0	8.8		
2		0.1	0.7	18.0		82.0					
4		0.6	3.6	19.3		34.8	38.4	5.8	1.8		
7		0.0	0.3	15.5		19.3	65.2				
9		0.5	1.2	43.0		15.7		35.6	5.7		
10		9.7	49.8	20.7		63.5	2.9	11.9	0.9		0.5
12		0.1	0.4	29.4		70.6					
15		1.0	6.0	19.4		20.2	1.1	16.0	43.3		
16		22.4	38.7	61.5		38.5					
19		1.1	5.7	27.7		72.3					

%RT (%) -- The percentage of the interval during which this job actually received CPU time.

DEMD (%) -- Summation of the percentages of the interval that each process in the job was active. If only one process in the job is active during the interval (the normal case), DEMD is less than or equal to 100%. If more than one process are simultaneously active, DEMD could exceed 100% (for a job of n processes, DEMD could be up to n*100%).

The rest of the variables in this display indicate what a job was doing during its "active" period. These statistics are all expressed as percentages of DEMD and thus their sum is 100%. When assessing the importance of the statistics for a specific job, you should multiply these percentages by DEMD to get the percentage of the interval time.

USED (%) -- The percentage of DEMD that the processes in this job spent using the CPU.

GRDY (%) -- The percentage of DEMD that processes in this job were runnable but could not fit in the balance set. The most common cause for processes to be on this list is that there is not enough memory to hold all runnable jobs.

BRDY (%) -- The percentage of DEMD that processes in this job were in the balance set but were not being run. Usually processes in this state are waiting for their turn to use the CPU.

SWPR (%) -- The percentage of DEMD that processes in this job waited for page faults from the swapping area to be satisfied.

DS (%) -- The percentage of DEMD that processes in this job waited for file pages to read in from the disk.

DSKW (%) -- The percentage of DEMD that processes in this job waited for file pages to be written to the disk.

RPQW (%) -- The percentage of DEMD that processes in this job waited for a physical memory page to become available for swapping into. Usually, when time is accumulating here, there is a shortage of memory on the system.

OTHR (%) -- The percentage of DEMD that processes in this job spent in any of the other wait states.

1.1.4.7 Memory, Response, And Disk Information -

The following is an example of the memory, response, and disk information portion of an extended per-job display:

JOB....	IMEM	NLD	NRSP	RESP	SR	WSS	UPGS	SWPR	DSKR	TPF	IFA
0	917.1	1	165	0.06	6	410.0	7.7	20	17	45	40
	99.9	0	12	0.07	6	8.0	7.0	0	0		
4	192.5	3	28	0.11	6	37.0	7.2	29	4	57	25
7	78.2	0	1	0.39	6	27.0	25.0	8	0	32	7
9	99.9	0	12	0.12	2	56.0	54.8	0	9	57	69
10	198.1	4	95	0.12	5	124.0	54.2	32	109	62	87
12	199.8	0	17	0.03	3	18.0	7.6	0	0		
15	99.9	0	12	0.61	5	27.0	17.2	3	24	45	51
16	99.9	0	357	0.13	2	22.0	20.6	0	0		
19	99.9	0	329	0.02	4	15.0	13.1	0	0		

MEM (%) -- The percentage of the time that the working sets of a job's processes are in memory. This number is the summation of the percentages for each process in the job and thus may exceed 100%.

LD (CNT) -- The number of times working sets for processes in this job were loaded into memory. If this number is zero, no working sets were loaded during the interval (i.e., the working sets were in memory for the whole interval).

RSP (CNT) -- The number of responses that a job had during the interval. A response is counted whenever a process wakes up for one of the reasons specified under WAKE:.

ESP (AV) -- The average response time in seconds during the interval. Responses that require more than two seconds of CPU time to finish are not counted in this column.

SR () -- The "stretch ratio" for each response (as represented in RESP). The stretch ratio is obtained by dividing the elapsed time of each response by the compute time required to satisfy it. ($SR = \text{elapsed time} / \text{CPU time}$). The only responses counted are those that require less than two seconds of CPU time to complete. Thus, the stretch ratio is the elongation perceived by the interactive user.

WSS (SUM) -- The sum of the maximum working set sizes of all active processes in the job.

UPGS (AV) -- The average number of pages actually in memory when a process from a job is in the balance set.

SWPR (CNT) -- The number of times a process in a job waited for a faulted page to be read in from the swapping area. This does not include pages that were preloaded by the working set manager.

DSKR (CNT) -- The number of times a process waited for a disk read to complete. Because many programs prefault pages, this count will be different from the actual number of pages read.

TPF (AV) -- The average number of milliseconds that it took to satisfy each page fault for this job during the interval.

IFA (AV) -- The "inter-fault average". This value represents the average compute time in milliseconds between page faults for job. A large "IFA" means that the working sets of processes in this job are very stable.

There is one expanded per-job data record (record type 03) for each job. The sequence numbers begin with "001" and increment by "001". The form of the record is

Item	Length	Picture
1. Record Type	2	99
2. Record Sequence Number	3	999
3. Date & Time (MM/DD/YY HH:MM:SS)	17	X(17)
4. Interval	7	9(5).9
5. Number of Jobs	3	999
6. Job Number	3	9(3)
7. TTY Number	3	9(3)
8. User Name	20	X(20)
9. Program Name	6	X(6)
10. %RT	4	99.9
11. DEMD	6	9(4).9
12. USED	5	999.9
13. GRDY	5	999.9
14. BRDY	5	999.9
15. SWPR	5	999.9

6. DSKR	5	999.9
7. DSKW	5	999.9
8. RPQW	5	999.9
9. OTHR	5	999.9
10. IMEM	6	9(4).9
11. NLD	4	9(4)
12. NRSP	5	9(5)
13. RESP	6	999.99
14. SR	3	999
15. WSS	7	9(5).9
16. UPGS	7	9(5).9
17. SWPR	5	9(5)
18. DSKR	5	9(5)
19. TPF	4	9999
20. IFA	4	9999

Total 170 Characters

.1.4.8 System Utilization Statistics -

Two sections comprise the system utilization statistics. The first consists of summaries for expanded per-job statistics, including summaries for the job utilization information and the memory, response and disk information. The second consists of additional system variables and several computations.

.1.4.9 System Summary Of Per-job Variables -

These statistics are listed under the per-job statistics on the line that begins "System Summary". The following is an example of such a summary:

DEMD	USED	GRDY	BRDY	SWPR	DSKR	DSKW	RPQW	OTHR
339.2	28.6		51.9	5.0	11.5	3.0		

IMEM	NLD	NRSP	RESP	SR	WSS	UPGS	SWPR	DSKR	TPF	IFA
102.9	37	1772	0.13	3	2168.	231.8	370	911	52	90

DEMD (%) -- The sum of each item in the DEMD column. This represents the total demand put on the system over the interval.

SET GRDY BRDY SWPR DSKR DSKW RPQW OTHR (%) -- These values represent the average percentage of the DEMD time that the jobs were in these states.

IMEM (SUM) -- The summation of all IMEM per-job values. This sum is significant as an indicator of how many working sets belonging to processes active during the interval were simultaneously in memory. For instance, the value 4102.9% indicates that approximately 41 working sets belonging to active processes were simultaneously in memory. This number can be compared with NWSM%.

NLD (SUM) -- This is the number of working sets loaded during the interval. It should correspond to the rate given by the variable NLOD%.

NRSP (SUM) -- The number of responses counted for all jobs during the interval.

RESP (AV) -- The average response time for those responses measured (requiring less than two seconds of CPU time) during the interval.

SR (AV) -- The average stretch ratio for all interactions requiring less than two seconds of CPU time.

WSS -- The arithmetic sum of the WSS values for all jobs. This represents the maximum amount of memory that would have been required during the interval if all active processes achieved their largest size at the same time and were all in memory.

UPGS (AV) -- The average number of pages needed by the active processes at any specified point in time.

SWPR (SUM) -- The total number of swap reads done by jobs on the system in response to page faults. This does not include pages preloaded by the working set manager.

DSKR (SUM) -- The total number of disk pages read that caused processes to wait.

TPF (AV) -- The average time required to wait for a page fault (swap or disk) to be resolved.

IFA (AV) -- The average amount of compute time a job spends between page faults.

Additional System Variables and Computations

The information in this display includes additional system statistics not available in the system statistics display and computations of various other variables. The following is an example:

TOTRC:	1992	LOKPGS:	104	SHR PGS:	245	AVAIL MEM:	1888
NRUN MIN,MAX:			1		11		
SUMNR MIN,MAX:			1879		2073		
NRPLQ MIN,MAX:			28		170		

SY MEM DMD =	255.2
SWAP RATIO (SUM WSS / AV MEM) =	1.15
ACTIVE SWAP RATIO (DMD/AVMEM) =	0.14
MEM UTILIZATION ((UPGS+SHRPGS)/AVMEM) =	0.25
AV WS SIZE =	28.84
AV CPU TIME (MS) PER INTERACTION =	40.57
THINK TIME (SEC) PER INTERACTION =	1.43

TOTRC -- The number of physical memory pages available. This is the total physical memory minus the number of pages required by the resident monitor.

LOKPGS -- The current number of pages locked down by the monitor beyond the resident monitor pages. Out of this set of pages comes the terminal buffers, magtape buffers, line printer buffers, and other pages locked down during certain file system operations.

SHR PGS -- The number of pages of physical memory being shared by more than one process at the end of the interval. This is included in the count "AVAIL MEM".

AVAIL MEM -- The difference between "TOTRC" and "LOKPGS". This is the actual number of pages available for use by user programs.

IR MIN, MAX -- The minimum and maximum number of simultaneously active processes during the interval.

SUMNR MIN, MAX -- The minimum and maximum number of pages belonging to working sets in memory during the interval.

IRPLO MIN, MAX: -- The minimum and maximum number of pages on the replaceable queue during the interval.

SYS MEM DMD -- The system average memory demand derived by computing the integrals of the memory forecast for each process during its active period, summing over all processes and dividing by the interval time. Whereas the "system summary UPGS" is the average amount of memory actually in use at any point in time, this value is the average amount forecast at any point in time.

WAP RATIO (SUM WSS / AV MEM) -- The swap ratio is the system WSS divided by the amount of available main memory. If this is greater than 1, it represents the amount by which main memory would have to be increased to avoid any swapping.

CTIVE SWAP RATIO (DMD/AVMEM) -- The active swap ratio is the system average memory demand divided by the amount of available main memory. If this number is greater than 1, it represents the amount by which main memory would have to be increased to hold all jobs wanting to run simultaneously.

MEM UTILIZATION ((UPGS+SHRPGS)/AVMEM) -- The memory utilization is the number of system-used pages divided by the amount of available main memory. For active swap ratios greater than 1, this indicates how well the monitor is doing in keeping memory used.

AV WS SIZE - The average working set size is calculated from the integrals computed from the working set demands over the active period of each process divided by the sum of the active periods of each process.

AV CPU TIME (MS) PER INTERACTION -- The average amount of CPU time a job spends between each response.

THINK TIME (SEC) PER INTERACTION -- The average time spent by the user between the time the system requests a response and when that response is received.

The system utilization portion of data record type 05 is shown under the "Disk I/O Statistics" section below.

7.1.4.10 Disk I/O Statistics -

The following is an example of the disk I/O statistics display:

DISK I/O

CHN,UNIT	SEEKS	READS	WRITES	
0,6	380	485	300	PS #1
0,7				REL4 #0
1,0	49	185	34	SNARK #0
1,1	2	6		LANG #0
1,2				
2,3	57	68	32	MISC #0
2,4				
2,5	602	652	449	PS #0

These statistics display the following information:

CHN,UNIT -- The channel number and the unit number on the channel to which the disk is connected.

SEEKS -- The number of times the disk heads had to be moved to get to the next request during the interval. If multiple requests can be answered on the same cylinder, no seek will take place.

READS -- The number of pages read on this unit during the

interval.

WRITES -- The number of pages written on this unit during the interval.

The last column shows the name of the structure and its relative unit number within the structure.

Data record type 05 contains the information from three displays: directory cache, system utilization, and disk I/O statistics. The following shows the format of these records:

Item	Length	Picture
1. Record Type	2	99
2. Record Sequence Number	3	999
3. Date & Time (MM/DD/YY HH:MM:SS)	17	X(17)
4. Interval	7	9(5).9
5. Number of Jobs	3	999
6. Dir Cache Hits	6	9(6)
7. Dir Cache Misses - Full	6	9(6)
8. Dir Cache Misses - New Entry	6	9(6)
9. TOTRC	5	9(5)
0. LOKPGS	5	9(5)
1. SHR PGS	5	9(5)
2. VAIL MEM	5	9(5)
3. NRUN MIN	5	9(5)
4. NRUN MAX	5	9(5)
5. SUMNR MIN	5	9(5)
6. SUMNR MAX	5	9(5)
7. NRPLQ MIN	5	9(5)
8. NRPLQ MAX	5	9(5)
9. SYS MEM DMD	7	9(5).9
0. SWAP RATIO	8	9(5).99
1. ACTIVE SWAP RATIO	8	9(5).99
2. MEM UTILIZATION	8	9(5).99
3. AV WS SIZE	8	9(5).99
4. AV CPU TIME	8	9(5).99
5. THINK TIME	8	9(5).99
6. Disk I/O		
Max 15 entries;		
each entry as follows		
a. Channel	2	99
b. Unit	2	99
c. Seeks	6	9(6)
d. Reads	6	9(6)
e. Writes	6	9(6)
f. Name	10	X(10)
g. Number	2	99
h. Number of Disk I/O entries	2	99

Total 667 Characters

7.1.4.11 Tune Mode Statistics -

Tune Mode is designed to display on one line some of the more interesting statistics so that a system programmer can easily monitor the changes in load during a test period. This mode is useful when a very short interval is desired (around 10 seconds). The information is abstracted from the system statistics and system utilization statistics displays and includes the following variables:

USED	SWPW	SKED	CTXS	WAKE	TDIO	NRUN	NWSM	NLOD	USED
52.8	2.4	6.8	41.2	16.6	13.1	2.2	62.5	6.87	34.8

GRDY	BRDY	SWPR	DSKR	DSKW	RPQW	OTHR
	41.7	2.3	19.5	1.8		

IMEM	NLD	NRSP	RESP	SR
2207.9	0	198	0.05	2

The definition of the variables on the first row can be obtained from the system statistics display and the definitions on the second and third rows from the system utilization statistics display.

The form of the tune mode data record (record type 06) is as follows:

Item	Length	Picture
1. Record Type	2	99
2. Record Sequence Number	3	999
3. Date & Time (MM/DD/YY HH:MM:SS)	17	X(17)
4. Interval	7	9(5).9
5. Number of Jobs	3	999
6. USED	7	9(4).99
7. SWPW	7	9(4).99
8. SKED	7	9(4).99
9. CTXS	7	9(4).99
10. WAKE	7	9(4).99
11. TDIO	7	9(4).99
12. NRUN	7	9(4).99
13. NWSM	7	9(4).99
14. NLOD	7	9(4).99
15. USED	5	9(3).9
16. GRDY	5	9(3).9
17. BRDY	5	9(3).9
18. SWPR	5	9(3).9
19. DSKR	5	9(3).9
20. DSKW	5	9(3).9
21. RPQW	5	9(3).9

22. OTHR	5	9(3).9
23. IMEM	6	9(4).9
24. NLD	4	9(4)
25. NRSP	5	9(5)
26. RESP	6	9(3).99
27. SR	3	999

Total 159 Characters

CHAPTER 8

WORKING SET SWAPPING

8.1 WORKING SET SWAPPING

With TOPS-20, the working set of a process can be brought in with the overhead pages (preloading) and is removed upon being elected to be removed (postpurging). Postpurging of process working sets always occurs. Preloading is not done by default, but is a parameter that can be set by the administrator. If preloading is not done, the process page faults its pages in much the same way as it did previous to Release 4.

8.1.1 WORKING SET DATA BASE

The information about the fork's working set is kept in a 28-word table in the PSB. This table is the working set cache. Each word contains four, nine-bit entries for a total of 512 entries. An entry appears as follows:

! V ! U ! Section ! Page !

here:

- V
indicates whether the entry is valid.
- U
indicates whether the entry is a user page or a monitor page.
- Section
indicates which section this page is in.

Page is the higher-order two bits of the address of this page.

The lower-order seven bits of the page number are used as an index into the table. There can be conflict since pages with the same lower-order seven bits will have the same index. For this reason there are four entries at each index. If the number of conflicts exceeds four, the word is shifted left one entry. The leftmost entry is lost and the new entry is added on the right. This way the four most recent entries are "remembered."

Much of the information pertinent to the working set swapping of a particular fork is contained in the table FKSHP. This table, indexed by fork, contains a flag that indicates whether the fork's working set is in memory as well as whether other types of information are used to determine a fork's eligibility to be swapped in or out.

```

FKSWP: +-----+
      !  flags  !  FKGOLN  !  FKHST  !
      +-----+
      !         !         !         !
      !         !         !         !
      !         !         !         !
      !         !         !         !
      !         !         !         !
      !         !         !         !
      +-----+

```

```

      0 1 2 3 4 5 6 7 8      17 18      35
+-----+
! ! ! ! ! ! ! ! FKGOLN ! FKHST !
+-----+

```

Symbol	Bits	Contents
FKWSL	0	Working set loaded -- on if the working set of the fork is loaded
FKBLK	1	Fork blocked -- on if fork is blocked and on the wait list; off if the fork is on the GOLST
FKIBS	2	Fork in balance set -- on if fork is in balance set (note that it is not necessarily in memory)
BSWTB	3	Balance set wait -- on if fork is in balance set (short-term, i.e. page fault) wait
BSNSK	4	NOSKED -- on if fork is NOSKED and not actually running
BSCRSK	5	Critical section -- on if fork is CRSKED and not actually running
FKIBH	6	In balance set hold -- on if fork entered balance set since last update to history
FKBSHF	7	In balance set hold -- local to AJBALS

FKGOLN 8-17 GOLST position
FKHST 18-35 Fork history -- value based on the recent
history of the fork, particularly whether
fork has been in the balance set or not

8.1.2 IMPLEMENTATION

Two areas of interest in the implementation of working set swapping are 1) the process of deciding which processes should be swapped in or out, and 2) the mechanics of maintaining a working set.

8.1.2.1 Swapping In/Out -

The working set manager must maintain enough available memory for processes to run, but it is desirable to keep in memory poorly behaved processes that block and unblock frequently to minimize swapping. To accomplish these goals, a fork's history is maintained.

The history (FKHST) of a fork is calculated periodically (currently twice a second) in the routine STEPFH using the following algorithm:

$$\text{FKHST (new)} = \frac{\text{FKHST (old)} * (\text{STEP} - 1) [+400000 \text{ (octal) if FKIBH}] + \text{STEP}}{\text{STEP}}$$

where:

FKHST (new) is the new history value of the process

FKHST (old) is the previous history value of the process

STEP is a time constant used to control the rate of change; currently set to 8

400000 (octal) is added if FKIBH is set, which indicates that the process was in the balance set during the previous interval

The larger the history value, the more recently and to a greater extent a process was in the balance set. This favors keeping the process' working set in memory.

To determine which working sets to keep in memory, which working sets to remove from memory, and which working sets to bring into memory, the working set manager calculates a value for each process. This value is calculated by tallying points of a process based on its characteristics. If a process has a specific characteristic, it gets the points indicated by that characteristic. The value's calculation is based on the information contained in the following table:

Item Checked	Characteristic	Value Added	Symbol For Value
PIBMP	Priority interrupt bump	40,,0	FHMBIP
BSNSK	NOSKED	4000,,0	FHMNSK
BSCRSK	Critical section	100,,0	FHMCSK
FKQN	On priority queue	400,,0	FHMPQ
FKIBS	In balance set	200,,0	FHMBS

If not runnable (based on the value FKBLK) add FKHST
(history).

If runnable (FKBLK again) add FKGOLN (GOLST position).

If memory is overloaded, remove the working set of the process in memory with the smallest value.

If the largest value for a process not in memory is greater than the smallest value for a process in memory, try to bring the largest process not in memory into memory.

8.1.2.2 Mechanics Of Maintaining The Working Set -

The following operations must be able to occur:

1. Working set swapped out:

The swapping-out is handled in a straightforward manner; the working set is selected and most of the mechanics is handled using the same code as XGC, the local garbage collection.

2. Working set swapped in:

Swapping-in is handled by first getting the overhead pages for the process (PSP, UPT, and JSB), if necessary. Then, the working set is swapped in based on the working

set table (that was being used the last time the process was running).

3. Working set table updated as pages are used:

Maintenance of the working set table occurs in two major areas. When a process page faults, the page is added to the working set table by the page fault code. XGC (local garbage collection) now scans the working set cache to decide which pages to remove.

CHAPTER 9

SYSTEM DEBUGGING

9.1 INTRODUCTION

This document contains information pertinent to debugging. Some of it is relevant mostly to crash analysis. The last section contains information about MDDT, EDDT, and FILDDT.

Analyzing a crash is different than debugging a crash. Analysis tells you what happened and debugging tells you why (and thus, how to fix it). The information here can help you analyze what happened.

Successful crash analysis depends heavily on how well you know the data base and whether you can discover inconsistencies that give you clues about what happened. There are some basic things that help you look at any crash. The information in this document shows you how to use available tools to look at a crash and how to find basic information about the state of the machine at the time of the crash. Further analysis of a crash requires that, based on the state of the data base, you are able to propose a reason for what happened that matches the state of the data base.

9.2 CTY OUTPUT

Collect any CTY output that is relevant to the crash. This should include the KLERR printout and the BUGHLT (as well as recent BUGCHKs and BUGINFs), if any. If the machine got a BEP-ALIVE CEASED, then the KLERR output is the only reliable information you get (in release 3A). See the section on the BUGHLT location for an explanation of why this is true.

9.2.1 Explanation Of KLERR Output

KLERR includes the PC, the last memory fetch and information on the PI system. The PI information includes the following:

PI STATE: ON or OFF	--indicates whether the PI system is on or not.
PI ON: n	-- n indicates which of the 7 channels are enabled.
PI HLD: n	-- n indicates which of the 7 channels have an interrupt in progress.
PI GEN: n	-- n indicates which of the 7 channels have a pending interrupt.

Here is an example of the KLERR output on a KEEP-ALIVE CEASED error:

%DECSYSTEM-20 NOT RUNNING

```
KEEP ALIVE CEASED
KLERR -- VERSION V02-02 RUNNING
KLERR -- KL NOT IN HALT LOOP
KLERR -- KL ERROR OTHER THAN CLOCK ERROR STOP
KLERR -- KL VMA: 000000 035717      PC: 000000 035717
KLERR -- PI STATE: ON, PI ON: 177, PI HLD: 004, PI GEN: 001
KLERR -- EXIT FROM KLERR
```

9.3 GETTING A DUMP

DUMP.EXE is a pre-allocated file into which BOOT writes the dump. When the system comes up, SETSPD copies DUMP.EXE to DUMP.CPY.

9.3.1 How To Get A Dump

If the system does an auto-reload, the console front end will give BOOT the commands to get a dump. If the auto-reload doesn't work for some reason, you can force a dump as follows:

KLI -- VERSION VB06-07 RUNNING

```

KLI -- ALL CACHES ENABLED
KLI -- BOOTSTRAP LOADED AND STARTED
?DUPL STR UNI?DUPL STR UNI;problem because two PS:
                                ;structures on line,
BOOT>/d                        ;request a dump (after the
                                ;problem is corrected).

BOOT>                          ;type CR for default monitor
```

1.3.2 Where BOOT Lands

The console front end loads BOOT into KL memory. Of course, this overwrites whatever used to be in that part of memory. Therefore, BOOT is always loaded into a part of the monitor that contains pure code (i.e., so no data is destroyed); currently BOOT is brought in on top of part of APRSRV. If you need to look at code that is loaded where BOOT lands, you have to go to the listings. BOOT also uses some of high core to build the EXE directory for the file DUMP.EXE.

2. SYSERR

2.4.1 Overview Of SYSERR Functions And Data Base

YSERR is a program that reads PS:<SYSTEM>ERROR.SYS and generates reports on hardware errors, system crashes, front end reloads, etc. Entries in the file ERROR.SYS are written via the SYERR SYS. For a description of each of the types of entries, see the monitor tables.

When an ERROR.SYS entry is desired, the caller (which is one of the system programs such as QUASAR or the monitor itself) builds a SYSERR block as described in the monitor tables and does the YERR JSYS. The SYERR JSYS adds the block to a queue in the monitor's address space; the queue header is SEBQOU (location 4). Then it wakes up the Job 0 task which processes the queue and writes the queued entries to ERROR.SYS.

2.4.2 Queued SYSERR Blocks In A Crash

BUGHLT entry is generated when the system BUGHLTs; this entry is queued but not written to ERROR.SYS. The system is considered to be in an unsafe state at the time of the crash. When the system comes back up, code in SETSPD is called to move any queued SYSERR blocks in the dump to ERROR.SYS.

The queue header is location 24 (called SEBQOU); each SYSERR block consists of the standard SYSERR header followed by the information in the specific block type.

Sometimes it is useful to look at the queued SYSERR blocks in a crash, particularly the BUGHLT block. The BUGHLT block contains certain status information at the time of the crash. The status words are described below:

1. CONI APR,

Read the status of the processor error and sweep flags. This information is stored in offset BG%APS of the BUGHLT block. The flags and status information returned by a CONI APR, are described on page 3-59 of the Hardware Reference Manual.

2. CONI PAG,

This information is stored in offset BG%PGS of the BUGHLT block. A CONI PAG reads the system status of the pager. If TOPS-20 pagins is on, there is a 1 in bit 21. Bits 23-35 contain the contents of the EBR (the address of the EPT). For a description of all the fields, see page 3-41 of the Hardware Reference Manual.

3. DATAI PAG,

This information is stored in offset BG%PGD of the BUGHLT block. A DATAI PAG returns the process status of the pager. DATAI PAG, returns the current and previous context AC blocks, and the address of the UPT. For a complete description of the fields returned by a DATAI PAG, see page 3-42 of the Hardware Reference Manual.

4. CONI PI,

This information is stored in offset BG%PIS of the BUGHLT block. A CONI PI returns the status of the priority interrupt system; it indicates which levels are on, whether the PI system is on, and on which levels interrupts are currently being held. For a complete description, see page 3-7 of the Hardware Reference Manual.

9.4.3 Moving SYSERR Blocks From A Crash To ERROR.SYS

As stated before, SETSPD moves queued SYSERR blocks from the crash to ERROR.SYS. A Job 0 task starts the SETSPD program at START3; this code copies DUMP.EXE to DUMP.CPY and then issues a SYERR JSYS for each queued SYSERR block in the crash.

.5 BUGHLT

ocation BUGHLT contains the location the BUGHLT came from. That location contains an XCT BUGHLT-name. All BUGHLT code is generated by the BUG macro which is defined in PROLOG.

.5.1 BUG Macro

```

EFINE BUG(TYP,TAG,STR,REGS,%NAM,%STR)<
    XCT [TAG::      JSR BUG'TYP
        IRP REGS,<
            Z REGS>
            SIXBIT /TAG/]
            .PSECT BGSTR
STR:      ASCIZ \STR\
            .ENDPS BGSTR
            .PSECT BGPTR
            XWD TAG,%STR
            .ENDPS BGPTR

```

his is an example of a call to the BUG macro:

UGHLT,JONRUN,<JOB 0 NOT RUN FOR TOO LONG, PROBABLE SWAPPING HANGUP
f a JONRUN BUGHLT occurred, then the data base would look like
his:

```

BUGHLT/ CAIA CLK2+6      ;address the BUGHLT came from
CLK2+6/ XCT JONRUN      ;generated by BUG macro
JONRUN/ JSR BUGHLT
        / SIXBIT \JONRUN\

```

.5.2 BUGHLT Contents

ocation BUGHLT is set up with the location the BUGHLT came from
f the machine BUGHLT'd. That location contains an XCT
UGHLT-name. All the BUGHLT's are listed in the OPERATOR's GUIDE
ith a short descriptive phrase.

f there is a zero in location BUGHLT the machine probably got a
EEP ALIVE CEASED. This happens if you get either a "clock error
tor-" or "deposit/examine failure". Both of these errors are
ai—are failures and Field Service should be called. Although
hese two errors are the most likely cause of keep alive ceased,
ou cannot rule out the possibility of a software bug. Get the

KLERR output from the CTY. It will have the PC and PI state.

Note that because the console front end just does a reload for a KEEP ALIVE CEASED, the information in the dump is not dependable. This is because the cache has not been written out, the AC's have not been saved, etc.; these things are normally done by the BUGHLT code. The only valid information is the CTY output from KLERR.

For more information on release 4 changes, refer to the chapter "Bug'Typ Macro Changes for Version 4 of TOPS-20".

9.5.3 Creating Your Own BUGHLT

If you should desire the machine to stop in a certain state and save the information in a dump to allow later examination, you may want to create your own BUGHLT or BUGCHK. To patch a BUGHLT into the monitor, follow these steps:

In the patch area, create the BUGHLT information that is generated by the BUGHLT macro. Its format is:

```
BUGHLT name/ JSR BUGHLT
      / .                ;addresses of any locations
                        ;whose contents are to be
                        ;dumped (left half must be
                        ;zero)
      /bughlt name in Sixbit
```

Redefine the patch area to begin after the BUGHLT information you just created.

Use the DDT patch facility to insert the code to check for the condition you wish to cause a BUGHLT.

9.6 PUSH DOWN LISTS AND RELATED DATA BASES.

The push down list that is currently in use implies generally what was going on. For example, if the scheduler was running, SKDPDL is the push down list. If a page fault was in progress, TRAPSK is the push down list. Also, the former P is saved.

9.6.1 How To Look At A Stack

1. P contains the current stack pointer.
2. If an entry was made on the stack by a PUSHJ, then it will look like a PC. This isn't a hard and fast rule, but it can help. A user mode PC ususally has bits 1,2,

and 3 on and a monitor PC has bits 1 and 2 on.

3. If a return address is still on the stack (i.e., has an address less than the stack pointer), then you haven't returned from the routine.
4. The monitor uses the stack for temporary storage. The macros STKVAR, TRVAR, etc. leave recognizable things on the stack. Knowing these conventions helps you to recognize which stack locations are being used as temporary storage.

.6.2 Push Down List/ Machine State

The monitor uses different stacks to do different things. Register P is the current stack pointer and indicates which stack is in use at the time of the stack. The stacks and their uses are listed below:

1. UPDL -- Used when running in exec mode for the user, i.e., doing a JSYS. Also used by the Job 0 tasks that run in exec mode.
2. TRAPSK -- Used for page fault handling.
3. PIPDB -- Used for software interrupt handling.
4. SKDPDL -- Used by the scheduler for the overhead cycle.
5. DTESTK -- DTE interrupt level stack (PI level 6).
6. PHYPDL -- Used by PHYSIO when queueing and IORB.
7. PHYIPD -- Used when PHYSIO is handling an interrupt.
8. MEMPP -- Used when handling APR interrupts.

.6.3 Stack Usage For Local Storage

There are several macros that provide local storage; they use the stack. What they put on the stack is usually recognizable. See MACSYM.MEM for further information.

1. STKVAR

STKVAR uses the stack as temporary storage; the local variables have names that are really stack locations. STKVAR uses n stack locations for local variables where

n is the number of local variables requested, a count of local variables, and the return address .STKRT. On the stack you will see:

```
    / local variable
    / local variable
    / .
    / .
    / local variable n
    / n,,n           ;count of local variables
                      ;(used to adjust the stack
    / .STKRT         ;routine to clean up the
                      ;stack and return
```

Therefore, when you find .STKRT on the stack, the word before it is the count of local variables and tells you how many locations on the stack are in use by STKVAR.

2. TRVAR

TRVAR uses the stack much the same way as STKVAR but it also uses AC15; this is pushed on the stack first. The stack locations it uses look like this:

```
    / AC15
    / local variable
    / local variable
    / .
    / .
    / local variable n
    / n,,n
    / .TRRET
```

Therefore, when you find .TRRET on the stack, the word before it is the count of local variables and register 16 is stored on the stack in front of the local variables.

3. ASUBR

ASUBR saves AC15, AC's 1-4, followed by the return address .ASRET, which is a routine to clean up the stack. When you see the address .ASRET on the stack you can expect the following in this part of the stack:

```
    / AC15
    / AC1
    / AC2
    / AC3
    / AC4
    / .ASRET
```

4. ACVAR

ACVAR can save AC5, AC5 and AC6, AC5-AC7, AC5-AC10, or AC5-AC14, depending on the arguments given. In each case, the return address to clean up the stack is the last item pushed on the stack by the ACVAR macro; the return address stored on the stack is the clue to what else was pushed on the stack. Each of the possible cases is listed below:

1. AC5 saved

```
      / AC5  
      / .SAV1+2      ;return address
```

2. AC5 and AC6 saved

```
      / AC5  
      / AC6  
      / .SAV2+3      ;return address
```

3. AC5, AC6, and AC7 saved

```
      / AC5  
      / AC6  
      / AC7  
      / .SAV3+4      ;return address
```

4. AC5, AC6, and AC7 saved

```
      / AC5  
      / AC6  
      / AC7  
      / AC10  
      / .SAV4+5      ;return address
```

5. AC5 through AC14 saved

```
      / AC5  
      / AC6  
      / AC7  
      / AC10  
      / AC11  
      / AC12  
      / AC13  
      / AC14  
      / .SAV8+7      ;return address
```

5. SAVEAC

SAVEAC takes a list of AC's to be saved as an argument. It pushes the list of AC's on the stack, followed by the address of a literal which is the routine that restores the stack. One of the instructions in the literal does a SUB P,[.NAC,,.NAC]. This macro does not leave such easily recognizable data on the stack, but if you find a return address on the stack that is a literal that does the following, SAVEAC was used. If you look at the code in the literal, you will be able to tell which AC's were pushed on the stack and how many there are (.NAC is the count of AC's pushed). The stack is left like this:

```
/ AC
/ AC
/ .
/ .
/ last AC saved
/ address of literal to restore stack
```

The literal to restore the stack looks (approximately) like this:

LIT= address of literal to restore stack
for this example.

```
LIT-1/ 3,,3      ;count of AC's saved =3
LIT/ CAIA 0
/ AOS -N(P)
/ MOVE 1,-2(17) ;restore AC1
/ MOVE 5,-1(17) ;restore AC5
/ MOVE 10,0(17) ;restore AC10
/ SUB 17,LIT-1   ;reclaim stack locations
/ POPJ P,        ;return to caller
```

6. SAVEP

This macro calls the routine SAVP (in APRSRV) to save the AC's P1-P6 on the stack followed by the address RESTP, which is the routine to restore the AC's.

```
/ P1
/ P2
/ P3
/ P4
/ P5
/ P6
/ RESTP
```

7. SAVEQ

This macro calls the routine SAVQ (in APRSRV) to save the AC's Q1-Q3 on the stack, followed by the address RESTQ, which is the routine to restore the AC's.

```
/ Q1  
/ Q2  
/ Q3  
/ RESTQ
```

8. SAVEPQ

This macro calls the routine SAVPQ (in APRSRV) to save the AC's Q1-Q3 and P1-P6 on the stack, followed by the address RESTPQ, which is the routine to restore the AC's.

```
/ Q1  
/ Q2  
/ Q3  
/ P1  
/ P2  
/ P3  
/ P4  
/ P5  
/ P6  
/ RESTPQ
```

9. SAVET

This macro calls the routine SAVT (in APRSRV) to save the AC's T1-T4 on the stack, followed by the address RESTT, which is the routine to restore the AC's.

```
/ T1  
/ T2  
/ T3  
/ T4  
/ RESTT
```

.6.4 Stack Adjustment

any times the stack pointer is adjusted. Table BHC, indexed by , contains n,,n which may be added to or subtracted from the t pointer.

9.7 MACHINE STATES AND RELEVANT DATA BASES

9.7.1 PC Storage

1. PC at the time of the crash

Location BUGHLT contains the PC at the time of the crash.

2. PC when JSYS began

Two copies of the PC are saved on the stack.

3. PFL/PPC

Current PC of process when the process was last context switched. Can be exec or user mode PC.

4. PIFL/PIPC

Exec mode PC saved here while the software interrupt code is in progress.

5. Temporary PC storage

When the system is changing state, it must always be prepared for a context switch. This is a concern when a JSYS is starting, when a process blocks, and when a software interrupt begins. In each case, the PC is temporarily stored in case of a context switch while the state change is in progress.

1. SKDFL/SKDPC - PC saved here while process is blocking.
2. MONFL/MONPC - PC saved here while nested JSYS is starting.
3. ENSKR/ENSKR+1 - PC saved here while entering the scheduler via the ENTSKD macro. This is the PC the ENTSKD macro was called with.

9.7.2 AC Storage

3.1 AC Storage In The PSB -

Each process's PSB contains several storage areas for saving AC's. AC's are saved in the PSB in these cases:

1. Nested JSYS (JSYS called by a JSYS)

When a user called JSYS is in progress, AC block 0 contains the monitor's AC's (the current JSYS code AC's) and AC block 1 contains the user mode AC's. If the JSYS code does a JSYS, AC block 1 (user mode AC's) are saved in the UACB area and the AC block 0 AC's are moved to AC block 1. For each level of JSYS, the AC block 1 AC's are pushed onto the UACB stack and the AC block 0 AC's are moved to AC block 1. Therefore, the AC block 1 AC's are always the previous context AC's; i.e., the AC's when the JSYS was called.

If a nested JSYS is in progress, then the user mode AC's are the first stacked AC's in UACB. If the nesting is more than one level deep, then each subsequent JSYS's calling AC's are also saved in UACB; the current JSYS AC's are saved in UAC if the process is not currently running or in BUGACU if the process was running at the time of the crash. The maximum nesting level for JSYS's is 5; this limit is dependent on how much storage is reserved for AC stacking in UACB.

ACBAS is the "pointer" for the AC stack UACB, but is not stored as an address. The contents of ACBAS must be shifted left 4 places to make it an address. The resulting address is the first saved AC for the last pushed AC block; i.e. the saved AC's for the next higher level of nesting. If there are no saved AC's pushed on the stack, ACBAS contains its initial value of <UACB>B39-1=37677; if ACBAS contains anything else, there are pushed AC blocks saved in UACB.

2. Process is context switched while running in user mode.

The current AC's (i.e., the user mode AC's) are saved in block UAC.

3. Process is context switched while running in exec mode.

The current AC's (i.e., the exec mode AC's) are saved in block PAC. The previous context AC's are saved in block UAC. The AC's saved in UAC are the user mode AC's unless a nested JSYS is in progress; in this case, the AC's saved in UAC are the AC's the nested JSYS was called with. The user mode AC's for this case are saved on the AC stack called UACB; the user mode AC's are the first saved AC's on UACB.

4. Software interrupt processing

The exec mode AC's are saved in block PIAC while a software interrupt is in progress.

9.7.2.2 AC Storage At The Time Of The Crash -

1. BUGACS

Exec mode AC's at the time of the crash. Copied to current AC's when using FILDDT.

2. BUGACU

Previous context AC's at the time of the crash. These are the user mode AC's unless a nested JSYS was in progress, i.e., a JSYS called from a JSYS. If a nested JSYS was in progress at the time of the crash, BUGACU contains the AC's the current JSYS was called with. In this case, the user mode AC's are saved in the AC stack called UACB.

9.7.2.3 Summary Of AC Storage -

1. UAC

Previous context AC's are saved here when the user is context switched. For the currently scheduled process, UAC contains the AC's the last time the process was dismissed. Once again, if a nested JSYS was in progress, UAC's contains the AC's that JSYS was called with. In this case, the user mode AC's are saved in the AC stack called UACB.

2. UACB and ACBAS

Pushed AC blocks when nested JSYS in progress.

3. PAC

Exec mode AC's saved there for process when it is dismissed.

4. PIAC

Exec mode AC's saved there when software interrupt is in progress.

5. BUGACS

Exec mode AC's at time of crash.

6. BUGACU

Previous context AC's at time of crash.

0.7.3 Fork Scheduled Or Not

If a fork is scheduled, location FORKX contains the fork's system fork number. The scheduled fork's PSB and per process pages, JSB and per job storage, and page table are all mapped into the monitor's address space. If no fork is currently scheduled, location FORKX contains a -1.

0.7.4 Fork NOSKED

If a fork is NOSKED, its fork number is stored in SSKED; if there is not a NOSKED fork, SSKED contains -1. The FKSWP table contains status bits for both NOSKED (no other process may be scheduled) and CRSKED. CRSKED is a new process state in release for processes manipulating the monitor resident free pool.

0.7.5 Extended Vs. Non-extended Addressing

If the machine supports extended addressing, flag EXADDR contains 1; if the machine does not support extended addressing, EXADDR contains 0.

0.7.6 Sizes (resident, Non-resident, Total)

MONCOR/ number of pages in resident monitor

TOTRC/ total number of swappable core pages

NHIPG/ highest physical core page number

9.7.7 MDDT Page (release 3A)

When MDDT is in use for a process, DDTPPG (currently page 774) exists. If the running process's page 774 exists, then that process has been using MDDT; you might suspect that the crash was caused by an accidental deposit in MDDT or some such thing.

9.7.8 Interrupt Vs. Non-interrupt Level

9.7.9 Relevant Data Base For Each Machine State

9.7.9.1 JSYS -

1. Stack

UPDL

2. Initial stack setup

Initial UPDL set up for JSYS if from user mode:

- / PC at time of JSYS
- / PC flags at time of JSYS
- / PC at time of JSYS
- / PC flags at time of JSYS

Initial UPDL set up for JSYS if from exec mode (nested)

- / INTDF
- / MPP (for higher level JSYS)
- / PC at time of JSYS (return PC)
- / PC flags at time of JSYS

3. Previous PC

The return PC is pushed on stack; MPP is stack pointer for return PC.

4. Saved AC's

AC's saved in UACB if nested JSYS. See section on AC STORAGE for a description of UACB.

5. Saved stack pointer

If the JSYS was from user mode, this is not relevant. If the JSYS is nested, the previous JSYS also used this stack and the MPP pointer can be used to determine where the stack pointer was when this JSYS began:

```
Previous  
stack ptr-> /  
           / INTDF  
           / MPP (for higher level JSYS)  
           / MONPC  
MPP => / PC at time of JSYS (return PC)
```

6. AC usage

There is no standard for AC usage that all JSYS's conform to.

7. Related Storage

1. MPP -- points to:

--return PC for JSYS

--last location of initial setup for this JSYS

2. FPC = KIMUPC -- dispatch address for JSYS

3. KIMUU1 - last UO from user in format:

```
KIMUU1/ flags,,opcode  
      / JSYS number
```

4. INTDF

Indicates whether the process is NOINT, and to how many levels. Set to -1 if the process is not NOINT; greater than or equal to zero if the process is NOINT.

.7.9.2 Page Fault -

1. Stack

TRAPSK

2. Initial stack setup

The initial stack setup differs for each of three cases:

--page fault from user mode

--page fault from exec mode

--recursive page fault

1. Stack setup on page fault from user mode

/runtime
/return PC
/return PC flags

2. Stack setup on page fault from exec mode

/ AC1
/ AC2
/ AC3
/ AC4
/ AC7
/ AC16
/ TRAPSW
/ runtime
/ PC
/ PC flags

3. Stack setup on recursive page fault

/ AC1
/ AC2
/ AC3
/ AC4
/ AC7
/ AC16
/ TRAPSW
/ PC
/ PC flags

3. Previous PC

Saved on stack; see initial stack setup for where.

4. Saved AC's

Those AC's that are saved are kept on the stack. See the initial stack setup for where each AC is saved.

5. Saved Stack Pointer

TRAPAP

6. AC usage

Differs for each type of page fault.

7. Related Storage

1. TRPID --identity of page causing trap in form PTN,,PN or PTN

This is the identity of the page the page fault handler is working on. TRPID contains the page's page table identity while the page's page table is brought into core (if the page table was not in core).

2. TRPPTR

Storage address of the page the page fault handler is working on.

3. TRAPSW (copy of TRAPS0)

4. TRAPC

0 if first level page fault; if greater than 0, it indicates the level of recursion.

5. TRAPFL/TRAPPC = UTPFL/UTPFO

Flags and PC at time of page fault.

6. TRAPS0 = UTPFW

Page fail word; contains the address that page faulted.

.7.9.3 Scheduler -

1. Stack

SKDPDL

2. Initial stack setup

None

3. Previous PC

Saved in PSB for a process when context switch to the scheduler; the PC is saved in PFL/PPC of the process's PSB. If FORKX contains a fork number, it is the number of the fork running when the scheduler was invoked. If FORKX is not setup, you cannot determine which fork was running last.

4. Saved AC's

The process's AC's are saved in block PAC (exec mode AC's) and block UAC (previous context AC's) of the PSB for the process. If FORKX is not setup, you cannot determine which process was running last.

5. Saved Stack Pointer

In the saved AC's.

6. AC usage

FX/ -1 if no fork chosen or system fork
number of chosen fork

7. Related Storage

1. FORKX

FORKX contains a -1 if no fork is chosen or the fork number of the chosen fork.

2. Temporary storage while entering scheduler

9.7.9.4 Physio Queueing Level -

1. Stack

PHYPDL

2. Initial Stack Setup

The P and Q AC's are saved on the stack by the macro SAVEPQ; the AC's are saved in order of Q1 through Q3 followed by P1 through P6. See the section on Stack Usage for Local Storage for the format.

3. Previous PC

Since PHYSIO is called with a PUSHJ, the previous PC is the top of the saved stack.

4. Saved AC's

- AC's Q1-Q3 and P1 through P6 are saved on the stack. See the section in Stack Usage for Local Storage entitled SAVEPQ.

5. Saved Stack Pointer

The previous stack pointer is saved in PHYSVP.

6. AC usage

P4/ address of IORB being queued.

P1/ address of CDB

P3/ address of UDB

P2/ address of KDB or 0 if no KDB

7. Related Storage

.7.9.5 Physio Interrupt Level -

1. Stack

PHYIPD

2. Initial Stack Setup

None.

3. Previous PC

The previous PC is saved by the XPCW instruction in a two word block, beginning at the CDB-6.

4. Saved AC's

PHYACS -- block where AC's saved

5. Saved Stack Pointer

The saved stack pointer is in PHYACS+17.

6. AC usage

P1/ address of CDB

P2/ address of KDB or 0 if none

P3/ address of UDB

P4/ IORB address or argument
indicating action code:

P4<0 schedule a channel cycle
(P4) = -1
P4=0 dismiss interrupt
P4> housekeep current request
(contains IORB address)

7. Related Storage

Home block check funtion. In STG, starts at CHBUDB.

9.7.9.6 APR Interrupt Level -

1. Stack

MEMPP

2. Initial stack setup

/UPTPFO= TRAPPC
/UPTPFL= TRAPFL
/UPTPFN
/UPTPFW= TRAPSO

3. Previous PC

Saved as double word PC by XPCW in location PIAPRX and
PIAPRX1

4. Saved AC's

MEMPA -- block where AC's 0-10 are saved.

NOTE

Beginning in 3A, uses a different AC block while
at APR interrupt level; therefore, no AC's are
saved.

5. Saved Stack Pointer

MEMAP -- previous stack pointer saved there.

6. AC usage

7. Related Storage

1. Sets "local" page fail routine to MEMPTP

.7.9.7 DTE Interrupt Level -

1. Stack

DTESTK

2. Initial Stack Setup

None.

3. Previous PC

Saved in DTETRA.

4. Saved AC's

DTEACB == block where AC's saved.

5. Saved Stack Pointer

Previous stack pointer is saved in DTEACB+17.

6. AC usage

F/ result of CONI DTEn,

A/ DTE number of DTE that caused interrupt

3/ count (if RSX20F protocol)

4/ Byte pointer (if RSX20F protocol)

7. Related Storage

.7.9.8 PSI Handling -

1. Stack

. PIPDB

2. Initial Stack Setup
3. Previous PC,
PIDL/PIPC
4. Saved AC's
PIAC -- block where AC's saved.
5. Saved Stack Pointer
Previous stack pointer is saved in PIAC+17.
6. AC usage
FX/ interrupt flags from FKINT
7. Related Storage
 1. How to tell if interrupt in progress

9.7.9.9 Job 0 Exec Mode Tasks --

1. Stack
UPDL
2. Initial Stack Setup
3. Previous PC
Since these are scheduled processes, this is not relevant.
4. Saved AC's
Since these are scheduled processes, this is not relevant.
5. Saved Stack Pointer
Since these are scheduled processes, this is not relevant.
6. AC usage

7. Related Storage

1. How can you tell this use of UPDL from a JSYS?

If the FKJOB entry for the running fork is Job 0, then this is probably a Job 0 task as opposed to a JSYS in progress. Also, if the PC is in a Job 0 routine, this indicates a Job 0 task.

.7.9.10 User Mode -

he system never BUGHLTs in user mode; but it could KEEP ALIVE EASE. The PC is from user mode if the flag UMODF is set in the C.

.8 DDT'S

.8.1 FILDDT

he latest version of DUMP.CPY is the last crash. The program IBUT is used to analyze a crash.

.8.1.1 How To Use FILDDT On A Crash -

o look at a crash with FILDDT you need the dump and the monitor file it came from (for symbols). For example:

```
@ENABLE
$FILDDT
FILDDT>LOAD <SYSTEM>MONITR.EXE ;load symbols
FILDDT>GET <SYSTEM>DUMP.CPY ;load dump
```

he AC's contain their contents at the time of the dump. By default you look at physical (not virtual) addresses.

.8.1.2 \$U Command -

ILDDT can simulate KL paging. You can set paging using the ILDDT \$U command. This command allows you to specify the location of the page table different ways, depending on how much information you have on where the page table is. For more information on the \$U command, refer to the section on DDT version 41. In the part of the monitor that BOOT loads, there is one-to-one correspondence between physical and virtual

addresses; MMSPTN is in this part of the monitor's address space.

If you wish to look at some fork's address space, find its page table's SPT slot in the left half of FKPGS, indexed by fork number.

If you wish to return to physical addressing (i.e., no KL paging simulation), type \$\$U.

9.8.2 Relevant DDT/FILDDT Commands

These are standard DDT commands; however, you may not be familiar with them. They are included here along with examples of their use.

9.8.2.1 Question Mark (?) -

If you type a symbol followed by a question mark, DDT tells you which module(s) that symbol appears in; the module name is followed by a G if the symbol is global. A local symbol may be defined in more than one module.

This facility can be used to locate symbols, like GLOB, but faster.

```
BUGSTO?
APRSRV          ;symbol is local and defined in APRSRV

SPT?
STG G           ;symbol is global and defined in STG
```

9.8.2.2 Underscore (_) -

A value, followed by underscore, is a request to DDT to find a symbol with that value.

This facility can be used to locate the symbolic address of a value.

```
14156_LSCHEd+5
101400_SPT
```

8.2.3 Effective Address Search (\$E) -

The \$E command is used to search for all locations where the effective address, following all indirect and index-register chains to a maximum length of 64, base 10, equals the address being searched for.

The format of the command is ac\$E; a is the range and is optional. If no range is specified, the whole address space is assumed. The c argument is the address to search for.

```
MMSPTN$E
PGRI10+3/ MOVEM T1,MMSPTN
FPTA4/ SKIPA T1,MMSPTN
MLKPGM+2/ CAMN T2,MMSPTN
SWPER3+2/ CAMN T2,MMSPTN
GSMLER+11/ HRL T1,MMSPTN
BSMGP1+2/ HRL T1,MMSPTN
212777/ HRL T1,MMSPTN
SNPF0A+15/ HRL T1,MMSPTN
SNPF5B+10/ HRL T1,MMSPTN
UT1LL+1/ HRL T1,MMSPTN
```

```
JSB<JSB+5>0$E
JOBMAP+2/ 0
JOBMAP+3/ 0
JOBMAP+5/ 0
```

8.2.4 Word Search (\$W) -

Word search compares each storage word with the word being searched for in those bit positions where the mask, located at M, has ones. The mask word contains all ones unless set by the user. If the comparison shows equality, the word search types out the address and the contents of the location; if the comparison results in inequality, the word search types out nothing.

The format of the command is ac\$W. a is the range and c is the quantity searched for. To set the mask type n\$M where n is the quantity to be placed in the mask word.

Suppose we wish to find all share pointers in the current user's page map between pages 0 and 10. We want to store a 7 (for pointer type) in bits 0-2 of the mask. Our command is PTA<UPTA+10>200000,,0\$W and works as follows:

```
7000000,,0$M

UPTA<UPTA+10>200000,,0$W
UPTA+2/ 206000,,1244
```

UPTA+4/ 206000,,1242

9.8.2.5 Not Word Search (\$N) -

Not word search works like word search, the only difference is that it types out the contents of the register when the comparison is an inequality and types nothing when an equality is reached.

Not word search is commonly used to type out all non-zero locations in some range. Suppose we wish to find all existint (non-zero) entries in the JSB map.

```
-1$M
JOBMAP<JOBMAP+66>0$N
JOBMAP/ 224000,,635
JOBMAP+1/ 124003,,7044
JOBMAP+4/ 124003,,2764
JOBMAP+6/ 124003,,7050
```

9.8.3 MDDT

MDDT is a part of the monitor that allows you to look at the running monitor with the standard DDT commands; your process is always the running process when you use MDDT. You can also call monitor routines to map pages, etc.; however, extreme caution should be taken when using MDDT. If you change any locations, you can crash the monitor. It is a good practice to type carriage return immediately after you open any location to prevent accidental deposits into memory.

You can enter MDDT in either of two ways. In the first example, the running fork will be the top fork of your job; i.e., the EXEC. In the second example, the running fork will be the fork running user level DDT.

```
@ENABLE
$^EQUIT
MX>/
MDDT
```

```
@ENABLE
$$DDT
MDDT%$X          (Jsyz 777)
MDDT
```


One can use either method to enter MDDT. Return from MDDT by calling the routine MRETN. Do this by typing:

MRETNSG

.8.3.1 MAPPING MONITOR PAGES -

There is oftentimes a need to examine pages of the Monitor which are not currently part of its virtual address space. Examples might be the JSB or PSB of a hung job or fork or the Index Block of a file. The safest and easiest way to do this is to use the SETMPG routine (or MSETMP for a set of contiguous pages) to map the page(s) into the User's address space. Optionally read and write or just read access can be requested. The only valid reason for write access would be if you intended to change a location in the mapped page in order to un-hang the job.

The following is the setup for calling SETMPG or MSETMP:

```

AC1/      Internal Identity of page
          (in form 0,,SPT index
           or
           SPT index,,Page number)

AC2/      Access,,Address to map to
          ( 500000,,XXX000 for read access
            mapping to page XXX
           or
           540000,,XXX000 if read and write
            access desired)

AC3/      Number of Pages
          (if calling MSETMP)

```

The following is an example of mapping the JSB area pages of a job and the PSB for a fork:

```

ena
get <system>monitr.exe      ;get monitor's symbols
st 140                      ;DDT
DT

```

```

idt%$x                      ;go to MDDT
DDT

```

Mapping the PSB

```

cpq 40[ 1672,,1704          ;SPT index of fork 40's PSB

```

```

T1!      1704      ;internal ID
T2!      500000,,psbpga ;access,,destination

call setmpgsx      ;only one page so call
<>                SETMPG

mretn$g           ;return to user
<>

uptpf1/   CAIA 0      ;flags,,pc of fork's
UPTPFO/   T1,,.SFRKV+12 ; last page fault
UPTPFN/   PGRTRP

fkrtl     4160      ;fork's runtime

mddt$gx           ;return to MDDT
MDDT

;Mapping the JSB pages for a job

fkjob 40[   22,,2152      ;get fork 40's JSB index

T1!      2152,,0      ;SPT index,,page 0 of JSB
T2!      500000,,jsbpga ;access,,destination
                        of mapping (JSB area)
T3!      js1sta'1000-jsbpga'1000;number of pages to map

                        ;JSBPGA is the address of the first page
                        ; in the JSB area (the JSB itself)
                        ;JSBLSTA is the addresss of the last page
                        ; in the JSB area
                        ;the first N locations in the JSB (where
                        ; N is the number of pages in the JSB
                        ; area) contain pointers to the JSB
                        ; area pages

call msetmps$g     ;call the routine
<>

mretn$g           ;return to user mode
<>                to see mapped pages

jobmap[   224000,,2152      ;look at first few locations
JSBPGA+1[   124003,,7170    in JSB
JSBPGA+2[   0

```

BPGA+3[0

```

ilnen mljfn[ 331501,,331503 ;pointer to extension
31503[ 77777,,2 $t string for first JFN
31504/ EXE
  
```

```

SRNAM[ 3 ;user name
SRNAM+1[ 422371,,640620 $t;DONAH
SRNAM+2/ UE
  
```

By GET'ing MONITR.EXE you can map the pages to their own addresses and then be able to use the defined symbols rather than determining the offsets within a different page address. Since the pages have been mapped to the User's Address space rather than to the Monitor's it is not necessary to unmap them when one. A RESET will get rid of them.

.8.4 EDDT

When you are in EDDT, timesharing ceases. Only the EDDT process can run.

EDDT must be locked in memory. This is accomplished by one of the following:

- Bringing the monitor up with EDDT and setting a flag to request that EDDT be kept locked in memory.

- Calling the routine LCKINI from MDDT; this routine locks EDDT in memory and must be called BEFORE entering EDDT.

.8.4.1 Debugging Switches -

The locations described in this section can contain flags that are useful for debugging. The flags must be set before normal monitor startup; i.e. from EDDT startup. The default setting for each of the locations is zero.

EDDTF	1	Keep EDDT in core when system comes up
DEBUGSW	0 or 1	Stop on BUGHLTs
	2	Write-enable swappable monitor, start up Sysjob using the file SYSJOB.DEBUG for the Sysjob commands and stop on BUGHLTs.
DCHKSW	0	Do not stop on BUGCHKs

DINFSW 0 Do not stop on BUGINFs

9.8.4.2 Loading The Monitor With EDDT -

Observe the following steps to load the monitor with EDDT locked in memory:

Load the monitor and start it at location 141 (EDDT startup).

Set the EDDTF flag to 1; this keeps EDDT locked in memory. The DBUGSW switch can also be set at this time if desired.

Set any desired breakpoints in the resident monitor. Only the resident monitor is loaded; therefore you cannot access any part of the swappable monitor. The breakpoint which is usually set at this point is GOTSWM. When the montiro reaches GOTSWM, the swappable monitor is already loaded.

Start the monitor at 147. This is the normal startup address. Since EDDTF is set, EDDT remains in memory.

When it reaches the breakpoint at GOTSWM, you may wish to do one or more of the following: remove the breakpoint, write-enable the swappable monitor (Call SWPMWE), lock the swappable monitor (Call SWPMLK), set breakpoints, continue the monitor (\$P).

Whenever you hit an EDDT breakpoint, timesharing ceases until you continue from the breakpoint.

9.8.4.3 Page Faulting In EDDT -

If you reference a page that is swapped out while you are in EDDT, the page is NOT fauled in; EDDT types ? in this case. EDDT does not change the state of the system. You can cause a page to be brought in by typing SKIP LOCsX where LOC is some location in the page you wish to have swapped in.

9.8.4.4 How To Re-enter EDDT -

Once EDDT has been locked in memory, you may re-enter by doing the following: enter MDDT; type EDDTsG. If the system hangs, and you wish to enter EDDT: type control backslash to get into the command parser; examine the PC to see where the system is hung; start the monitor at location 140. If this fails, ask to

start at 141 to force a Reset and then enter EDDT.

9.8.4.5 To Leave EDDT -

To exit from EDDT (unless in a breakpoint), type MDDTSG to reenter MDDT; type MRETNSG to return from the Jsys.

9.8.5 Sending TOPS-20 Crash Dumps In To Be Analyzed

When sending your crashes in for analysis, here are some hints which will make the analysis easier:

Put a copy of the dump AND a copy of the monitor that was running at the time of the crash on a magtape. The tape should NOT be in DUMPER interchange format. Use a density of 1600bpi. Put a second copy of the dump and monitor on the tape to minimize the possibility of read problems.

Use the DUMPER Print command to make a hard copy of the savesets and file names on the tape, and send it with the tape.

Include the CTY output at the time of the crash.

CRASH DUMPS
=====

Each time there is a BUGHLT there is an automatic dumping of the system core image into PS:<SYSTEM>DUMP.EXE. If there is sufficient room on the disk the data that was previously in DUMP.EXE will be copied into DUMP.CPY by SETSPD after the system is reloaded. DUMP.CPY does not get deleted and you may find several generations of DUMP.CPY.

In the case you have set no auto reload you can dump the crash by and by typing /D to the system BOOT> prompt. You can get into BOOT if you are reloading the system by bringing the system up from the watch registers rather than hitting <ENABLE> <DISK> on the console. See the Operators Guide for a discussion of the meaning of the various watches on the DEC-20.

9-Sep-80

CRASH ANALYSIS

First when analyzing software or software/hardware problems be sure you have the proper tools:

1. Internals reference materials (tables and flowcharts).
2. A full copy of the current release microfiche MONITOR and EXEC.
3. A MONITOR CALLS REFERENCE MANUAL.
4. A SYSERR manual.
5. A listing of the SYSERR log, especially if hardware is suspected.
6. A CTY output for BUGHLTs and BUGINFs or other problem indications, or an accurate reproduction of this information.
7. Any other manuals you may need for reference such as the proper version Installation Guide, Operators Guide, System Managers Guide, etc.
8. A TOPS-20,BWR file.

You will need listings of the latest versions of monitor modules in case the microfiche are not up to date. FILDDT is on the customers distribution tape.

Be sure you have analysed the SYSERR log. Be sure, also, that you have looked up the BUGHLT and/or BUGCHKs in question in the listings (microfiche) and have at least read the comments around them. Probably tracing down how it got called is a good idea. If you happen to be without a GLOB (provided on microfiche) you can find the BUGHLT tag of interest in the monitor as follows:

```
$GET <SYSTEM>MONITR.EXE
$ST 140
DDT
ILPP3?          ; BUGHLT of interest followed by "?"
PAGEM G         ; it is defined in PAGEM and is global
```

Some other useful bits of information. There is a GLOB listing provided in the microfiche which contains a list of all the global symbols in the monitor. Most of the symbols are defined in the module STG.MAC. If you don't know a tag name but want to look at the storage for DTEs, say, look through STG. STG also contains some small portion

Code mostly to do with restart, start, auto reload, dispatches for I channels and A few scheduler tests. STG stands for storage. Note that some stuff may be defined in PROLOG, and of course lots of stuff is defined throughout the monitor. You may also want to get a listing of MACSYM to be able to understand the macros you see while reading the monitor listings; MONSYM is also useful at times. Be sure you know how PARAMS has been changed in case it has. See BUILD.MEM on the distribution tapes for the currently distributed information on what to do to change various system parameters in PARAM0.MAC. Be sure that you know about any variables that the site may have changed in STG as well.

9-Sep-80

EXAMINING THE MONITOR

Debugging a complex, multi-process software system is largely a matter of absorbing sufficient knowledge, experience and folklore about the particular system with a considerable element of personal preference, or 'taste' also involved. This document is a cursory description of features built into the system to aid debugging, and such folklore as can be described in written English.

There are four different versions of DDT that may be used to examine the monitor. Each is used for a different purpose and has special capabilities. The versions of DDT are:

1. UDDT (user DDT) used to examine or modify the MONITR.EXE file.
2. MDDT (monitor DDT) used to examine or modify the running monitor under timesharing.
3. EDDT (exec DDT) used to examine or modify the running monitor from the CTY in a stand-alone mode.
4. FILDDT used to examine dumps.

All the DDT's are versions of TOPS-20 DDT documented in the TOPS-20 DDT manual, and have all of the features described in the manual. See also the document DDT41.MEM.

The use of all four versions of the DDT's is the same and will be described latter, however, each version is started differently.

9-Sep-80

D
-28

To use UDDT to modify your MONITR.EXE file on system, you must give the following EXEC commands:

```
@GET <SYSTEM>MONITR.EXE
@START 140          or on Release 4 systems, @DDT
```

This causes EDDT to start in user mode. This is the same DDT that is used when examining any program. You may now look at or change any part of the monitor. If you make changes to the monitor and want to save it, you should get back to the EXEC by typing ^Z. Then you may save the monitor.

You will probably have to be enabled in order to save the monitor back in <SYSTEM>. This is the safest, best, and recommended method of putting patches into the monitor.

9-Sep-80

MDDT:

A version of DDT which runs in monitor space is available. It can examine and change the running monitor, and can breakpoint code running as a process but not at PI or scheduler level. When patching or breakpointing the swappable monitor, the normal write protection must be defeated, either by setting DBUGSW to 2 on startup, or calling SWPMWE. If you insert breakpoints with MDDT, remember monitor code is reentrant and shared so that the breakpoint could be hit by any other process in the system. In this event, the other process will most likely crash since it will be executing a JSR to a page full of zeros.

To use MDDT you must have WHEEL or OPERATOR capabilities. You first issue the EXEC command:

```
@ENABLE
$^EQUIT
```

```
MX>/ ; You are now in the mini-exec and receive a prompt
; of MX>. Now you give the "/" command:
```

```
MX>/ ; You are now put into MDDT. To return to the EXEC
; you can issue a ^Z or a ^C which produces a
; message like "INTERRUPT AT 17372" and returns you
; to the mini-exec. If you type a ^P in MDDT you
; will get a message, "ABORT", and be returned to
; the mini-exec. If you once go into the mini-exec
; the CONTROL-P interrupt is enabled and typing this
; character will return you to the mini-exec. This
; is a good thing to use when debugging programs
; that do CONTROL-C trapping. From the mini-exec
; you may give either:
```

```
MX>S ; or
MX>E
```

```
; The S is filled out as START and the E as EXEC.
; both of these commands will return you to the
; EXEC. See the document EXEC-DEBUGGING.MEM for more
; about ^P and getting out of the EXEC to MX> and
; returning from MX> to either your copy of the EXEC
; or the system EXEC.
```

```
; You may also give the command:
```

```
MRETNSG
```

```
; From MDDT to return directly to the EXEC. While
; in MDDT you may examine any core location in the
; running monitor. You may also change any location
; in the resident monitor (done frequently by
; accident). If you wish to change any of the
```

; locations in the swappable monitor you must give
; the command:

CALL SWPMWEsX

; To write enable the monitor. After you have made
; your changes you must give the command:

CALL SWPMWPsX

; to write protect the monitor again.

MDDT may also be entered from process level via JSYS:

JSYS 777sX

or

MDDT%\$X ; will enter MDDT from the context of the current process

If you wish to examine the system from the EXECs inferior fork
onitor context:

@ENA
\$SDDT
DDT

JSYS 777sX
MDDT

return to user context:

MRETNSG

se SETMPG to map pages to this context:

page 677 has been traditionally used for this;
but any unused page may be used. To make sure that the page
is currently unused type:

ADDRESS/ ? ; the question mark from DDT indicates that the
; page is nonexistent.

when the destination page has been found, set up AC2 as:

AC2/ ACCESS,,677000

If the page has its own SPT slot:

AC1/SPT INDEX

the source page does not have its own SPT slot, it will belong to
ther a file or process page table. It will be represented as an
index into this page table:

AC1/ SPT INDEX OF PAGE TABLE,,INDEX INTO PAGE TABLE

Access = read or/and or write access
Read/Write access = 140000 in LH

Therefore, to map a page, call with either:

AC1/SPT INDEX OF PAGE
AC2/140000,,677000

or

AC1/SPT INDEX OF PAGETABLE,,INDEX INTO PAGE TABLE
AC2/140000,,677000

AND SAY:

CALL SETMPG\$X

The page will then be mapped to page 677. In examining locations 677000-677777, you will be looking at the contents of the page.

If you desire to map another page into this slot, merely call SETMPG again with arguments for the new page. You need not first un-map the old page. However, when you are finished, page 677 should be un-mapped in the following manner:

AC1/0
AC2/ACCESS,,677000
CALL SETMPG\$X

WARNING:

Calling SETMPG incorrectly can crash the system. Be CAREFUL! Do not use SETMPG on a time sharing system if a crash will cause bad feelings.

9-Sep-80



NOTE

Not to be confused with ^EEDDT command
to get into UDDT used with the command
processor.

To get into EDDT you must bring the system up using the
witch-register. See the DECSYSTEM-20 Operators Guide for a
discussion of switches. Go through the KLINIT dialog and when you get
the prompt BOOT>, respond with:

```
BOOT>/L  
BOOT>/G141
```

The "/L" command causes the monitor to be loaded, but not started.
The "/G141" starts the monitor at location 141, which is a jump to
EDDT. You can use EDDT like UDDT under timesharing on the MONITR.EXE
file by giving the following commands:

```
SGET <SYSTEM>MONITR.EXE  
SSTART 140
```

EDDT is linked into the monitor and is always there. You may also get
to EDDT from MDDT by issuing the following:

EDDT\$G

From MDDT. This stops timesharing. To resume timesharing and /or get
back to MDDT give the command:

```
MDDT$G ; back to MDDT  
MRETNSG ; back to normal timesharing
```

Breakpoints may be inserted in the resident monitor with EDDT,
but not in the swappable monitor in general, because its pages may be
swapped out and be unavailable to EDDT. You can bring them in by
typing:

```
SKIP LOC$X ; where LOC is some address not in core
```

There are some locations in the monitor that are very useful when using EDDT for debugging. They must be set before going on to start the monitor.

They are:

EDDTF	1	keep EDDT in core when system comes up
	0	delete DDT when system comes up (default)
DBUGSW	0	do not stop on BUGHLTs, crash and reload
	1	stop on BUGHLTs (hit EDDT breakpoint)
	2	write enable swappable monitor, do not start up SYSJOB, and stop on BUGHLTs. Also it doesn't run CHECKD automatically on startup.
DCHKSW	0	do not stop on BUGCHKs (default)
	1	stop on BUGCHKs (hit EDDT breakpoint)
DINFSW	0	do not stop on BUGINFs (default)
	1	stop on BUGINFs (hit EDDT breakpoint)

In addition the symbol GOTSWM appears in the code just after the swappable monitor is loaded. So, if you want to debug the swappable part of the monitor you must put a breakpoint at GOTSWM (to get swappable part in core) by,

GOTSWMSB

Then start the MONITOR by,

147\$G

CALL SWPMLKSX

CALL SWPMLK is used to lock swappable monitor in core for debugging. You must have more than 96k of core to give this command since the resident and swappable monitor are larger than 96k. To start up the monitor after you have gone into EDDT and set up your breakpoints (remember the last two are used for BUGHLT and BUGCHK) give the command:

147\$G

or

SYSGO1\$G

If you are in EDDT and DBUGSW is not 2, that is the monitor is write protected, you can use the routines SWPMWE and SWPMWP to write enable and write protect the monitor. CALL SWPMWESX in DDT.

ILDDT:

ILDDT is distributed on the customer software tape.

The following is an chewed-up FILDDT.HLP file.

ET(FILE) FILE-SPEC

loads a file for DDT to examine. If you are looking at a monitor dump you must load DUMP.CPY explicitly. FILDDT looks for MUMBLE.EXE not JMBLE.CPY that is DUMP<ESC> will tell you that there is no such file it will load DUMP.EXE. When looking at a dump and you wish to load the symbols you must first issue the load command followed by the get command. Be sure that the file from which you get the symbols is the same version as the dump. Be sure, also that the monitor that was dumped is the same monitor you use for symbols. That is don't get UNMED symbols to use with MONBCH etc.

LOAD (SYMBOLS FROM) FILE SPEC

loads specified file and builds internal symbol table. This must be the first command to FILDDT before "GET" when looking at a dump. You will most probably use <SYSTEM>MONITR.EXE which would have been the monitor running at the time of the dump.

LOAD (FROM FILDDT)

returns to command level. You then may type a save command if a load command was just done to preload symbols. You will get a version of FILDDT that has the symbols you just loaded in it so you no longer need to "LOAD" symbols. You now have a monitor specific FILDDT, which is common practice for TOPS-10, but is not generally done for PS-20.

LP

prints something like this text.

ENABLE PATCHING

allows writing on an existing file specified by a GET.

ENABLE DATA-FILE

assumes file is raw binary (i.e. no ACs, and not an EXE file).

FEATURES:

EP\$OU Sets monitor context for FILDDT mapping. EP is a symbol which is equal to the page number of the EPT. (Rel 4)

<CTRL/E> Returns to FILDDT command level.

TRACKING DOWN UNMAPPED ADDRESSES:

The resident monitor may be looked at without any difficulties, but the swappable monitor may not be in core at the time of the dump. If the value of the symbol is in the swappable monitor you must sometimes go through the monitor map to find where the location really is. The location MONCOR contains the number of pages of resident monitor and the location SWPCPO contains the first page of real core for swapping. So if the value of the symbol is greater than contents of MONCOR times 1000 then it is in swappable monitor.

If the page of the swappable monitor you want to look at is in core it will probably not be in core in the location that it's address refers to since the dump is of core and relocation of pages does not happen. To find where a symbol really is in the dump, first type the symbol followed by an "=", DDT will respond with the value of this symbol. The value of the symbol can be divided into two, three octal digit, fields. The high order three digits are the page number and the low order three digits are the offset into the page.

If the value of the symbol is 324621 the high order three digits, 324, are the page number and the low order three digits, 621, are the offset into the page. To find the location of the page in question in the dump you must look at the monitor map indexed by the page number. For example:

MMAP+324/

would give you the monitor map word for page 324. This word contains some protection bits for the page and the address of the page when the dump was taken.

The page may have been in core, on the swapping area or on the disk at the time of the dump.

If bits 14-17 in the monitor map word are non-zero the page was on the swapping area or disk and is no longer available.

If bits 14-17 are zero then the page was in core, and the right half of the word contains the page number in the dump of the page you are looking for (the dump program overwrites the last several pages of memory, the dump therefore does not contain these last pages.)

If the page was in core the new address of the symbol you are looking for can be found by using the page number from the monitor map word and appending the offset into the page to it. For example if MMAP+324 contains 104000,,256; then the new address of our symbol would be 256621.

1. Address in the swappable monitor must be resolved in this manner. In addition address of 600000 and above are in the JSB or PSB (PSB is page 777) and must be resolved by finding the page containing the JSB or PSB of the process that was running when the dump occurred. There are some locations and tables in the monitor that make this easy:

NAME	INDEX	DESCRIPTION
FORKX	none	Number of the fork that was running at the time of the dump, -1 if in the scheduler.
JOBNO	In PSB	Job number to which current fork belongs.
FKJOB	Fork #	Job number,,SPT index of JSB
JOBDIR	Job #	logged in directory number
JOBPT	Job #	controlling TTY number,,top fork number
FKSTAT	Fork #	test data,,address of fork wait routine
FKPGS	Fork #	SPT index of page table,,SPT index of PSB

PT indexes are indexes into a share pointer table starting at SPT. To find the PSB of fork 20, you first look at FKPGS+20. If this location contains 425,,426, the word at SPT+426 is the pointer to the PSB. This pointer can point to disk, swap area, or a page in the dump. If bits 14-17 are zero it is a pointer to a page in the dump and the right half of the SPT word is the page number of the PSB in the dump.

When you look at a dump, you should first try to find why the dump occurred by looking at the location BUGHLT. If BUGHLT is zero then you should check the CTY log to find out why the dump was taken and for information like the PC at the time of the dump and the status of the I/O system. If BUGHLT is non-zero it is the address of where the BUGHLT was issued. You should look up the BUGHLT in BUGSTRINGS.TXT or I/O.SMAC to find additional information about the BUGHLT. If at this point you are not sure as to why the BUGHLT occurred, you will have to look at the listings for more information. A copy of BUGSTRINGS.TXT is in Appendix A of the Operators manual. You can find the location of the call to the BUGHLT by typing the BUGHLT tag to DDT followed by "?". DDT will tell which monitor module the BUGHLT is in and you can go to your microfiche and read all about the conditions precipitating the BUGHLT.

Next if necessary look at FORKX. If it contains a -1 the scheduler is running; otherwise it is the number of the fork that was running when the crash occurred. The registers are saved at BUGACS on a BUGHLT, but if BUGACS+17 contains something,,BUGPDL+n, then the registers are invalid and you must go to the SYSERR buffer to get the good registers. This is done by adding to the right half of the SYSERR buffer pointer, SEBQOU, the offset into the buffer for the loading and ACS, SEBDAT+BG%ACS. This value points to a 16 block of words containing the users ACS. You may have to chain down more than one queued-up SYSERR entry to get to the BUGHLT block.

Do not forget to get a print out of the SYSERR log which will give you and the field service representative much of the information you can get out of the dump. The SYSERR output is much easier to examine, however, clearly you cannot get as much info as you can from a dump.

Some other locations in the PSB of interest are:

LOCATION	DESCRIPTION
UAC	User's ACs when he did his last JSYS.
PAC	monitors ACs
PPC	processors PC
UPDL	users pushdown stack while in a JSYS
NSKED	0 = ok to run scheduler >0 = cannot run scheduler
INTDF	-1 = ok to receive software interrupts >= 0 , cannot receive software interrupts

It may be useful to know the status of a fork when it is hung or you are unsure of its status. This can be determined by looking at FKSTAT indexed by the fork number. The right half of this location is the address of a test routine and the left half is data to be tested. For example if FKSTAT+12 contains 23,,FKWAT, then fork 12 is waiting for fork 23 to complete. FKWAT is a routine that waits for another fork to complete and its data (the left half of the word) is the number of the fork it is waiting for. There are many different wait routines and you will have to look at the code to see what individual ones are waiting for. There is a memo on scheduler tests which details most all of the scheduler tests in the monitor.

You can easily determine all of the forks associated with a job by giving the commands:

```
-1,,0$M  
FKJOB<FKJOB+NFKS>N,,0$W
```

Where N is the job you are looking for. A fork structure can usually be determined by looking at the FKSTAT of the forks and seeing which forks are waiting on which forks. A FKSTAT of FKSKP indicates a fork is inactive.

You should refer to STG.MAC for other fork and job tables and other locations in the PSB and JSB of interest. All of the above locations can be examined with MDDT or EDDT while the monitor is running. Of course at these times you do not have to go through MMAP and the PSB and JSB that are in core are your own.

There are two separate patch areas in the monitor (FFF and SWPF). FFF is the resident patch area and SWPF is the swappable patch area. These two symbols should be updated to point to the next free location in

h patch area when a patch is inserted. PAT.. is defined to be equal to SWPF. By convention, all distributed patches are applied at FF. This serves the purposes of reducing confusion, always working until the patch area is exhausted, and leaving patches always present in a dump for the cases where that is important.

here are several general purpose routines that can be used to look at the monitor while it is running. These routines should be used with caution since it is certainly possible to crash the monitor by using them incorrectly. Two of the more general routines are MAPDIR, or mapping a directory into core, and SETMPG for mapping pages someone else's PSB or JSB) into core. You will have to look at the listing for the exact use of these and other general routines. Beware of the precautions that should be taken when using them. You can find the module they are located in by looking in the GLOB listing which is a cross reference listing of all the global symbols in the monitor. You get a GLOB listing in your microfiche.

9-Sep-80

BUGHLT, BUGCHK, BUGINF

The monitor contains a considerable number of internal redundancy checks which generally serve to prevent unexpected hardware or software failures from cascading into severely destructive reactions. Also, by detecting failures early, they tend to expedite the correction of errors.

There are two failure routines, BUGCHK and BUGHLT for lesser and greater severity of failures. Calls to them with JSR are included in code by use of a macro which records the locations and a text string describing the failure. The general form is:

BUG (TYPE,NAME,<STRING>)

Where type is HLT or CHK, and string describes the cause.

For example,

BUG(HLT,SKDPFL,<PAGE FAULT FROM SCHEDULER CONTEXT>)

The strings are constructed during loading and are dumped into a file. The BUGSTRINGS.TXT file will produce an ordered listing of the bug messages for operator or programmer use.

BUGCHK is used where the inconsistency detected is probably not fatal to the system or to the job being run, or which can probably be corrected automatically.

Typical is the sequence in MRETN in the SCHED module.

AOSGE INTDF
BUG(HLT,IDFOD2,<AT MRETN - INTDF OVERLY DECREMENTED>)

This BUGCHK is included strictly as a debugging aid. Detection of a failure takes no corrective action. This situation usually results from executing one or more excessive OKINT operations (not balanced by a preceding NOINT). It is considered a problem because a NOINT executed when INTDF has been overly decremented will not inhibit interrupts and will not protect code changing sensitive data.

BUGHLT is used where the failure detected is likely to preclude further proper operation of the system or file storage might be jeopardized by attempted further operation. For example, the following appears in the SCHED module:

MOVE 1,TODCLK ;CURRENT TIME
CAML 1,CHKTIM ;TIME AT WHICH JOB0 OVERDUE
BUG(HLT,JONRUN,<JOB 0 NOT RUN FOR TOO LONG>)

This check accomplishes two things:

1. A function of JOB0 is to periodically update the disk version of bittables, file directories and other files. Absence of this function would make the system vulnerable to considerable loss of information on a crash which loses core and swapping storage. JOB 0 protects itself against various types of malfunction, this BUGHLT detects any failure resulting in a hangup.
2. Detects if the entire system has become hung due to failure of the swapping device or some such event, on the basis that if JOB 0 isn't running, nobody's running.

NOTE

For Release 4, the program form the BUGxxx calls takes has been modified, and the new file BUGS.MAC contains hopefully useful information on each of the BUGxxx calls in one place. This should be considered a required debugging file.

BUGSW:

monitor cell, DBUGSW, controls the behavior of BUGHLT and BUGCHK when they are called. DBUGSW is set according to whether the system is attended by system programmers.

If C(DBUGSW)=0, the system is not attended by system programmers, so all automatic crash handling is invoked. BUGCHK will return +1 immediately, appearing effectively as NOP. BUGHLT will, if called from the scheduler or at PI level, invoke a total reload from the disk and a restart of the system. The BUGCHK/INF output will appear on the CTY and in the SYSERR log when JOB0 gets around to them.

If the system continues to run or is restarted properly, the location of the bug (saved over a reload) and its message will be reported on the CTY.

If C(DBUGSW).NEQ.0, the system is attended, and one of the EDDT breakpoints will be hit. This allows the programmer to look for the bug and/or possibly correct the difficulty and proceed. There are two predefined non-zero settings of DBUGSW, 1 and 2, which have the following distinction.

C(DBUGSW) = 1

Operation is the same as with 0 except for breakpoint action. In particular the swappable monitor is write protected and SYSJOB is started at startup as described.

C(DBUGSW) = 2

Is used for actual system debugging. the swappable monitor is not write protected so it may conveniently be patched or breakpointed, and the SYSJOB operation is not started to save time.

BUGCHK and BUGHLT procedures are the same as for 1.

The following is a summary of DBUGSW settings:

MEANING	0 Unattended	1 Attended	2 Debugging
BUGCHK action	NOP	Hit Breakpoint	Hit Breakpoint
BUGHLT action	Crash System	Hit Breakpoint	Hit Breakpoint
SWPMON write protect?	Yes	Yes	No
CHECKD on startup	Yes	Yes	No

Other console functions:

In addition to EDDT, several other entry points are defined as absolute addresses. The machine may be started at these as appropriate.

140	JRST EDDT	; go to EDDT
141	JRST SYSDDT	; reset and go to EDDT
142	JRST EDDT	; copy of EDDT address
143	JRST SYSLOD	; initialize file system
144	0	
145	JRST SYSRST	; restart
146	JRST SYSGOX	; reload and start
147	JRST SYSGO1	; start

The soft restart (address 145, EVRST) restarts all I/O devices, but leaves the system tables intact. If it is successful, all jobs and all (or all but 1) process will continue in their previous state without interruption. This may be used if an I/O device has malfunctioned and not recovered properly. The total restart initializes core, swapping storage and all monitor tables.

A very limited set of control functions for debugging purposes has been built into the scheduler. To invoke a function, the appropriate bit or bits are set into location 20 via MDDT. The word is scanned from left to right (JFFO). The first 1 bit found will select the function.

BIT 0:

Causes scheduler to dismiss current process if any and stall (execute a JRST .), with -1 in ACO. Useful to effect a clean manual transfer to EDDT. System may be resumed at SCHED0.

BIT 1:

Causes the job specified by data switch bits 18-35 to be run exclusively. Temporarily defeats JOB 0 not run BUGHLT.

BIT 2:

Forces running of JOB 0 backup function before halting the system.

9-Sep-80

BUG'TYP MACRO CHANGES FOR VERSION 4 OF TOPS-20

Version 4 of TOPS-20 will include some changes in the BUG code generation. The purpose of these changes is to generate a document describing the TOPS-20 BUGCHKs, BUGHLTs, and BUGINFs that are more descriptive than the previous BUGSTRINGS.TXT file.

The logistics of this change include moving the BUG definitions out of the monitor source listings and into a central source file. This source file will serve both as the definition file for the bugs and as documentation for the BUGS. This file is called BUGS.MAC and will be distributed to all sites on the distribution tape. These BUGS are still referenced in the source module where the bug is invoked but they are defined in BUGS.MAC.

This involves a modification to the old BUG macro and a new macro called DEFBUG. The BUG macro appears in the source modules and the DEFBUG macro appears in BUGS.MAC.

The format of the new BUG macro is as follows:

```
BUG (BUGNAM,<<x1,des1>,<x2,des2>...>)
```

This is placed in the monitor code where the BUG called BUGNAM is to occur. This macro executes a macro with name 'BUGNAM' which generates a XCT BUGNAM where the contents of BUGNAM is a JSR BUG'TYP. Following the location BUGNAM are the Accumulators to be printed (one AC per word) followed by SIXBIT/BUGNAM/. The Accumulators to be printed are defined with the DEFBUG macro while the locations specified in the BUG macro are for documentation only.

Accompanying this BUG macro is a DEFBUG macro which is placed in the file BUGS.MAC. This entry completely defines the BUG, including its type (BUGHLT, BUGCHK, or BUGINF) and documentation.

The format of the DEFBUG macro is:

```
DEFBUG (TYP,TAG,MOD,WORD,STR,LOCS,HELP)
```

For a description of the arguments to this macro see the SWSKIT article called BUGS.MEM.

In order to make listings (output from MACRO or CREF) more informative than before, the BUG macro will cause the statement of the short description displayed in the listing where the BUG macro is called. Also, the flavor of bug (INF, CHK, or HLT) and whether it's hardware or software related will be displayed in the listing. Hence the OVRDTA bug would appear in the listing as

BUG(OVRDTA)
;BUG Type: hardware-related BUGINF
;BUG description: PHYSIO - OVERDUE TRANSFER ABORTED

When fully documented, the BUGS.MAC file will be extremely useful for specialists. It will describe, in one convenient place, what the additional data printed on the console is, what caused the bug, and what the site or specialist should do if that particular bug occurs.

Here is a section of the current BUG definition/documentation for the BUG GIVTMR from BUGS.MAC:

EFBUG(INF,GIVTMR,JSYSA,SOFT,<GIVOK TIMEOUT>,<<T2,FUNC>>,<

ause: The access control job has not responded with a GIVOK within the designated time period.

ction: If this consistently happens with the same function code, you should see if the processing of the function can be made faster.

If there is no obvious function code pattern, you may need to increase the timeout period or rework the way in which the access control program operates.

at FUNC - the GETOK function code

>)

INF specifies the bug is a BUGINF. GIVTMR is the name of the bug. SYSA is the module that the bug would occur in. SOFT specifies that it is likely the bug is caused by a software bug. <GIVOK TIMEOUT> is the bug string. <T2,FUNC> specifies the data that will be printed on the operator's console. The initial spec called for the descriptor FUNC to be included in the operator's message but at this time, this descriptor is just for source documentation.

The blurbs following the initial line of the BUG definition attempt to describe to the specialist, in a more detailed manner than the description printed on the console, what it means when this bug occurs and what should be done first in order to resolve the situation. In this case the ACTION is to examine the GETOK routine which is executed for the additional data FUNC. This routine is getting hung up. Sometimes, the ACTION will state to call the hot line or to submit an PR. These descriptions will help the specialist be more informed about the bugs which may occur at one of their sites and save them the time of calling the hot line or searching through the source module for an idea of the problem.

TOPS-20 SCHEDULER TEST ROUTINES

The following is a tabulation of (hopefully) all of the scheduler tests used by the TOPS-20 monitor, time-frame approximately Release 3A. This includes ARPA and DECNET tests. This is the data one finds in the monitor table FKSTAT indexed by fork number for forks which have blocked and left the GOLST (i.e. LH(FKPT) contains WTLST). The format of the FKSTAT table words is TEST DATA,,TEST ROUTINE ADDRESS. The scheduler test routines are called periodically to determine if a process can be unblocked. This is indicated by a skip return from the scheduler test. A nonskip return is taken if the process cannot yet be unblocked.

When examining the monitor because of a hung job or fork, the FKSTAT table can often reveal the reason the fork is hung, and this sometimes even allows corrective action to be taken.

The table below gives routine name, what you should expect to see in the FKSTAT table, and the module in which the scheduler test is defined, followed finally by a short description of what the particular condition is which is being tested.

SCHEDULER TESTS

TEST ----	CONTENTS OF T1 AT TIME OF SCHEDULER CALL -----	DEFINED -----
BALTST	[CONNECTION #,,BALTST] Wait for network bit allocation.	[NETWRK]
BATTST	[UNIT #,,BATTST] Wait for US.BLK, the lock bit for the BAT blocks on the unit, in the UDB to be zero.	[DSKALC]
BLOCKM	[TIME,,BLOCKM] Wait for TIME in BLOCKM format which is the low order 17 bits of the desired future time to be compared against a suitably masked TODCLK.	[SCHED]
BLOCKT	[TIME,,BLOCKT] Wait for TIME in BLOCKT format which is a value that is shifted left 10 bits and compared against a suitably masked TODCLK, providing a longer delay than BLOCKM, but less precision.	[SCHED]
BLOCKW	[TIME,,BLOCKW] Wait for TIME in BLOCKW format (same as BLOCKM).	[SCHED]
CDRBLK	[UNIT NUMBER,,CDRBLK] Wait for card-reader offline, or not waiting for a card.	[CDRSRV]

CHKLOK	[ADDRESS,,CHKLOK] Wait for NSP block lock at address to free.	[NSPSRV]
COFTST	[TIME,,COFTST] Wait for job in FKJOBN to be attached or time in BLOCKT form to elapse.	[MEEXEC]
DBWAIT	[DTE #,,DBWAIT] Wait for the TO-10 doorbell from the given DTE.	[DTESRV]
DGLTST	[0,,DGLTST] Wait for DIAGLK lock to be free.	[DIAG]
DGUIDL	[UDB ADDRESS,,DGUIDL] Wait for the unit to show as idle in the UDB.	[DIAG]
DGUTST	[UDB ADDRESS,,DGUTST] Wait for the maintenance bit to set in the UDB.	[DIAG]
DISET	[ADDRESS,,DISET] Wait for contents of ADDRESS to be zero.	[SCHED]
DISGET	[ADDRESS,,DISGET] Wait for contents of ADDRESS to be positive.	[SCHED]
DISGT	[ADDRESS,,DISGT] Wait for contents of ADDRESS to be greater than zero.	[SCHED]
DISLT	[ADDRESS,,DISLT] Wait for contents of address to be less than zero.	[SCHED]
DISNT	[ADDRESS,,DISNT] Wait for contents of ADDRESS to be non-zero.	[SCHED]
DMPTST	[COUNT,,DMPTST] Wait for COUNT to be less than DMPCNT to indicate dump mode buffers freed.	[IO]
DSKRT	[PAGE #,,DSKRT] Wait for CSTAGE for PAGE # to not be PSRIP, meaning disk read completed.	[PAGEM]
DWRTST	[PAGE #,,DWRTST] Wait for DRWBIT to clear in CST3(PAGE #), meaning write completed.	[PAGEM]
ENQST	[FORK #,,ENQST] Wait for the lock on ENFKTB+FORK #.	[ENQ]
FEBWT	[ADDRESS OF FE UDB,,FEBWT] Wait for EOF or input bytes available from FE. Wake also on invalid assignment.	[FESRV]

FEDOBE	[ADDRESS OF FE UDB,,FEDOBE] Wait for output buffer empty and all bytes are acknowledged by the FE. Wake also if not a valid assignment.	[FESRV]
FEFULL	[ADDRESS OF FE UDB,,FEFULL] Wait for the current count of output bytes to be less than the count of bytes in the interrupt buffer. Wake also on invalid assignment.	[FESRV]
FORCTM	[SUPERIOR FORK INDEX,,FORCTM] Identifiable wait forever, forced termination.	[SCHED]
FRZWT	[PREVIOUS TEST,,FRZWT] Identifiable wait forever, frozen fork.	[FORK]
HALTT	[SUPERIOR FORK INDEX,,HALTT] Identifiable wait forever for halted fork.	[SCHED]
HIBERT	[TIME,,HIBERT] Wait for TIME in BLOCKT format.	[SCHED]
HUPTST	[<0:9>TIME<10:17>HOST #,,HUPTST] Wait for IMPHRT bit set for host or time out in BLOCKW form.	[NETWRK]
IDVTST	[0,,IDVTST] Wait for the lock on IDVLCK to free, lock it.	[IMPDV]
IMPBPT	[0,,IMPBPT] Wait for IMPFLG nonzero, or IBPTIM timer to run out, or IDVLCK lock free and output scan needed for the IMP.	[IMPDV]
JBOTST	[TIME,,JBOTST] Wait for JBOFLG set nonzero for explicit request or time in BLOCKT form to elapse.	[MEXEC]
JRET	[0,,JRET] Wait forever, interruptible.	[SCHED]
JSKP	[0,,JSKP] Unconditional skip used to schedule immediately.	[SCHED]
JTQWT	[0,,JTQWT] Wait for JSYS trap queue.	[SCHED]
LCKTSS	[ADDRESS,,LCKTSS] Wait for lock at ADDRESS to unlock, lock it.	[IO]
LKDSPT	[0,,LKDSPT] Wait for room in LDTAB table of directories currently locked.	[STG]

LKDTST	[INDEX INTO LDTAB,,LKDTST] Wait for bit in LCKDBT to clear, indicating directory unlocked.	[STG]
LODWAT	[ADDRESS OF STATUS WORD,,LODWAT] Wait for flag LP&LHC to set in the addressed word, indicating loading has completed of the VFU or RAM file.	[LINEPR]
PTDIS	[UNIT ADDRESS,,LPTDIS] Wait for an error condition on the addressed unit, or for all buffers cleared and no bytes still in the front-end, before finishing close operation on the device.	[LINEPR]
TARWT	[IORB ADDRESS,,MTARWT] Wait for IRBFA in the IORB to indicate that this IORB is no longer active.	[MAGTAP]
TAWAT	[UNIT #,,MTAWAT] Wait for all outstanding IORBs for unit to be finished.	[MAGTAP]
IDWT1	[UNIT #,,MTDWT1] Wait for the count of outstanding requests on the unit to go to one.	[MAGTAP]
NCPLKT	[0,,NCPLKT] Wait for lock NCPLCK to free, lock it.	[NETWRK]
ICTST	[0,,NICTST] Wait for SUMNR less than or equal to MAXNR or only one fork in BALSET.	[PAGEM]
JTTST	[<0:8>CONNECTION #,<9:17>STATE,,NOTTST] Wait for connection to leave state.	[NETWRK]
NSPTST	[0,,NSPTST] Wait for KDPFLG nonzero, indicating KMC11 wants service, or MSGQ nonzero, indicating messages to process.	[NSPSRV]
TTNTT	[<0:8>OPTION #,<9:17>LINE #,,NVTNTT] Wait for completed NVT negotiation.	[TTNTDV]
OFNLKT	[OFN,,OFNLKT] Wait for OFN unlocked--SPTLKB zero in SPTH(OFN).	[PAGEM]
PIDWAT	[FORK #,,PIDWAT] Wait for bit for fork in PDFKTB to set.	[IPCF]
SEBTST	[0,,SEBTST] Wait for SECHKF to go nonzero before starting Job 0 task to write queued SYSERR entries.	[SYSERR]

SEEALL	[0,,SEEALL] Waits for SNDALL to go to zero, indicating the send-all buffer available.	[TTYSRV]
SPCTST	[0,,SPCTST] Wait for a node.	[DTESRV]
TST	[0,,SPMTST] Wait for page in SPMPG to be on SPMQ or the time SPMTIM to expire.	[PAGEM]
ST	[0,,SQLTST] Wait for the special queues lock SQLCK and lock it.	[IMPDRV]
ST	[SDB ADDRESS OF STRUCTURE,,STRTST] Wait for the structure lock to be free.	[MSTR]
AT	[ADDRESS OF STATUS WORD,,STSWAT] Wait for flag CD%SHA to come on in the addressed word, indicating that cardreader status has arrived.	[CDRSRV]
AT	[ADDRESS OF STATUS WORD,,STSWAT] Wait for flag LP%SHA to set in the addressed word, indicating that printer status has arrived.	[LINEPR]
KT	[FORK #,,SUSFKT] Wait for fork to be on WTLST in either SUSWT OR FRZWT.	[FORK]
I	[PAGE #,,SWPRT] Wait for CSTAGE for PAGE # to not be PSRIP, meaning swap read completed.	[PAGEM]
IT	[0,,SWPWTI] Wait for NRPLQ nonzero. Increment CGFLG each time test is unsuccessful.	[PAGEM]
IT	[FORK #,,TCIPIT] Waits for no interrupts pending for FORK #.	[TTYSRV]
IT	[LINE #,,TCITST] Wait for line inactive, no fork in input wait, or input buffer non-empty.	[TTYSRV]
TCOTST	[LINE #,,TCOTST] Wait for line inactive, or output buffer not too full to add a character to it.	[TTYSRV]
TRMTS1	[0,,TRMTS1] Identifiable wait forever for inferior fork termination.	[FORK]

IR _{ST}	[FORK #,,TRMTST] Wait for FORK # to be on WTLST for either HALTT or FORCTM.	[FORK]
IRPOCT	[MINIMUM NRPLQ,,TRPOCT] Wait for NRPLQ to be above stated minimum or normal minimum. Increment CGFLG each time test is unsuccessful.	[PAGEM]
ISACT1	[LINE #,,TSACT1] Wait until line inactive, becoming active, or has a full length dynamic block assigned.	[TTYSRV]
ISACT2	[LINE #,,TSACT2] Wait for line available--inactive or fully active.	[TTYSRV]
ISACT3	[LINE #,,TSACT3] Wait for line inactive--dynamic data unlocked.	[TTYSRV]
ISTSAL	[0,,TSTSAL] Wait for SALCNT to go to zero, indicating the send-all is finished for this buffer.	[TTYSRV]
TBUFW	[NUMBER,,TBUFW] Wait for NUMBER of buffers.	[TTYSRV]
TIBET	[LINE #,,TIBET] Wait for line inactive or input buffer empty.	[TTYSRV]
TOAV	[LINE #,,TTOAV] Wait for line inactive and output buffer not empty.	[TTYSRV]
TOBET	[LINE #,,TTOBET] Wait for line inactive or output buffer empty.	[TTYSRV]
DITST	[0,,UDITST] Wait for at least two free IORBs on UIOLST.	[PHYSIO]
DWDON	[IORB ADDRESS,,UDWDON] Wait for IS.DON to set in IRBSTS for this IORB.	[PHYSIO]
PBGT	[CONNECTION INDEX,,UPBGT] Wait for LTDF connection done flag to set, or output buffers to appear.	[IMPDV]
SGWAT	[0,,USGWAT] Wait for lock on queued USAGE blocks to free.	[JSYSA]
VBWAT	[UNIT #,,VVBWAT] Wait for the MDA to reset TPVV handling EOF.	[TAPE]
ATTST	[<0:8>CONNECTION #<9:17>STATE,,WATTST]	[NETWRK]

TOPS-20 Monitor Internals
TOPS-20 SCHEDULER TEST ROUTINES

Page 9-28

Wait for connection to be in state.

WTFKT

[FORK #,,WTFKT]
Wait for fork to be on WTLST.

[FORK]

WTSPTT

[PAGE #,,WTSPTT]
Wait for share count on PAGE # to go to 1.

[SCHED]

9-Sep-80

MAPPING DIRECTORIES IN MDDT

Release 3 of TOPS-20 can take advantage of the extended addressing features of the Model B processor. Some of its data has been reorganized and moved into non-zero sections of the addressing space. One of the things moved was directories. Directories are now mapped into section 2, starting at the beginning of the section. Thus the old procedure of reading a user's directory in MDDT is no longer valid. This will describe how to map a directory correctly, for release 2 and for releases 3, 3A, and 4.

The procedure for release 2 was the following. You first have to find out the structure number and directory number for the directory to be mapped. You can use the TRANSL program to get the directory number, or use the ^EPRINT command to list the directory information. As an example, suppose you want to find the directory and structure information for the directory SNARK:<DBELL>. You run TRANSL and obtain the results:

```
TRANSL SNARK:<CURDS>
N: <CURDS> (IS) SNARK:[4,117]
```

The "programmer number" obtained is the directory number, in octal. In this example, the directory number is 117. If the directory is in bad shape, and you can't run TRANSL or use ^EPRINT, you will have to find out the directory number by looking at the output from a DLUSER or ULIST run, or from BUGCHK output.

To find the structure number, you have to work harder. If the structure is mounted as PS:, its structure number is always 0. For structures mounted other than PS:, you do the following. You get into DDT, and look at the table STRTAB. This table contains all of the addresses of the structure data blocks in the system. The first word of each structure data block is the structure name in SIXBIT. So you search the tables looking for the desired structure. The offset into the table STRTAB is then the structure number. For our example:

```
ENABLE
SDDT
DT
SYS 777$X
DDT
$6T
TRTAB/ ,8[ / PS
TRTAB+1/ M^I / REL3
TRTAB+2/ M_% / SNARK
```

In the example above, you see that PS: is the first structure, followed by the structures REL3: and SNARK:. Since the offset into

STRTAB was 2 for SNARK:, the structure number you want is 2.

Knowing the structure number and the directory number, you can now map the directory and look at it. When the directory is mapped, location DIRORA will point to the area in the monitor you can find it at. This is currently the address 740000. To save typing, you can use the symbol DA, which has the value 740000 (none of the examples here uses this symbol however). To map the directory, you call the routine MAPDIR which is in the module DIRECT. It takes two arguments. The directory number goes in AC1, and the structure number goes in AC2. For our example, the output looks like:

```
DIRORA[ 740000
740000/  ?
```

```
1! 117
2! 2
CALL MAPDIR$X
$$
```

```
740000[ 400300,,100
```

The skip return from MAPDIR means you have successfully mapped the directory. You can now look at the whole directory by examining the proper locations. The number of pages that are mapped by MAPDIR is 30, which is the length of a directory, so the whole thing is available to look at. By examining or changing location 740000+N in core, you are examining or changing location N of the directory. When you are finished, you can just leave MDDT by jumping to MRETN or by typing ^C.

In release 3, however, when you examine location DIRORA after calling MAPDIR, it doesn't have to contain 740000. If it does, then your machine cannot support extended addressing and the monitor is running the same as release 2 did. In this case you can ignore the rest of this document. If your machine does have extended addressing, when you examine location DIRORA you will see the number 2,,0. This address is now in section 2 of the monitor, and MDDT cannot read the data there directly. If you look at the location 740000 after calling MAPDIR, it will still be unreadable, since the directory is no longer read in there. Those pages are now unused..

To be able to read the directory now, you have to tell the monitor to map in the pages where you can see them with MDDT. The first step is to examine the location DRMAP. This location is the section pointer for section 2, where the directories are mapped. This is a share-type pointer, which contains the OFN for the desired directory in the right half. This number is one of the arguments for the MSETMP routine. MSETMP takes the following arguments. AC1 contains the OFN in the left half, and the first page number to be mapped in the right half. AC2 contains flag bits in the left half, and the address where you want to map the pages in the right half.

AC1 contains the number of pages to be mapped. For mapping directories, you can use 740000 as the address, and you want to map 30 pages. You also want to set flag bits so that the directory can be changed. For the example, you do the following:

```
DRMAP[ 224000,,147
```

```
!! 147,,0
!! 140000,,740000
!! 30
CALL MSETMP$X
```

After the call to MSETMP, the directory is now mapped in 740000, and you can proceed as you used to in release 2. When you are finished with the directory, you should call MSETMP again to unmap the directory. This is done by supplying the same arguments as before, except that ac 1 contains zero. As an example:

```
!! 0
!! 140000,,740000
!! 30
CALL MSETMP$X
```

Now you can simply ^C out of MDDT or jump to MRETN.

For Release 4 of TOPS-20, the various flavors of DDT have been trained to understand extended addresses, so the mapping contortions used for 3 and 3A are once again unnecessary. On extended machines one can reference section two directly as below:

```
IRORA[ 2,,0
,,0[ 400300,,100
```

When done, you can still just ^C out or jump to MRETN.

9-Sep-80

An Easy Way to Examine the PSB and JSB of Another Job

There is an occasional need to look at the state in detail of another job on the system. A common reason for doing this is to find the cause and cure of a "hung job" which cannot be logged out. To find out what the job is doing you usually start by looking at the JSYS stack in the PSB. But you cannot examine such data easily because the fork data in the PSB and the job data in the JSB are not in the monitor's address space until the fork is run. If you try to look at the PSB or JSB using MDDT you will see the data for your own fork. To look at the data for another fork you must do what the monitor does, and that is to map it.

A procedure for doing the mapping of a PSB or JSB was given in the release 3 and 3A SWSKITS. You first find the SPT index of the PSB or JSB you want to map, then you call SETMPG or MSETMP to set up pointers to the data, and then you can examine it. But there are several problems in using that method, which are:

1. You have to find an empty set of pages in the monitor's address space which can be used for mapping.
2. There is not enough room to map all of the PSB and JSB. So if you want to examine many different things you have to do the mapping many times.
3. The routines SETMPG and MSETMP do no validity checking of their arguments. Thus if you feed them bad data the system will probably crash. So if you need to map things many times your chances are you will make a mistake once too often.
4. The addresses of the data are not correct. To look at PPC for example, you can't just examine location PPC (which would be for your own fork). You have to look in the page you are using for mapping. So every reference has to be offset by some constant.
5. When you are done looking at the fork, you can't simply leave MDDT. You have to call SETMPG or MSETMP again to unmap the data.

Since that documentation was written I have found a procedure which is much easier. It eliminates almost all of the above problems. The procedure is this:

1. Do a "GET" of the file the monitor was loaded from, usually SYSTEM:MONITR.EXE.
2. Enter user mode DDT in the file you got, and then do a JSYS 777 to get into MDDT.
3. Find out the SPT indexes as before, and call MSETMP to map the PSB or JSB to the USER address space, in the correct place!!
4. Return from MDDT, and examine PSB and JSB locations directly, and see the correct data in the right place.
5. When you are done, just ^C and do a RESET.

The rest of this document will document step by step how the procedure above is done, by using an example. Assume that we wish to examine the state of fork 105, which belongs to job 21. We then

9-Sep-80

@ENABLE
\$GET PS:<SYSTEM>MONITR.EXE
\$START 140
DDT

!Get a copy of the monitor
!Get into user DDT

JSYS 777\$X
MDDT

!Enter MDDT

!Following is an example of the procedure to map the JSB of a job:

FKJOB+105[25,,2035

!Get the SPT index of the JSB
!of fork 105

T1! 2035,,0
T2! 540000,,JSBPGA
T3! JSLSTA'1000-JSBPGA'1000

!Put SPT index in left half
!* Flags and where to map to
!Number of pages to map

CALL MSETMP\$X
\$

!Do the mapping

!Following is an example of the procedure to map the PSB of a fork:

FKPGS+105[2657,,2332

!Get the SPT index of the PSB
!of fork 105

T1! 2332,,PSBMAP-PSBPGA
T2! 540000,,PSSPSA
T3! PSBMSZ

!Put SPT index in left half,
!and offset in right half
!* Flags and where to map to
!Number of pages to map

CALL MSETMP\$X
\$

!Do the mapping

9-Sep-80

Example of returning to user mode and looking at data from both the PSB and the JSB of the fork:

RETNSG	!Return to user mode
SRNAM[3	!Examine job's user name
SRNAM+1[422050,,546230 \$T;DBELL	
TRLTT[777777,,777777	!Controlling terminal
ILBYT+MLJFN[4400,,334010	!Start of data block for JFN 1
PC/ T1,,DISXE#+2	!Current PC of the fork
AC+17/ -215,,UPDL+62	!Current stack pointer
PDL/ CHKH05#	!First few stack locations
PDL+1/ CAM CHKAE0#+12	
PDL+2/ CHKH05#	
PDL+3/ CAM CHKAE0#+12	
PDL+4/ T1,,.COMND+1	
PDL+5/ -273,,UPDL+4	

Example of terminating the mapping we have done:

C	
RESET	!To finish, just quit and reset

The procedure as given above maps the JSB and PSB write-enabled. If you find something you want to change, you can simply deposit the new value into the location. If you want the data to be write-protected, then change the 540000 to 500000 in the two steps marked with an asterisk.

Warning: The procedure of mapping things into your user address space has its limitations. Mapping the JSB and PSB works because the user core used for mapping was previously empty. In general, you can only map things into your user core if your core pages are either nonexistent or are private. If you call MSETMP or SETMPG and map something over a shared page, the old file page is unmapped without the share counts being updated, which prevents your job from logging it later. To get around this problem you can BLT your core image to mark all of the pages to be private.

HOW TO USE BREAKPOINTS IN CODE THAT MANY USERS EXECUTE

When inserting a breakpoint into the running monitor, you have to be careful that no other users will execute the code containing the breakpoint. If some other user hits the breakpoint, they will blow up with an illegal instruction since MDDT will not be there to handle the breakpoint. This normally limits the places you can set breakpoints, since most of the monitor can be gotten to by any user. Even if you run the system stand-alone, it is possible that the routine you are debugging will be called by job 0. However, it is still possible to do such debugging, even on a system which is not stand-alone, and this document will describe how this is done.

The essential element of this technique is to put in the patch in such a way that only your own fork can ever reach the breakpoint. First you write a simple routine which will skip if it is not being run by your particular fork. This can be done easily if you remember that the location FORKX contains the currently running fork number. An example of such a routine is the following:

```
@ENABLE
SSDDT
DDT
JSYS 777$X
MDDT
```

```
FORKX[ 23 ; check our fork number

FFF/ 0 NOTME: PUSH P,T1 ; save an AC
NOTME+1/ 0 MOVE T1,FORKX ; get currently running fork number
NOTME+2/ 0 CAIE T1,23 ; is it us=23?
NOTME+3/ 0 AOS -1(P) ; no, setup skip return
NOTME+4/ 0 POP P,T1 ; restore the saved AC
NOTME+5/ 0 POPJ P, ; and return to caller
NOTME+6/ 0 FFF: ; reset the position of FFF
```

The routine above simply saves AC T1, gets the currently running fork number, compares it with your own fork number which you obtained by looking at location FORKX, and skips if they differ.

Now assume that you want to set a breakpoint into the following code, which is in the routine BLKSCN in the module DIRECT.

```

LKSC2/   HLRZ C,BLKTAB(B)
LKSC2+1/   CAME A,C
LKSC2+2/   AOBJN B,BLKSC2
LKSC2+3/   JUMPGE B,BLKSC2
LKSC2+4/   HRRZ B,BLKTAB(B)
  
```

Assume you want the breakpoint at location BLKSC2+3. You do the following:

```

LKSC2+3/   JUMPGE B,BLKSC2   FFF$<      ; patch this location
FF/   0   PUSHJ P,NOTME      ; call the NOTME routine
FF+1/   0   .SB   JFCL$>     ; me if it gets here, set breakpoint
FF+2/   JUMPGE B,BLKSC2
FF+3/   JUMPA A,BLKSC2+4
FF+4/   JUMPA B,BLKSC2+5
LKSC2+3/   JUMPA NOTME+6
  
```

Notice that the breakpoint has been set in the JFCL instruction following the call to NOTME. Only your fork will execute it, so you can now debug the section of code while other users are executing it at the same time. Remember to remove the breakpoint when you are done.

To run a particular program while having breakpoints set, you must remember that the breakpoint has to be set by the same process which you expect to hit it. So for example, typing ^EQUIT, setting a breakpoint, returning to the EXEC and running your program will not work. You must enter MDDT and set the breakpoints from your program you want to debug. As an example:

```

ENABLE
SET PROGRAM      ; get the program to be used
DDT              ; enter DDT
)T
SYS 777$X        ; and enter MDDT from there
)DT

PUT IN "NOTME" ROUTINE AND SET BREAKPOINTS HERE)

RETNSG           ; return to the context of the test program
;               ; start the test program
  
```

Using Address Break to Debug the Monitor

Sometimes when examining a set of dumps, you will notice the crashes are caused by some location being destroyed. If you have no idea where the destruction is done from, finding the problem could be very difficult. One useful procedure in such cases is to use the address break feature of the hardware to track down the problem (except for 2020's!). The only problem is that the use of address break is not obvious. This is a manual describing how to use address break in the TOPS-20 monitor.

In order to use address break, four things must be done. First, the current routines the monitor uses to set address breaks for users must be disabled. Secondly, your own address break must be set from MDDT or EDDT. Thirdly, instructions which you want to execute properly have to be modified so that they will not cause an unwanted address break. Finally, breakpoints must be placed in the monitor so that the state of the monitor can be examined when the address break occurs. The following is a step by step example of doing this.

1. Load the monitor for debugging, and enter EDDT. The procedure starting from BOOT is the following:

```
BOOT>/L                ;Load monitor but don't start it
BOOT>/G140             ;Start EDDT
EDDT
DEBUGSW/ 0 2           ;Set debugging mode
EDDTF/ 0 1             ;Keep EDDT once system starts
GOTSWMSB               ;Install useful breakpoint
SYSGOI$G              ;Start the monitor
```

```
[PS MOUNTED]
$1B>>GOTSWM 0$1B       ;Remove breakpoint now
```

2. Disable the monitor's normal changing of the address break. This is currently done at two places:

```
KISSAV+4/ DATAO UNPFG1+26 JFCL ;Disable instruction
SETBRK+12/ DATAO A JFCL       ;Here too
```

1. Set your own address break at the desired location. Refer to the Hardware Reference Manual for details. The instruction to set an address break is:

```
DATAO APR,ADDR ;Note: APR = 0
```

where ADDR contains the following fields:

Bits	Description
-----	-----
9	Break at given address on instruction fetches
10	Break at given address on reads
11	Break at given address on writes
12	0=exec address space, 1=user address space
13-35	Address to break on.

So now assume you want to catch a bug which is blasting location CURDS. You want to break only for writes, and want to use exec virtual space. Therefore you type the following:

```
FFF/ 0 100000000+CURDS ;Put data in convenient place
DATAO APR,FFFSX ;Set the address break
```

Now you want to disable address break for all instructions which you expect to change the given location. Assume in this example that only location DIDDLE should change location CURDS. Then you do the following for a model B CPU:

```
FFF! IT: ;Define location to get old flags
IT+1! ;Old PC
IT+2! ;New flags
IT+3! IT+4 ;New PC
IT+4! EXCH IT ;Save AC and get old flags
IT+5! TLO 1000 ;Set address break inhibit bit
IT+6! EXCH IT ;Restore flags and AC
IT+7! JRST 5,IT ;Return to caller
IT+10! FFF: ;Redefine FFF

DIDDLE/ MOVEM A,CURDS FFFS< ;Insert patch
FFF/ 0 JRST 7,ITS> ;Call above routine
FFF+1/ 0 MOVEM A,CURDS ;Typed by DDT when finishing patch
FFF+2/ 0 JUMPA A,DIDDLE+1
FFF+3/ 0 JUMPA B,DIDDLE+2
DIDDLE/ MOVEM A,CURDS JUMPA IT+10
```

The JRST 7,IT instruction is used to save the old PC at IT and IT+1, and take a new PC from IT+2 and IT+3. There the old PC is changed to include the address break inhibit bit. Then a JRST 5,IT is done which returns to the caller. The next instruction then executes without causing an address break. You have to insert the JRST 7,IT instruction at every

instruction you want to succeed.

For model A CPUs the procedure is similar, but a little easier:

```

FFF!   IT:                                ;Define location to hold PC
IT+1!  EXCH IT                            ;Get old PC and save AC
IT+2!  TLO 1000                          ;Set address break inhibit flag
IT+3!  EXCH IT                            ;Restore PC and AC
IT+4!  JRSTF @IT                          ;Return to caller
IT+5!  FFF:                              ;Redefine FFF

DIDDLE/  MOVEM A,CURDS   FFFs< ;Insert patch
FFF/    0   JSR ITS>      ;Call above routine
FFF+1/  0   MOVEM A,CURDS ;Typed by DDT when finishing patch
FFF+2/  0   JUMPA A,DIDDLE+1
FFF+3/  0   JUMPA B,DIDDLE+2
DIDDLE/  MOVEM A,CURDS   JUMPA IT+5

```

5. Now put the breakpoints into the monitor so that when an address break occurs, you will get into EDDT. There are two locations to patch, one for PI level and one for non-PI level. You also have to patch a monitor bug in release 3 and 3A so that the page fail dispatch code works properly.

```

ADRCMP$B                                ;Set breakpoint at non-PI routine
PFCD23$B                                ;Set breakpoint at PI routine
PIPTRP+1/  MOVE A,TRAPSW   MOVE A,TRAPSO ;And fix a bug
$P                                                ;Now let the monitor proceed

```

6. When either of the above breakpoints is hit, the flags and PC of the instruction which caused the address break will be in locations TRAPFL and TRAPPC. If the address break was from JSYS level (breakpoint was to ADRCMP and location INSKED is zero) then an \$P will proceed properly. If the address break was from the scheduler or from PI level, doing \$P will be useless since the monitor will then BUGHLT because it doesn't want to see an address break under these conditions. However, this is ok if all you want to do is find the instruction causing the trashing.

If the location still gets trashed after trying to catch it this way, either your procedure is wrong; you are trying this on a 2020 (which has no address break feature); the location is being changed by some IO being done (RH20s, DTEs, etc); or else the machine is having some hardware problems.

RECOVERING FROM DIRECTORY ERRORS

Sometimes after a monitor crash due to disk problems, some of the directories on the system will contain errors. These errors cause UGCHKs such as DIRFDB, NAMBAD, DIRPG0, and DIRPG1. It is sometimes possible to find the error in the directory by getting into MDDT, mapping the directory, finding what is wrong, and fixing it. This procedure is described in the SWSKIT. However, this is not always easy, and may take a lot of time. It is therefore better in many cases to simply delete the bad directory and recreate it. This is easy to do for most directories. But special procedures are necessary for the directories <SYSTEM> and <SUBSYS>. The rest of this memo will describe the methods of recovering from bad directories, handling in particular the difficult case of the <SYSTEM> directory.

You can first try to give the EXPUNGE command with the REBUILD and PURGE subcommands. If the problem with the directory is very simple, it may fix your problem. As an example, suppose the directory PS:<SICK-DIRECTORY> is incorrect. You would type:

```
$EXPUNGE (DIRECTORY) PS:<SICK-DIRECTORY>,  
$$REBUILD (SYMBOL TABLE)  
$$PURGE (NOT COMPLETELY CREATED FILES)  
$$  
PS:<SICK-DIRECTORY> [NO PAGES FREED]  
$
```

If this does not help the problem, you will have to delete the directory and then recreate it. Before proceeding, you should make sure that any files you can reference are copied to another directory, or else are saved on tape. Now first try to delete the directory normally, as follows:

```
$BUILD (USER) PS:<SICK-DIRECTORY>  
[OLD]  
$$KILL  
[CONFIRM]  
$$  
$
```

If this is successful, then simply recreate the directory again, and restore the user's files. You should recreate the directory with the same directory number as it had before, so that DLUSER's data will still be correct.

The procedure above will fail if either the directory is mapped by another job, or if it is totally unusable. If it is mapped, and the directory is a random user, you can wait until the directory is no longer in use, or you can take the system stand-alone so that no user can reference it.

If the directory is totally unusable, you will then have to try to delete it the hard way. Before proceeding, you should try to delete and expunge all files in the directory. This will minimize the amount of lost pages that will result. Now there are two cases to consider. If the directory is not a sub-directory, you type the following:

```
$DELETE (FILE) PS:<ROOT-DIRECTORY>SICK-DIRECTORY.DIRECTORY,  
$$DIRECTORY (AND "FORGET" FILE SPACE)  
$$  
  <ROOT-DIRECTORY>SICK-DIRECTORY.DIRECTORY.1 [OK]  
$
```

If the directory is a subdirectory, you modify the above command by replacing "ROOT-DIRECTORY" by the name of the next higher directory. Thus if the directory was PS:<ANOTHER,BAD-ONE>, you type:

```
$DELETE (FILE) PS:<ANOTHER>BAD-ONE.DIRECTORY,  
$$DIRECTORY (AND "FORGET" FILE SPACE)  
$$  
  <ANOTHER>BAD-ONE.DIRECTORY.1 [OK]  
$
```

The above procedure tells the monitor to treat the directory file like a normal file, and to delete it as such. This means that any files in the directory will become "lost". The disk pages can be recovered later with CHECKD. If the above works, you simply can recreate the directory and restore the files.

The only reason the above command should fail is if the directory is still mapped. For PS:<SUBSYS>, you can bring up the system stand-alone so that no programs are run from it, and then delete it. For PS:<SYSTEM>, even taking the system stand-alone will not help, for it is always mapped by job 0. But there are two procedures you can use which do work.

The safest method can be used if the user's system has mountable structures. If you have built another PS: structure, you can mount the pack with the bad directory as an alias, and then the directory will not be mapped and can be deleted. As an example:

```
$SMOUNT (FILE STRUCTURE) SICK:,  
$STRUCTURE-ID (IS) PS:  
$$  
WAITING FOR STRUCTURE SICK: TO BE PUT ON LINE...  
STRUCTURE SICK: MOUNTED  
$  
$DELETE (FILES) SICK:<ROOT-DIRECTORY>SYSTEM.DIRECTORY,  
$$DIRECTORY (AND "FORGET" FILE SPACE)  
$$  
SICK:<ROOT-DIRECTORY>SYSTEM.DIRECTORY.1 [OK]  
$
```

Then you can build the new directory, restore the files to it, and then use it again for your normal PS: pack. Be sure to build the new directory with the same number. This is especially important for the special system directories.

If you do not have another disk drive or another PS: disk, or if you don't want to bother SMOUNTing the disk, you can fix the <SYSTEM> area by using MDDT. The basic idea is to patch the monitor so that it no longer thinks that the directory is in use. This is done as follows:

```
$^EQUIT  
  
INTERRUPT AT 17117  
MX>/MDDT  
CHKOFN/ JSP CX,.SAVE JRST RSKP  
MRETN$G  
  
$
```

Then you should have no problems deleting the directory. Immediately after doing the delete, you should reload the system. When the system restarts, you can read the monitor and the EXEC either from the distribution magtape or from another directory where you had kept copies. Then recreate the <SYSTEM> area, making sure to give it the same directory number as it had before. Then you can restore the files and let the users back on. Finally, you should run CHECKD to come to recover the lost pages.

INDEX

020	6-1
o references	1-10
objsp	1-10
objn	1-6
atch class	3-5
las control	2-1
oot	9-3
ught	9-5
yte instructions	1-11
lass scheduling	3-1
ompatibility	1-5, 1-12
efault section	1-6
irectory cache	7-2, 7-15
imp.exe	9-2
de	9-31
ffective address computation	1-5
ecute-only	4-1
extended addressing	5-1, 9-15
extended format indirect word	1-7
lddt	9-25
rmware	1-1
ork history	8-3
lobal addresssing	1-2
lobal stack pointer	1-11
idden symbols	5-6
mediate mode	1-9
plementation	1-2
crementing the pc	1-4
indexing	1-6
idirection	1-6
instruction format indirect word	1-7
ip	1-10
ir	1-10
.	6-1
lex	9-2
ad averages	7-2, 7-13
ocal addressing	1-2

Mddt	9-28
Model b kl	1-1, 1-3
Monitor address space	5-1
Monitor modules	6-1
Monitor stacks	9-7
Opr	2-2, 3-5, 3-7
Page map	1-2
Pmap%	1-2
Pop	1-10
Popj	1-10
Postpurging	8-1
Preloading	8-1
Program counter	1-3
Push	1-10
Pushj	1-10
Rsmapt%	1-2
Section	1-2
Section map	1-2
Sked%	2-2
Smap%	1-2
Support	1-1
Syerr%	9-3
Syserr	9-3
Usectb	1-2
Vax/vms	1-1
Virgin process	4-1
Virtual address space	1-4
Watch	7-1
Windfall	3-2
Working set swapping	8-1
Xhlll	1-9
Xjrstf	1-5
Xmovei	1-9

Copyright (C) 1978, 1979

Digital Equipment Corporation, Maynard, Massachusetts, U.S.A.

This software is furnished under a license and may be used and copied only in accordance with the terms of such license and with the inclusion of the above copyright notice. This software or any other copies hereof may not be provided or otherwise made available to any other person. No title to and ownership of the software is hereby transferred.

The information in this software is subject to change without notice and should not be construed as a commitment by Digital Equipment Corporation.

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

1.0 Introduction and Overview

This document is designed as a users guide to DDT version 41 in so far as it has changed from previous versions of DDT. It is not a complete users guide to all the wonders of DDT, just those new features which have recently been implemented (although directed primarily at new features only in DDT version 41, some documentation is included to describe other aspects of DDT which have been around for a longer period of time, but were never fully understood or otherwise documented).

Throughout this document it is assumed that the reader is already familiar with DDT and the MACRO assembly language in general as well as the appropriate operating system(s).

This is the first revision of this document, incorporating the additional changes to DDT version 41 as of edit 260.

2.0 Configurations

DDT version 41 will run on KA-10's, KI-10's, KL-10's, and KS-10's, using no paging, KI-paging, or KL-paging, with or without extended addressing in user or executive mode (user and file DDT's run only in user mode) with no special assembly needed. DDT version 41 must be assembled to run under either the TOPS-10 or the TOPS-20 operating system.

It traditionally has been a goal to maintain one single set of source files from which all flavors of DDT are built. This goal has been maintained.

Note

TOPS-20 UDDT (and SDDT) now use memory locations 764000 through 777777 (previously 770000 through 777777), but the starting address for DDT continues to be location 770000. This requires version 4 of the PA1050 "compatibility package".

3.0 Memory and Address Control

The single biggest change to DDT version 41 from earlier versions is in the realm of memory control and how the user addresses memory locations.

3.1 Extended addressing

All flavors of DDT except FILDDT will run in any memory section. Full extended addressing is supported, as are "large" addresses - DDT will now accept a full 36-bit expression as an address although obviously only FILDDT can actually handle an address over 30-bits wide. In all cases the actual address must be positive (i.e., effectively a 35-bit address).

1.1 Symbol table restrictions

There are certain restrictions however which must be adhered to in order for DDT to function correctly. The first restriction is that the symbol table logic is essentially section-dependent, i.e., the symbol table and its pointers (.JBSYM=116 and .JBUSY=117, also .JBHSM=6 relative to the start of the "high segment") must reside (i.e., be mapped) in the same section as that in which DDT itself is running. Further, the symbol table can be no longer than 128K words in length and must be in RADIX-50 format.

Much thought is being given towards the implementation of a totally new symbol table scheme which would address all of these problems, the single biggest one of which is simply how is extended addressing going to be used - as a single fixed address space with one or more "global" symbol tables (like the TOPS-20 monitor currently works), or as a collection of independent sections each of which has section-local symbols/symbol tables (whatever that means), or what?

1.2 Breakpoint restrictions

The second restriction of which the user must be aware concerns breakpoints. Since the hardware has no facility to unconditionally transfer control to DDT using only 36-bits, DDT must be mapped into each section (at the same relative address obviously) which contains code into which the user wishes to place breakpoints.

1.3 Location examining restrictions

When running on an extended addressing machine if DDT is running in section 0, then only locations within section 0 (addresses 0 to 17777) may be manipulated. DDT will make no effort to outsmart the combined efforts of the user and the operating system by sneaking into non-zero section even momentarily to do the memory reference.

2 Effective address calculation

DDT version 41 can calculate effective address references using either "local" or IFIW (Instruction Format Indirect Word) or "global" or EFIW (Extended Format Indirect Word) formats. In a normal DDT address-opening command ("/", "\", <TAB>, etc.) a single <ESC> delimiting the address expression (e.g., "MOVE 3,@200(10)\$/" or just "\$[") instructs DDT to treat the expression as an IFIW word and calculate the effective address exactly like the hardware would, were the hardware to recode that 36-bit word as an instruction at location "." (whether or not location "." is currently open).

So <ESC>'s delimiting the address expression instructs DDT to treat the 36-bit expression as an EFIW word and calculate the effective address exactly as the hardware would, were the hardware to indirectly address the 36-bit expression at location "." (whether or not location "." is currently open). A strange case can come up about which the user should be cautioned - there is an ambiguity as to where (i.e., in which section) to start the effective address calculation. DDT assumes the left half of "." (i.e., the last location opened by the user). If for example having opened location 0,1234 which contains 4321 the user issues the command "\$s[" then DDT will calculate the effective address as the contents of location 4321 in section 0 indexed by the right half of register 7, and if bit 13 is on, treating

that word as an IFIW and continuing the address calculation. This, although probably not what was expected, is in fact exactly what the hardware would do since the indirect word came from section 0. Had the user opened location 1,,1234 (containing 7,,4321) then DDT would take the contents of location 7004321 and continue from there.

If no <ESC>'s delimit the address expression, then DDT simply uses the full 36-bit expression as the address (e.g., "30,,30/" says open location 30000030 and "-1/" says open location 777777777777). Again, only FILDDT can actually reference an address greater than 30-bits wide (not that anyone has that much disk space, but the hardware will not permit an address space over 30-bits wide), and in any case the address must be a non-negative 36-bit integer.

There is a special case in which DDT does something "kinky" - if a space was typed in entering the address expression, or if no explicit address was typed (i.e., the user is using the "last word typed" by simply typing only (for example) <TAB>), DDT will form the 36-bit actual address by using only the right half of the 36-bit address expression plus the left half of "." as the section number. This not-at-all-obvious behavior is so that the user can type in expressions such as "JRST PAT<TAB>" and have DDT go to location PAT in the same section as the JRST PAT instruction rather than going to address 254000000000+(PAT modulo 2**18). Another common usage of this "feature" would be in chaining down linked lists where the link pointer is an 18-bit section-local address in the left half of a word. To do this the user may type "sp\$\$Q/" (where "sp" means space). This is one of those cases where usefulness outweighs cleanliness of implementation and documentation.

3.3 Modifying memory

Two new commands have been added to facilitate DDT's manipulation of the user address space.

3.3.1 Automatic write-enable

The \$W or \$OW command instructs DDT to, if the user attempts to deposit it into a write-protected memory location, automatically attempt to write-enable the memory location, do the memory deposit, then finally re-write-protect the memory location (default for TOPS-10); the \$\$W or \$\$OW command instructs DDT to simply give an error indication if the user attempts to change a write-protected memory location (default for TOPS-20). For FILDDT the use of this command is restricted to non-file usage such as "DDT'ing" the running monitor/memory space.

3.3.2 Automatic page-creation

The \$1W command instructs DDT to automatically try to create the page the user is trying to deposit into if it doesn't already exist (default for TOPS-20); the \$\$1W command instructs DDT to simply give an error indication if the user attempts to write into a non-existent page (default for TOPS-10). EDDT and FILDDT doing super I/O or "DDT'ing" an .EXE file will NEVER attempt to create a non-existent page. For FILDDT the user must specify patching the file when he starts FILDDT in order to be able to create new pages (e.g., extend the file or fill in a gap in the middle of the file (TOPS-20 only)).

4 Page mapping and physical addressing

In DDT version 41 all flavors of DDT support page mapping and address allocation as well as register and physical address manipulation. All of these functions use some variation of the \$U/\$\$U DDT command. In general these functions may be mixed together (for example address allocation and page mapping).

*** Warning ***

The \$U command syntax in DDT is totally different (and mainly incompatible) from previous versions of DDT! The user is MOST strongly urged to carefully read this section on memory mapping and addressing!

4.1 Physical addressing

DDT now has the concept of "physical" addressing in addition to its normal "virtual" addressing. The \$U command instructs DDT to use normal virtual addressing (what it used to do); the \$\$U command instructs DDT to manually track down the honest physical address rather than the virtual address space in which DDT finds itself running. Physical addressing is really applicable only to EDDT or to FILDDT looking at running monitor/memory (TOPS-10 only). User mode DDT (including EDDT running in user mode, MDDT (TOPS-20 only), and VMDDT (TOPS-10 only)) and FILDDT looking at a disk all treat \$U and \$\$U identically. In physical addressing location 0 is not register 0 (i.e., DDT's internal copy of user register 0) but rather physical memory location 0 page 0 bank 0 box 0 (that memory location on the hardware memory bus that responds to all address bits = 0).

When the \$\$U DDT command is issued "physical" locations 0 to 17 become "registers" 0 to 17. For user mode DDT this means locations 0 to 17 become DDT's registers rather than the user's registers (although the user's registers will be properly restored on DDT-exit, \$\$U merely directs DDT not to use the internal "fake" (i.e., user) registers). For FILDDT this means file words 0 to 17 (as mapped by the .EXE directory if used) become locations 0 to 17 (normal for a data file).

Subsequent issuance of the \$U DDT command will redirect locations 0 to 17 to being DDT's internal "fake" registers again, except for FILDDT looking at an data file or doing super I/O to a disk.

Note that for executive mode EDDT to utilize physical addressing the paging hardware must have been enabled PRIOR to DDT-entry. This requirement exists because EDDT, in order to access all of physical memory, needs to map the desired physical address into its own (executive) virtual address space, which it does by fondling the already-stant page maps. For EDDT to provide physical addressing capability without this restriction would require 2 (3 if KL-paging) more memory to be dedicated to EDDT for building temporary page maps, plus support code etc.

For FILDDT to examine/modify physical memory a 7.00 or later release of the TOPS-10 monitor is required; no release of TOPS-20 supports FILDDT'ing physical memory.

3.4.2 Page mapping

All flavors of DDT now support page mapping in both the KI- and the KL-tradition. EDDT in executive mode will dynamically figure out which style of paging is in effect and operate accordingly. All other flavors of DDT (including EDDT running in user mode) will assume the mode of paging used by the operating system for which DDT was assembled - KI-paging for TOPS-10 and KL-paging for TOPS-20. To select KI-paging emulation the flgs10U command is used; to select KL-paging the flg\$11U command is issued; in either case if flg is zero then the paging emulation is disabled, if flg is non-zero then the appropriate paging emulation is enabled.

In executive mode EDDT or FILDDT looking at running monitor/memory space DDT will internally utilize physical addressing in order to provide the user the true mapped virtual address space desired.

3.4.2.1 KI-paging - For KI-paging (TOPS-10 default) the page mapping command for the executive virtual addressing space is [upt<lept\$[0]U where upt is the optional physical memory page number of the user process table (for setting the "per-process" addressing space - exec virtual addresses 340000 through 377777) and ept is the physical memory page number of the executive process table. The user virtual addressing space is selected by the upt\$1U command. The command \$U returns DDT to regular unmapped virtual addressing.

3.4.2.2 KL-paging - For KL-paging (TOPS-20 default) the page mapping command for the executive virtual addressing space is ept\$[0]U where ept is the physical memory page number of the executive process table, or ep\$[0]U where ep is the index into the SPT of the executive process table pointer. To select the user virtual addressing space the command is upt\$1U where upt is the physical memory page number of the user process table, or up\$[0]U where up is the index into the SPT of the user process table pointer. The command \$U returns DDT to regular unmapped virtual addressing.

To map a single section (256K address space) under KL-paging the command is either sec\$2U where sec is the physical memory page number of a KL-paging section map, or sex\$2U where sex is the index into the SPT of the section map.

Basically, under KL-paging, \$0U selects the ept, \$1U selects the upt, and \$2U selects a single section. A single \$ indicates the physical memory page number and two \$'s indicate an SPT index.

3.4.3 Setting the SPT

FILDDT will automatically define the start of the SPT from a disk file (assumed monitor dump) from the symbol "SPT" if it exists (TOPS-20 only). The command spts6U specifies to DDT that the SPT starts at address spt.

3.4.4 Register addressing

The command acs\$5U instructs DDT to use the 20 consecutive locations starting at acs as the registers (DDT maintains an internal copy of the registers so changing "register" 3 will not affect (for example) acs+3). FILDDT, when reading an .EXE file, will automatically load its internal "fake" registers as though the user had typed CRSHACS\$5U if

DPS-10 or BUGACS\$5U if TOPS-20. Note that if physical addressing mode has been entered (the user has issued the \$\$U command) then the internal "fake" registers are ignored; if the user subsequently reenters virtual addressing (via some form of the \$U command) then an acs\$5U command may also have to be re-issued to get the registers back (this does not affect the saving and restoring of the hardware registers in user or executive DDT, only what DDT will use for typing out locations to 17).

The command flgs3U explicitly controls the usage of DDT's internal "fake" registers - if flg is 0 then the "fake" registers are ignored i.e., 0 to 17 are taken from the true current addressing space), if flg is non-zero then addresses 0 to 17 are taken from DDT's internal copies of the registers.

The \$U command, except for FILDDT'ing a data file or doing super I/O to a disk, will return DDT to its internal "fake" registers. The selection of registers is completely independent of any page mapping in effect. Changing virtual address spaces does not change the "registers".

In executive mode DDT only the command ns4U will switch DDT to use and thus display) hardware AC block n (available only for KL-10's and S-10's). The user is warned that 7\$4U on a KL-10 will bring rapid and certain death (the microcode uses AC block 7). On DDT exit DDT will restore the ac block context to the state it was in at DDT entry.

4.5 Address relocation and protection

As an aid to looking at data structures which are formed using pointers as offsets rather than pointers as absolute values DDT version 41 will allow the user to set both a base relocation address to be added to all addresses used in location examining commands and a protection address beyond which the user "virtual" (note the use of "virtual" here as meaning pre-relocated) address is illegal. This is coincidentally exactly analogous to the KA-10 hardware relocation and protection strategy, and in fact could be used as such to "mimic" the KI/KL/KS-10 functionality on a KA-10 in executive mode. The form of this command is bas\$8U where bas is the base virtual address, and prt\$9U where prt is the maximum address the user will be allowed to type in. Note that page mapping and address relocation and protection are independent mechanisms, with address relocation and protection being performed before any mapping is done. The protection address has no effect on the final "physical" address generated by any mapping currently in effect.

4.6 \$U command summary

4.1 \$U/\$\$U commands take the following form:

\$U	Unmapped virtual addressing
\$\$U	Unmapped physical addressing
opts[\$][0]U	Select executive virtual addressing
upts[\$]1U	Select user virtual addressing
secs[\$]2U	Select single section
flgs3U	Select (deselect) internal fake registers
acbs4U	Select hardware ac block
acs\$5U	Load internal fake registers

9.	spts6U	Select base of SPT
10.	bass8U	Set base relocation address
11.	prts9U	Set protection address
12.	flgs10U	Select (deselect) KI-paging
13.	flgs11U	Select (deselect) KL-paging

where:

1. acb := integer ac block number
2. acs := address of 20-word register block
3. bas := base relocation address
4. ept := executive process table page number
5. flg := selection flag, zero to deselect, non-zero to select
6. prt := protection (maximum allowable) address
7. sec := section map page number
8. spt := address of SPT
9. upt := user process table page number

3.4.7 Address checking (Executive EDDT only)
EDDT version 41, when running in executive mode, now is much more extensive in validity-checking memory references. In particular, EDDT will not cause a NXM (page fault) trap to the resident operating system if the user types in an illegal (non-existent or unmapped) address, but rather will simply type its ubiquitous ?<DINK><TAB> error message.

3.4.8 Address breaking
DDT will no longer cause an address break to occur when examining or depositing a location at which an address break condition has been set. This applies only to "user" examines and deposits, an address break set in DDT will still cause an address break to occur.

4.0 Specifying the Start Address

The \$G command now expects a 36-bit address (obviously with bits 0 to 5 off) at which to start the user program. This means that the users of programs such as the TOPS-10 monitor which define symbols like "DEBUG=:<JRST .>" can no longer go either DEBUG\$G or DEBUG\$X at the users whim but must decide on one form or the other (the default obviously being to do nothing - i.e., to settle for the DEBUG\$X form)

5.0 Symbolic expression typein and typeout

DDT version 41 has expanded the range of both symbolic typein and symbolic typeout.

5.1 Symbolic typein

The JSYS opcode (opcode 104) has been added to TOPS-20 DDT, as have all the TOPS-10 UWO's (but not the CALLI's etc.) for debugging programs which run under the compatibility package.

2 Multiply-defined symbol typein

If the user types an ambiguous symbol (a symbol defined two or more places outside of the current local symbol table and not in the current local symbol table) DDT will issue an "M" error message.

3 Selecting no local symbol table

The \$: command issued without an explicit module name to use as the local (or "opened") symbol table will deselect any local symbol table. This is the initial state in which DDT starts.

4 Symbol cache

DDT now has a symbol "cache" of symbols recently used to type out values. This cache is used primarily for typeout; typein will check the symbol cache for a matching symbol from the currently opened or local symbol table, if no match is found the cache is ignored and the global symbol table is used. The symbol cache is "flushed" on the issuance of any \$: command.

5 Symbolic typeout

DDT now goes to great pains to find any possible user-defined symbol (as an OPDEF) to match the expression DDT is trying to type out. The order in which DDT searches for a symbol match in symbolic typeout is as follows:

- Full 36-bit match; OP, AC, I, X, and Y fields (e.g., the TOPS-20 monitor calls such as GTJFN)
- OP, I, X, and Y fields (e.g., the TOPS-10 monitor calls such as FILOP.)
- OP and AC fields (e.g., the TOPS-10 monitor calls such as INCHWL or "instructions" such as HALT)
- OP field only (e.g., user UUD's or "OPDEF XMOVEI [SETMI]")
- DDT's internal hardware opcode table

The order in which DDT searches for a symbol match in symbolic typeout is as follows:

- I/O OP and DEV fields (bits 0 to 12 - e.g., KL-10 APRID or KS-10 RDCSB)
- Regular (non-I/O) OP field (e.g., KS-10 UMOVE)

6 ASCII typeout

DDT version 41 adds the typeout mode commands \$8T and \$9T to typeout 8 bit ASCII or 9 bit ASCII respectively (i.e., pick up 8 or 9 bit bytes and "type" them straight as is - which with current TOPS-10 and TOPS-20 operating systems means as 7-bit ASCII).

7.0 Command files

The \$Y command (TOPS-10 DDT only) has been changed somewhat, both in input and output (logging) functions.

7.1 Command input

If the user does not type a 36-bit expression to be used as a file name (such as \$"FILNAM"\$Y) but just types \$Y by itself then DDT will prompt with "File: ". After the prompt the user can enter a TOPS-10 file specification in the form dev:name.type[directory]/switches where [directory] can of course contain SFD's.

7.1.1 /A switch

The /A switch instructs DDT to abort the command file if a DDT-detected command error occurs (such as reference to an undefined symbol).

7.2 Command output (logging)

When reading a command file (\$Y command) DDT will no longer "log" all output onto device LPT; but rather just type out onto the user terminal.

8.0 Automatic patch insertion

The automatic patch insertion facility (\$< and \$> commands) are basically the same as in version 40 of DDT with only minor differences.

8.1 Patch opening

The user may specify patching either by sym\$< where sym is the name of a symbol (which will be automatically updated at the termination of the patch) or via exp\$< where exp is any 36-bit expression representing the address of the resultant patch. If the later form of the patch command is used no symbol will be updated to the end of the patch.

8.2 Default patching symbol

The list and order of default patching symbols which DDT uses when the user does not supply an explicit patching symbol is now:

1. PAT (TOPS-10 EDDT only)
2. FFF (TOPS-20 EDDT and MDDT only)
3. PAT.. (all flavors)
4. PATCH (all flavors)
5. PAT (all flavors except TOPS-10 EDDT)

8.3 Default patching address

If the user does not supply an explicit patching symbol and DDT is unable to find one of the default patching symbols then the address specified by the right half of location .JBFF (even on TOPS-20) is used. On patch close (\$> command) if the patching address was defaulted to via .JBFF, then both the right half of location .JBFF and the

each half of location .JBSA are updated.

4 Patch closing confusion and restriction

With DDT version 41 it no longer matters how (when) the user types the > command, either immediately after the final word expression, or after a <CR> or <LF> to terminate the final word expression - DDT will never generate a 0 word for free.

There is a very obscure restriction however on the use of the # command in conjunction with the \$> command. If the user is referencing an undefined symbol in the expression for the last word of the patch then that expression must explicitly be terminated in such a fashion as to close the location before terminating the patch. For example, "MOVE T1,BLETC#\$>" is illegal but "MOVE T1,BLETC#crs>" (where "cr" indicates a carriage return) is ok.

0 Breakpoints

The breakpoint logic in DDT version 41 has been extensively revamped in order to support extended addressing. The default number of breakpoints is now 12 (decimal); and can be set (by defining the symbol BP=number of breakpoints) arbitrarily high (within memory space limitations) rather than being limited to 9 or 36 (decimal) depending on which code restriction one chooses to believe.

1 Setting breakpoints

DDT can now set a breakpoint in code running in any section with two restrictions:

- If DDT is currently running in section 0 then breakpoints can only be set in section 0 (see section 3.1.3 above).

- DDT must be mapped in the section containing the code in which breakpoints are to be placed (the logic of this is that since there is no way for DDT to cause unconditional transfer of control to DDT with only 36 bits some portion of the section address space must be devoted to DDT; therefore, given this restriction, one might just as well put all of DDT in that section since it makes for a cleaner and simpler implementation). Note that this does not mean DDT must be running in that section, but merely that DDT must be mapped in that section!

It does not matter into how many different sections the same code is mapped as long as DDT is mapped into the same sections since DDT is "section-independent". For example (taking the TOPS-20 monitor which maps section 0 and section 1 identically) if a breakpoint is set at address 1004567 (or 1,,4567) but the PC was 4567 (or 0,,4567, i.e., in section 0 rather than section 1) when the breakpoint was executed DDT does not care (as long as DDT is mapped in that section, which in the case of the TOPS-20 monitor it is).

The syntax for setting a breakpoint is now `opn<bpt$>nB` where `n` is optional and, if specified, declares the breakpoint number to be assigned to that address; `bpt` is the 36-bit address at which to place a

breakpoint; and opn is an optional 36-bit address to open and display upon execution of the breakpoint. The syntax was changed because two full 30-bit addresses could not be squeezed into two halfwords.

DDT will no longer assign two different breakpoints to the same address, either accidentally or under user control - if the user attempts to set a breakpoint at a location at which a (different) breakpoint is already set, the old breakpoint is cleared first.

9.2 Breakpoint typeout

Upon execution of a breakpoint DDT will now always typeout the user instruction (in instruction format regardless of the permanent typeout mode) at that breakpoint and set "." to the breakpoint address. If, further, opn was specified as in section 9.1 above, then DDT will also display the contents of location opn in the permanently set typeout mode and "." will be updated to opn (with the breakpoint address itself becoming the previous PC sequence and so available via the \$<CR> etc. commands).

9.3 Examining breakpoint locations

The \$nB command continues to be the "address" of breakpoint n's database, but \$nB is no longer equal to \$n-1B+3. The breakpoint database of interest to the user now has the following format:

1. \$nB+0/ If nonzero the address for breakpoint n
2. \$nB+1/ The conditional break instruction (break if skips)
3. \$nB+2/ The proceed count (break on transition to 0)
4. \$nB+3/ If greater than or equal to zero then the address to be displayed

The rest of the breakpoint data base should not be of use to the user.

9.4 Unsolicited breakpoints

DDT version 41 has a new breakpoint facility - the ability to handle unsolicited breakpoints (i.e., breakpoints that DDT did not itself set). If control passes to location \$0BPT+1 (\$0BPT is a global DDT symbol) then DDT will act as if a breakpoint had been set at the address-1 contained in location \$0BPT. The address in \$0BPT must be setup as if the cpu executed a JSR \$0BPT instruction - if in section 0 then flags,,PC otherwise just global 30-bit PC. After "hitting" an unsolicited breakpoint the user can proceed with program execution with the \$P command (all arguments to the \$P command such as proceed count or auto-proceed (\$\$P) are ignored).

Although this facility gives programs the ability to cause breakpoints at any time (thus getting into DDT with the program state carefully preserved) it is intended to be of most use in conjunction with an as-yet-unimplemented monitor command (such as control-D) to "force" a breakpoint on a program without having to control-C/DDT the program. Then the user could simply continue with the program by typing \$P.

0. Single-stepping the program

The \$X DDT command has been significantly modernized (and sped up in general) with version 41 of DDT.

0.1 New opcodes

The ADJSP, DADD, DSUB, DMUL, and DDIV instructions have been added to DDT's \$X table although double- and quad-word integers (for DADD etc.) are still typed out as two or four single words rather than one big multiple precision integer. All of the extended JRST-class instructions are correctly simulated/traced. A user-UUO being executed in a non-zero section is simply XCT'ed and is not traced.

0.2 Byte-manipulation typeout

A rudimentary byte-manipulation-instruction typeout facility was added to DDT version 40 (actually) to display the byte pointer and the contents of the effective address of the byte pointer. The EXTEND-class instructions are not handled.

0.3 Effective address calculation

DDT now always calculates the effective address of the instruction being \$X'ed rather than just blindly "doing it" in order to both prevent DDT from getting an illegal memory reference as well as to make the independent of the section in which the user PC resides (i.e., DDT does not have to be mapped into the user PC section to handle cases although if the user PC is in a non-zero section then DDT must be in a non-zero section). Besides, it's usually faster too!

0.4 KS-10 I/O instruction trace

KS-10 specific I/O instructions which reference the UNIBUS (execute mode only) are not traced, only the contents of the register specified in the AC field are displayed. Since the UNIBUS device registers can be reference-volatile (i.e., merely referencing one can cause it to change - such as the DL-11 data registers) DDT does not typeout the contents of the referenced UNIBUS address. Further, since the effective address of the instruction is not calculated in a standard format (at least as far as DDT is concerned) the effective address itself is not even displayed.

0.5 PC skipping

If the user instruction being \$X'ed skips then DDT will now typeout "<SKIP>" if the PC skips by one location, or "<SKIP n>" if the PC skips by n locations, where n is less than or equal to the DDT assembly parameter SKPMAX (by default 3). If the PC changes more drastically than that (e.g., goes to a smaller address) DDT will type "<JUMP>" instead.

6. ERJMP

(TOPS-20 only) will now handle instructions followed by either an AL or an ERJMP instruction (which is really just a 72-bit instruction).

tion with two effective addresses). If the instruction being executed does not take the error jump then DDT will print "<ERSKP>" after the normal instruction trace to indicate to the user that an ERCAL or ERJMP was just skipped (i.e., the PC incremented by 2 rather than 1) and will not display the ERCAL or ERJMP instruction. If the instruction does take the error jump then the ERCAL or ERJMP instruction will be displayed, if an ERCAL instruction then register 17 will also be displayed, and the PC will be changed to the error address.

DDT will print "<ERSKP>" rather than showing the ERCAL or ERJMP instruction since DDT has no way of telling whether or not the instruction itself caused the skip (as in a SKIP) or if the PC merely "fell through" the ERCAL or ERJMP instruction (as in a successful MOVE).

Users of EDDT and MDDT should be cautioned about \$Xing instructions followed by an ERCAL or ERJMP in non-zero sections - the monitor has a tendency to transfer control to the error address in section 0, which will cause a BUGHLT because DDT (running in executive mode) does non-zero section things thinking it is still in a non-zero section.

10.7 \$X'ing an INIT

DDT will now let the user \$X an INIT (TOPS-10) monitor call. DDT will print out <SKIP 2> if the INIT fails or <SKIP 3> if the INIT succeeds.

10.8 \$X speed up

By building into DDT a table of instructions which can cause the state of the known world to change, and assuming the state of the world does not change if the instruction being \$X'ed is not so marked, the time required to \$X an instruction is cut by roughly a factor of 10. This results in a dramatic performance increase especially for EDDT on KL-10's where waiting for the console front end to switch between secondary and primary protocol is very time-consuming.

10.9 Repetitive \$X'es

The \$\$X command now takes an optional address range. Normally \$\$X will terminate when the user PC inclusively enters the range .+1 to .+ SKPMAX (default value of SKPMAX is 3). The user may specify lwr<upr>\$\$X where lwr is the lower address boundary and upr is the upper address boundary which, if the user PC ever inclusively enters the range so specified, terminates the \$\$X. If only lwr is specified then upr defaults to lwr+SKPMAX. This command is very useful for recovering from having \$X'ed a (for example) PUSHJ instead of having \$\$X'ed the (for example) PUSHJ.

10.10 \$X'ing from instr\$X

If the user \$X'es a return from a subroutine which was entered by doing an instr\$X (for example "PUSHJ P,SUBRTN\$X where SUBRTN has a breakpoint in it) then DDT simply "returns" from the original instr\$X rather than proceeding to \$X the internals of DDT itself. This is a very obscure condition so don't worry too much about it.

0.0 \$\$X status

DDT will now respond to a ? character being typed during an \$\$X sequence by typing "Executing: " followed by the current user "pc" and instruction being executed. Typing any other character terminates the \$\$X immediately.

0.12 \$X PC

The \$. command now acts like the . command only \$. returns the value of the \$X PC (i.e., the address of the next instruction to be \$X'ed). The \$\$\$. command returns the previous \$. value (useful for \$\$,<\$\$X as in section 10.9 above).

1.0 Searches

Most of the differences in how DDT version 41 handles searches are simply bug fixes, not major changes in the logic of searching.

1.1 Non-existent pages

DDT version 41 now simply skips over pages which don't exist in the address space being searched, rather than terminating the search as soon as a hole has been found.

1.2 Missed matches

A bug which caused TOPS-20 DDT to miss many valid matches is fixed in DDT version 41.

1.3 Effective address searches

Since almost all address calculations start with an IFIW basis (with a few exceptions being such things as interrupt vectors and the like on KS-10's or KS-10's), DDT version 41 will assume that each word it examines is an instruction and perform an IFIW effective address calculation. The final result must match in all 30 bits (actually internally DDT will do a full 36-bit compare so the address being searched for had better not contain anything in bits 0 to 5).

1.4 Address limit defaults

With the advent of extended addressing and physical addressing the address limits are defaulted somewhat differently than from previous versions of DDT:

EDDT, MDDT (TOPS-20 only), UDDT, and VMDDT

1. Lower Limit: <current section>,,0
2. Upper Limit: <current section>,,777777

1.5 ILDDT looking at an .EXE file

1. Lower Limit: 0
2. Upper Limit: highest virtual address mapped

3. FILDDT looking at a data file
 1. Lower Limit: 0
 2. Upper Limit: highest word written in file
4. FILDDT looking at disk structure/unit
 1. Lower Limit: 0
 2. Upper Limit: highest word in disk structure/unit
5. FILDDT looking at running monitor
 1. Lower limit: 0
 2. Upper limit: 777777
6. FILDDT looking at physical memory (TOPS-10 only)
 1. Lower Limit: 0
 2. Upper Limit: Highest extant memory address

As with any defaults not all cases will be properly "guessed" by DDT. In particular if the user has mapping or address relocation in effect the virtual address range so produced may have nothing whatsoever in common with the address limit defaults chosen by DDT.

11.5 Search matches

DDT will leave each address matched by its search on the "pc stack" available to \$<CR> etc. commands. When the search is terminated DDT will set "." to the last address searched.

11.6 Searching status

DDT will now respond to a ? character being typed during a search by typing "Searching: " followed by the current location and value being searched. Typing any other character terminates the search immediately.

12.0 Watching

DDT version 41 allows the user to "watch" a location, waiting for it to change. Although primarily useful for FILDDT'ing the running monitor, it is present in all flavors of DDT for completeness. The syntax of the watching command is exp\$V, where exp is the address to be watched. If no explicit address is specified the last location opened by the user will be used.

Upon initial issuance of the \$V command the location is displayed. Thereafter the location is continuously monitored, and will be displayed every time its contents change. In user mode DDTs (and this includes TOPS-20 MDDT as well) the location is checked once a clock tick (approximately 50 to 60 times a second), in exec mode EDDT the location is continuously being monitored - no "pause" is attempted.

DDT will respond to a ? character being typed during an \$V sequence by

g "Watching: " followed by the current location and contents being watched. Typing any other character terminates the \$V immediately.

3.0 Zeroing memory

The algorithm used by DDT previous to version 41 has only limited usefulness in today's modern virtual world (especially on TOPS-20). However, to avoid "breaking" already extant control or MIC files which may use the \$\$Z command it remains unchanged. A new command has been implemented - lwr<upr>exp\$Z where lwr is the lowest (starting) address, upr is the highest (ending) address, and exp is the 36-bit quantity to deposit in each word inclusively bounded by lwr and upr. Both lwr and upr must be specified. If exp is not specified then 0 is used as the default.

Special note: The creation of zeroed pages (which formerly were non-existent) by the \$Z and \$\$Z commands is under the control of the automatic page create flag (i.e., the \$1W and \$\$1W commands - see section 3.3.2).

DDT will now respond to a ? character being typed during an \$Z sequence by typing "Zeroing: " followed by the current location and value being "zeroed". Typing any other character terminates the \$Z immediately.

4.0 Special masks

DDT version 41 (it actually started with DDT version 40) has several new "masks" (for lack of a better name and/or command) of interest to the user. None of these masks are currently displayable (e.g., "\$3M/") in FILDDT although they may be set normally.

4.1 \$0M - Search mask

The operation of the search mask continues unchanged. The search mask may now be referenced by either the \$M (old style) or the \$0M commands. The default value remains 777777777777.

4.2 \$1M - TTY control mask

This mask controls special TTY behavior (primarily TOPS-10 and executive EDDT).

4.2.1 Tab separator display

Bit 17 controls whether DDT will print its usual <TAB> or three spaces for the <TAB> separator. A 0 (the default) selects three spaces, a 1 selects a <TAB>.

4.2.2 Tab simulation

Bit 18 controls tab simulation. A 0 selects literal <TAB> characters (i.e., the terminal handles <TAB>'s directly, a 1 selects space-fill instead. This condition is automatically set for user mode DDT's (in user mode <TAB>s are always output literally) - it is only useful to

manually set tab simulation in exec mode EDDT.

14.2.3 Rubout control

Bit 35 controls rubout (and ^W) operation. A 0 selects "hardcopy" operation (DDT will echo a "\" character and the character being operation (DDT will echo a "\" character and the character being deleted), a 1 will cause rubouts to echo as a backspace, space, backspace sequence. This condition is automatically set for user mode DDT's (if TTY DISPLAY is set then rubouts echo as <BS><SP><BS>) - it is only useful to manually set fancy rubouts in exec mode EDDT.

14.3 \$2M - Offset range

The 36-bit "mask" in this case is really a value, used as the maximum offset allowable for typing addresses in the form symbol+offset. The default offset is 1000 (octal).

14.4 \$3M - Byte mask

This mask is used in conjunction with the \$0 command for typing bytes in a word that are not necessarily evenly spaced. Whenever an \$0 command is issued without an explicit byte size the byte boundaries are determined by one-bits in the byte mask - each one bit in the byte mask marks the low order bit of a byte. Bit 35 is always considered on. The default value is 0 (i.e., one 36-bit byte). For example the DDT command 040100200401\$3M sets the byte mask for typing right-justified 8-bit bytes (preceded by the leading 4-bit byte).

15.0 RADIX=50 symbol typein

Since prehistoric times DDT has supported RADIX=50 symbol typein, but that fact was never documented. The syntax for using a RADIX=50 symbol as an 36-bit item in an expression is sym\$5" where sym is the desired RADIX=50 symbol. For example, to search for all occurrences of the symbol PAT., the DDT commands 37777,,-\$5M (only look at low-order 32 bits) and PAT.,\$5"\$W suffice.

16.0 New DDT runtime information

Several new words have been added to DDT's runtime table describing the state of the machine upon (executive mode only) DDT-entry. These words are all accessible via the DDT command \$I+offset (not available in FILDDT):

1. \$I-01/ APR CONI word
2. \$I+00/ PI CONI word
3. \$I+01/ Mask of PI channels turned off by EDDT
4. \$I+02/ Executive virtual address of EPT
5. \$I+03/ Executive virtual address of UPT
6. \$I+04/ Executive virtual address of CST
7. \$I+05/ Executive virtual address of SPT
8. \$I+06/ Original AC-block word (DATAI PAG) if acbs4U

7. Obsolete commands

The executive mode paper tape facilities (^R, \$J, and \$L DDT commands) are no longer supported. The code is left in the source file for reference purposes but will soon be removed.

3.0 FILDDT startup and commands

FILDDT is a special version of DDT with the facilities for "DDT'ing" address spaces other than its own, such as disk files and in particular .EXE files. FILDDT has existed for years but has always been off in the background as a specialized "tool" for the exclusive use of monitor programmers looking at crash dumps. With DDT version 41 FILDDT is now a general purpose utility for use by the "general public", particularly people who have databases resident in disk files (.REL files for example).

3.1 Symbols

For efficiency considerations FILDDT builds the symbol table(s) it will actually use at runtime in its own address space. Virgin FILDDT has no symbols (the symbol table (if any) for FILDDT in FILDDT.EXE is completely independent of the address space being FILDDT'ed and does not count). There are special commands to instruct FILDDT to extract and build internal-to-FILDDT copies of) symbol tables from .EXE files (see below). Once FILDDT has setup its internal symbol table(s), it may then be SAVED with the internal symbol table(s) for later use by returning to monitor level (with the ^Z FILDDT command) and typing the "SAVE" command.

3.2 TOPS-10

When FILDDT is started it will prompt "File: ". The user may at this time optionally enter a standard TOPS-10 file specification in the form dev:name.type[directory]/switch. At least one function switch is mandatory. SFD's are of course legal in the directory specification.

3.2.1 /D command

The /D command or function switch instructs FILDDT that the file specified is a data file - i.e., do not map the file as an .EXE file and use real file words 0 to 17 for locations 0 to 17.

3.2.2 /F command

The /F command or function switch instructs FILDDT to "DDT this file anyway". It is useful only in conjunction with the /S command or function switch which normally re-prompts for another file specification. Used in conjunction with /S (which implies an .EXE file) FILDDT will save the file from which symbols were extracted as the file to be DDT'ed.

3.2.3 /H command

The /H command or function switch instructs FILDDT to type out a brief help text, abort the current command, and prompt the user for another command.

18.2.4 /J command

The /J command or function switch is applied to a job number rather than a file specification and instructs FILDDT to "DDT" the address space of the job specified. Since FILDDT uses JOBPEK monitor calls to access the specified job's address space the success or failure of any given memory reference is dependent on the job being resident in main memory - if the job is swapped out or if the memory reference is to a page which is paged out the memory reference will fail. This is a privileged command.

18.2.5 /M command

The /M command or function switch instructs FILDDT to "DDT" the currently running monitor and physical memory address space (controlled by use of the \$U and \$\$U commands). This is a privileged command.

18.2.6 /P command

The /P command or function switch instructs FILDDT to enable for writing as well as reading the specified address space. Note that DDT's internal fake registers are always writable.

18.2.7 /S command

The /S command or function switch instructs FILDDT to only extract the symbol table from the file specified, replacing any symbol table FILDDT may already have. Unless overridden by the inclusion of a /F command FILDDT will, after having read the symbol table, again prompt the user for the next FILDDT command.

18.2.8 /U command

The /U command or function switch is applied to a file structure or disk unit only rather than a complete file specification and indicates to FILDDT that the user wants the entire physical address space represented by that file structure or disk unit name independent of any "file structure mapping" normally imposed by the monitor. This is a privileged command.

18.3 TOPS-20

With DDT version 41, FILDDT on TOPS-20 runs in native mode, and in particular, uses the PMAP monitor call for all regular file access. FILDDT will also type a brief message telling what address space is about to be "DDT'ed" before going into DDT mode.

18.3.1 DRIVE command

The format of the DRIVE command is:

DRIVE (FOR PHYSICAL I/O IS ON CHANNEL) c (UNIT) u

The DRIVE command allows examination of the disk unit u on system channel c without regard for whether it is mounted as part of a file structure, or indeed whether it even has the necessary information so that it could be so mounted (as if the HOME blocks were wiped out). If, however, the drive is part of a mounted file structure, FILDDT will type a message indicating the structure to which it belongs. This is a privileged command.

8.2 ENABLE DATA-FILE command

The ENABLE DATA-FORMAT command instructs FILDDT to treat the file as pure data, even if a valid .EXE directory is detected, and in particular to use real file words 0 to 17 as locations 0 to 17.

8.3.3 ENABLE PATCHING command

The ENABLE PATCHING command instructs FILDDT to enable any subsequent / specified address space for patching (writing). This command is ignored when looking at the running monitor since there is no monitor call to "poke" the running monitor.

8.3.4 EXIT command

The EXIT command instructs FILDDT to return to command level. If FILDDT has an internal symbol table (due to a previous LOAD or GET command) then a SAVE command will save FILDDT with the symbols re-loaded.

8.3.5 GET command

The format of the GET command is:

GET (FILE) filespec (optional switches)

The GET command instructs FILDDT to set up the disk file filespec as the address space to be "DDT'ed", as modified by the optional switches from previous ENABLE commands. The available switches are:

8.3.5.1 /DATA - The /DATA switch is equivalent to a previous ENABLE DATA-FILE command.

8.3.5.2 /PATCH - The /PATCH switch is equivalent to a previous ENABLE PATCHING command.

8.3.5.3 /SYMBOL - The /SYMBOL switch instructs FILDDT to extract symbols from the specified .EXE file before "DDT'ing" the file, discarding any symbols that FILDDT may already have. This switch is legal only with .EXE files.

8.3.6 HELP command

The HELP command instructs FILDDT to type out a short summary of the available FILDDT commands.

8.3.7 LOAD command

The format of the LOAD command is:

LOAD (SYMBOLS FROM) filespec

The LOAD command instructs FILDDT to extract symbols from the disk file filespec, which must be an .EXE file, then to return to FILDDT command level. This command is legal only for .EXE files.

8.3.8 PEEK command

The PEEK command instructs FILDDT to use the currently running monitor to set the address space to be "DDT'ed". The address space so available is currently limited to monitor executive virtual addresses 0 to 777777, since the PEEK monitor call will only accept 18-bit address arguments for executive virtual addresses. Physical memory addressing is not

available. This is a privileged command.

18.3.9 STRUCTURE command

The format of the STRUCTURE command is:

STRUCTURE (FOR PHYSICAL I/O IS) str:

The STRUCTURE command instructs FILDDT to use as the address space to be "DDT'ed" the entire disk file structure str independent of any "file structure mapping" normally imposed by the monitor. This is a privileged command.

18.4 Defaults

Following is a list of the various defaults supplied by FILDDT:

1. DSK: is the default file device unless super I/O is specified (which requires an explicit file structure or disk unit name).
2. .EXE is the default file type or extension unless either a data file or super I/O is specified, in which case there is no default file type or extension.
3. The default directory is the user's default directory.
4. The specified address space is read-only.
5. If "DDT'ing" an .EXE file and FILDDT does not already have a symbol table, extract the symbol table (if any) from the .EXE file first.
6. If "DDT'ing" an .EXE file and the symbol CRSHAC (if TOPS-10) or BUGACS (if TOPS-20) exists, give a "free" CRSHAC\$5U or BUGACS\$5U command. If the CRSHAC/BUGACS symbol does not exist then use file words 0 to 17 (if any) as mapped by the .EXE directory for locations 0 to 17. For TOPS-20 only, if the symbol SPT exists then also give a free SPT\$6U command as well.

18.5 Other FILDDT-specific commands

Following are the commands which are unique (or different) to FILDDT.

18.5.1 ^E command

The ^E command instructs FILDDT to exit the current address space and prompt the user for a new address space. The ^E command is equivalent to a ^Z, START command sequence.

18.5.2 ^Z command

The ^Z command instructs FILDDT to exit to monitor level after having written out any changes to the current file (if any). It is most important that the user exit only via ^Z (or ^E which does an implicit ^Z) in order to guarantee the integrity of the file data (if any) - a ^C can leave a file in an indeterminate state (some changes written out to the disk and some not).

18.5.3 I/O errors

Should FILDDT incur an I/O error reading or writing a disk file, a warning message will be issued but FILDDT will otherwise ignore the error. This is to allow the user the ability to manually fix a file with bad data by rewriting the data correctly (hoping the rewriting

Position clears the error condition - if the physical disk surface itself is at fault, then it is probably hopeless).

TOPS20 Coding Standards

SUBROUTINE CALLING - JSYS

Monitor-call JSYSes may be used in user or monitor code. All ACs are preserved over a JSYS call unless an explicit statement to the contrary appears in the JSYS description. ACs are changed over a JSYS call only when values are to be returned to the caller.

The JSYS name shall appear as the opcode in the statement which performs the call. The JSYS mnemonic includes the instruction field, so no other fields are supplied by the user.

Unimplemented JSYSes will invoke the illegal instruction sequence (with error code ILINS2). Defined and implemented JSYSes will return to caller +1 on success, or will invoke the illegal instruction sequence on failure. The illegal instruction sequence recognizes an ERJMP or ERCAL following the failing JSYS and causes the appropriate action. If the following instruction is not an ERJMP or ERCAL, an illegal instruction interrupt is requested which will be handled by the executing fork if enabled, or otherwise cause a forced fork termination. See paragraph below on JSYS returns for proper indication of JSYS failure.

All constant values, bits, and fields of JSYS arguments shall have mnemonics defined according to the rules in MONSYM. The JSYS code itself shall use these symbols for loading arguments, testing bits, etc.

When writing code to implement a JSYS, the following conventions shall be observed:

TOPS20 Coding Standards

1. The entry point of the JSYS is defined by a global tag which consists of a DOT concatenated with the symbolic name of the JSYS, e.g. .GTJFN::.
2. The first statement of the JSYS code shall be MCENT (Monitor Context ENTry). This establishes the normal JSYS context for a "slow" JSYS. At this writing, MCENT is a null macro and the JSYS entry procedure is invoked automatically. The use of MCENT is required so that this implementation may be changed in the future if necessary.
3. All caller ACs are automatically preserved by the entry and exit procedures. Therefore JSYS routines are specifically required NOT to save and restore the ACs. The contents of the caller's ACs 1-4 are copied into the callee's ACs. No callee ACs are copied back to the caller's AC block on return however; one of the "previous context" instructions* must be used to return any values to the caller. E.g.,

```
UMOVEM T1,T1      ;store monitor T1 into user T1
```

A previous context instruction may also be used at any time to fetch the original contents of the caller's ACs unless they have been explicitly changed by a previous context store operation. E.g.,

```
UMOVE T2,T1       ;load user T1 into monitor T2
```

- - - - -
* - UMOVE, UMOVEM, XCTU [instruction], etc.

TOPS20 Coding Standards

4. Return from JSYS code should be effected by the statement

MRETNG ;Monitor RETURN Good

This transfers to the JSYS exit sequence (returning caller +1) and should be used to indicate successful completion of the JSYS. If the JSYS could not be completed successfully, the following statement should be used:

ITERR errcod ;causes an Instruction Trap
;ERROR, leaves
;the error code in LSTERR

Certain other statements are defined which effect JSYS returns according to a previous convention. They are:

RETERR errcod ;RETURN ERROR, return
;caller +1 with error code
;left in AC1 and LSTERR

EMRETN errcod ;Error Monitor RETURN, return
;caller +1 with error code left
;in LSTERR

These should not be used in new JSYS code but may be needed if existing JSYSes are modified.

All error returns shall include an error code (mnemonic) which shall be defined in MONSYM.MAC. If the appropriate error code has already been loaded into AC1, then the errcod field may be omitted from the above and the contents of AC1 will be taken as the error code. No JSYS shall return other than +1 or instruction trap, therefore no occurrence of AOS 0(P) should ever be required in JSYS code.

When invoking a JSYS error return, it is not necessary to pop temporary quantities from the stack. The successful return however, should be given only when the stack is properly cleared.

TOPS20 Coding Standards

SUBROUTINE CALLING - Internal monitor routines

The allocation of ACs for all inter- and intra-module subroutine calls shall be:

ACs 1,2,3,4 -- General temporary, may be clobbered by subroutine.

ACs 6, 5-15 -- Preserved, not changed by subroutine (or saved and restored if necessary).

AC 16 -- Temporary, used by JSYS call/return procedure and reserved for use by other call/return procedures.

AC 17 -- Global stack pointer

Call and return shall be effected by PUSHJ P, and POPJ P, respectively. A set of assembler mnemonics has been defined for subroutine mechanics as follows:

'CALL' (= PUSHJ P,) shall be used to call subroutines, e.g. CALL SUBR.

'RET' (= POPJ P,) shall be used to return +1 from subroutines.

'RETSKP' shall be used to return +2 from subroutines. RETSKP is equivalent to:

```
JRST [ AOS 3(P)
      RET]
```

'RETBAD errcod' shall be used to return +1 with an error code from a subroutine. The error code field is optional as with JSYS error returns above. RETBAD is equivalent to:

```
JRST [ MOVEI A,ERRCOD
      RET]
```


TOPS20 Coding Standards

'CALLRET' may be used to call a subroutine and return immediately thereafter. It is an abbreviation for

```
CALL SUBR
RET
or
CALL SUBR
RET
RETSKP
```

Note that CALLRET is not guaranteed to be a single instruction; therefore it may not be skipped over. The other returns above are guaranteed to be single instructions.

These mnemonics are used to emphasize the FUNCTION being performed (calling, returning) rather than the mechanics of the function (pushing, jumping, etc.). Also, these mnemonics could continue to be used even if a more general calling standard were adopted at some time in the future.

Return may also be effected by transferring control to the global tag R or RSKP, e.g.

```
JUMPE A,R           ;equivalent to JUMPE A,[RET]
JUMPN A,RSKP        ;equivalent to JUMPN A,[RETSKP]
```

The general temporaries shall be used for passing arguments to subroutines and returning values. AC1 shall be used for a single argument routine, ACs 1 and 2 for a two-argument routine, etc.

A routine defined to return caller +2 (skip) on success and caller +1 (noskip) on failure is acceptable. Returns greater than caller +2 are not permitted.

TOPS20 Coding Standards

AC DEFINITIONS

The following mnemonics have been chosen to be consistent with the AC use conventions above. The preserved ACs are divided into three groups, F (1 AC) intended for Flags, and Q1-Q3 and P1-P6 intended for general use. The ACs within each group are consecutive.

0 - F	10 - P1
1 - T1	11 - P2
2 - T2	12 - P3
3 - T3	13 - P4
4 - T4	14 - P5
5 - Q1	15 - P6
6 - Q2	16 - CX
7 - Q3	17 - P

The programmer should assume that each group (Tn, Qn, Pn,) is in ascending order, e.g. that $T2 = T1 + 1$, but that the specific assignment of numbers may change. Explicit numeric offsets from AC symbols (e.g. $T1 + 1$) should NEVER be used. Instructions which use more than one AC (e.g. DIV, JFFO) must be given an AC operand such that the other AC(s) implicitly affected are in the same group. E.g. T3 (and T4) is OK for IDIV because $T3 + 1 = T4$, but Q3 is not because $Q3 + 1 = ??$.

There are several facilities in the monitor to save and automatically restore ACs. Each of these will save all of the indicated ACs on the stack at the point of execution and will place a dummy return on the stack which causes these ACs to be restored automatically when the current routine returns. Use of these facilities eliminates the need for matching PUSH/POP pairs at the entry at exits of routines and eliminates the bugs which often arise from an unmatched PUSH or POP. The available macros are:

SAVEQ - saves ACs Q1-Q3

TOPS20 Coding Standards

SAVEP - saves ACs P1-P6

SAVEPQ - saves ACs Q1-Q3 and P1-P6

SAVET - saves ACs T1-T4

Defining a different mnemonic for a preserved AC may be of value when the AC is used for a specific function by a large body of code. However, it offers the possibility of confusion because two different symbols may refer to the same AC unbeknownst to the programmer. In smaller programs, use of certain ACs can be restricted to specific functions, and a global definition is appropriate. A timesharing monitor however, is too large to accomodate all of the possible dedicated ACs.

Therefore, when a specific function-oriented AC definition is made, it shall be explicitly decided which modules shall use the definition. Within these modules, the usual name for the AC must be purged so that there is no possibility of using two different symbols for the same AC.

Only preserved ACs may be used for special definitions. Parameters to subroutines may be passed in functionally defined ACs in the following cases:

1. On an intra-module call where the contents of the AC are appropriate to its function definition.
2. On an inter-module call where the same definition exists in both modules and the AC is being used for its intended function.

TOPS20 Coding Standards

A parameter may NOT be passed in a preserved AC unless both caller and callee know it by the same name, and that name must be a specific one related to the function which the AC is performing.

The procedure for declaring a functionally defined AC is:

```
DEFAC NEWAC,OLDAC
```

This must be done at the beginning of an assembly, and it defines NEWAC to be equal to OLDAC. OLDAC must be the mnemonic for one of the regular preserved ACs, and this mnemonic will be purged and therefore unavailable for use in the current assembly.

An AC with a special definition should not be used for other purposes; e.g. "JFN" should not be used to hold some quantity other than a JFN merely because it happens to be available.

SUBROUTINE DOCUMENTATION

The following is a suggested format for documenting the calling sequence of a JSYS or subroutine. A description of this sort should appear at the beginning of every subroutine, no matter how short.

```
;name of subroutine - function of subroutine, etc.  
; T1/ description of first argument  
; T2/ description of second argument  
; ...  
; CALL NAME or JSYSNAME  
; RETURN +1: conditions giving this return,  
; T1/ value(s) returned  
; RETURN +2: conditions and values as above.
```

1. The arguments, if any, should be documented as the contents of registers and/or variables as shown. MONSYM mnemonics should be used when available; e.g. at JSYS entry points.

TOPS20 Coding Standards

2. The actual instruction to do the call should be shown. This will be "CALL subname" in the case of internal subroutines, and the single-word JSYS name in the case of a JSYS entry point.
3. The return(s) should be noted as "shown; "ALWAYS" or "NEVER" may be used as the condition where appropriate; the +2 return need not be shown if it does not exist; values returned should be described in the same form as arguments.

Examples:

```
;SIN - COMPUTES SINE OF AN ANGLE
; T1/ ANGLE IN RADIANS, FLOATING POINT
;      CALL SIN
; RETURN +1: FAILURE, UNNORMALIZED NUMBER OR OUT OF
RANGE
; RETURN +2: SUCCESS, T1/ VALUE, FLOATING POINT
SIN:: ..
```

- - - - -

```
;GJINF - GET JOB INFORMATION JSYS
;      GJINF
; RETURN +1: ALWAYS,
; T1/ LOGGED-IN DIRECTORY NUMBER
; T2/ CONNECTED DIRECTORY NUMBER
; T3/ JOB NUMBER
; T4/ TERMINAL NUMBER OR -1 IF DETACHED
.GJINF:: ..
```

MULTI-LINE LITERALS

The use of multi-line literals is encouraged as a technique for making code more readable and easier to follow. The following additional rules apply:

TOPS20 Coding Standards

1. The opening bracket for a multi-line literal should occur in the position that the first character of the address field would have appeared if the instruction had an ordinary address, e.g.

```
SKIPGE FOO
JRST [    ] ; COMMENT
```

2. The first and all following instructions within the literal shall begin at the second tabstop, e.g.

```
JRST [MOVE A,MUMBLE] ; COMMENT
JRST FIB ; COMMENT
```

The tab between the open bracket and the first opcode may be omitted if the line position is already at or beyond the second tab stop, e.g.

```
JUMPGE A,[MOVE A,MUMBLE
```

3. The closing bracket shall follow the last field of the last instruction (as above), and shall be before the comment on the same line.
4. Nesting of multi-line literals to a depth greater than one is discouraged because of awkward formatting problems.
5. Tags may not appear in multi-line literals.
6. No hard and fast rules can be given as to when to use or not use multi-line literals. However, a literal longer than about 10 lines becomes suspect.
7. Use of ".+1" is legal in a literal to return to the main sequence.

TOPS20 Coding Standards

FLOW OF CONTROL - BRANCH CONVENTIONS

In general, jumps should be to tags forward (down the page) from the point of branch except for loops. Tops of loops should be identified by comment.

The expressions ".+1" and ".-1" are the only legal uses of "." (this location). All other potential uses should be avoided in favor of an explicitly defined tag.

"Global" jumps should be avoided altogether. Higher-level languages do not permit them and with good reason. The only exceptions are jumps to well defined and published exit sequences, e.g. R, RSKP (see subroutine conventions, above).

NUMBERS

In general, there should be no occasion to use a literal number in in-line code. All parameters, bit definitions, CONO/CONI codes, etc. should be defined mnemonically at appropriate places. It is much easier to err in the direction of too little use of mnemonics rather than too much; therefore, when in the slightest doubt, define a mnemonic.

TOPS20 Coding Standards

TOPS20 Coding Standards

TOPS20 Coding Standards

TOPS20 Coding Standards

TOPS20 Coding Standards

TOPS20 Coding Standards

Appendix A

LIVING IN AN IMPERFECT WORLD

Much of the present TOPS20 code was written before the existence of this standard and therefore does not conform to it. A great deal of systematic editing has already been done to improve conformance, but obvious irregularities exist. In general, new code being added should conform exactly to this standard even if being integrated with old code. The following are some specific problems which may arise and the recommended solutions:

1. AC Mnemonics

Some code uses absolute numeric ACs. If new code is being integrated into a sequence which uses numeric ACs, it is desirable that the existing code be edited to use the standard mnemonics, particularly for the preserved ACs. If the programmer cannot take the time to do that, then the mnemonics T1-T4 should be used for ACs 1-4, and other ACs should be referenced in the same way as is done by the existing code.

Some code uses mnemonics A,B,C,D for the temporary ACs. These same mnemonics should be used for new code integrated into this existing code, or all references can be edited to use the standard mnemonics.

You may write some code using the standard mnemonics for preserved ACs and then discover that the module into which you wish to put this code has redefined some of these ACs. The solution is one or a combination of the following:

1. Move the new code to a module which does not redefine the preserved ACs.
2. Use different preserved ACs -- ones which have not been redefined. (Note it is not acceptable to use an AC with a special definition for other than its special purpose.)

Clearly, code which needs some of the special definitions must be placed in a module which has these ACs defined and must therefore use only the other preserved ACs.

TOPS20 Coding Standards

Standard 1.0000

Note that a value which usually resides in a special AC need not ALWAYS reside there. For example, if code in JSYSF needs to call a routine in PAGEM and pass a JFN index as an argument, the JFN should be loaded into T1-T4 for the call since PAGEM does not have JFN defined and cannot accept an argument in it.

2. Stack Handling

Use of the several stack variable facilities defined in MACSYM is recommended. Some old code uses explicit PUSH and POP and references of the form --n(P) however, and when anything more than trivial modifications must be made to such code, it is most strongly recommended that the code be edited to use STKVAR or TRVAR. Failing that, references must be consistent with the existing code.

Standard 2.0000

