XVM/DOS
SYSTEM MANUAL

DEC-XV-ODSAA-A-D

XVM Systems

digital

XVM/DOS
SYSTEM MANUAL

DEC-XV-ODSAA-A-D

digital equipment corporation · maynard. massachusetts

First Printing, January 1976

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

The following are trademarks of Digital Equipment Corporation:

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

CONTENTS

CONTENTS (Cont'd)

CONTENTS (Cont'd)

FIGURES

TABLES

TABLES (Cont'd)

LIST OF ALL XVM MANUALS

The following is a list of all XVM manuals and their DEC numbers, including the latest version available. Within this manual, other XVM manuals are referenced by title only. Refer to this list for the DEC numbers of these referenced manuals.

BOSS XVM USER'S MANUAL	DEC-XV-OBUAA-A-D
CHAIN XVM/EXECUTE XVM UTILITY MANUAL	DEC-XV-UCHNA-A-D
DDT XVM UTILITY MANUAL	DEC-XV-UDDTA-A-D
EDIT/EDITVP/EDITVT XVM UTILITY MANUAL	DEC-XV-UETUA-A-D
8TRAN XVM UTILITY MANUAL	DEC-XV-UTRNA-A-D
FOCAL XVM LANGUAGE MANUAL	DEC-XV-LFLGA-A-D
FORTRAN IV XVM LANGUAGE MANUAL	DEC-XV-LF4MA-A-D
FORTRAN IV XVM OPERATING ENVIRONMENT MANUAL	DEC-XV-LF4EA-A-D
LINKING LOADER XVM UTILITY MANUAL	DEC-XV-ULLUA-A-D
MAC11 XVM ASSEMBLER LANGUAGE MANUAL	DEC-XV-LMLAA-A-D
MACRO XVM ASSEMBLER LANGUAGE MANUAL	DEC-XV-LMALA-A-D
MTDUMP XVM UTILITY MANUAL	DEC-XV-UMTUA-A-D
PATCH XVM UTILITY MANUAL	DEC-XV-UPUMA-A-D
PIP XVM UTILITY MANUAL	DEC-XV-UPPUA-A-D
SGEN XVM UTILITY MANUAL	DEC-XV-USUTA-A-D
SRCCOM XVM UTILITY MANUAL	DEC-XV-USRCA-A-D
UPDATE XVM UTILITY MANUAL	DEC-XV-UUPDA-A-D
VP15A XVM GRAPHICS SOFTWARE MANUAL	DEC-XV-GVPAA-A-D
VT15 XVM GRAPHICS SOFTWARE MANUAL	DEC-XV-GVTAA-A-D
XVM/DOS KEYBOARD COMMAND GUIDE	DEC-XV-ODKBA-A-D
XVM/DOS READER'S GUIDE AND MASTER INDEX	DEC-XV-ODGIA-A-D
XVM/DOS SYSTEM MANUAL	DEC-XV-ODSAA-A-D
XVM/DOS USERS MANUAL	DEC-XV-ODMAA-A-D
XVM/DOS V1A SYSTEM INSTALLATION GUIDE	DEC-XV-ODSIA-A-D
XVM/RSX SYSTEM MANUAL	DEC-XV-IRSMA-A-D
XVM UNICHANNEL SOFTWARE MANUAL	DEC-XV-XUSMA-A-D

PREFACE

This manual is written for customer systems programmers, DEC Software Specialists, and internal maintenance programmers. Readers must be familiar with the XVM/DOS Users Manual. In addition, Chapter 9 requires familiarity with the BOSS XVM User's Manual.

CHAPTER 1
INTRODUCTION

XVM/DOS is a disk-based, single-user interactive operating system for the XVM and PDP-15 computers. This manual describes the internal operation of principle components of XVM/DOS. In addition it describes certain programming techniques of interest to system programmers.

Chapter 2 describes miscellaneous advanced system programming features. Chapter 8 describes how to write an I/O driver. These two chapters contain the most commonly used information in this manual. The rest of the manual describes the internal operation of various portions of XVM/DOS.

Chapters 3, 4, and 5 describe the resident monitor, the nonresident monitor, and the system loader respectively. Chapter 6 describes various blocks, tables, and other data structures used within XVM/DOS. Chapter 7 describes the formats of DECtape, magtape, and disk file structures. Chapter 9 describes BOSS mode and non-BOSS Batch mode. The information in these chapters is intended primarily for informational purposes. It would be of immediate use only if the user were to modify XVM/DOS.

CHAPTER 2
ADVANCED PROGRAMMING FEATURES

This chapter describes those XVM/DOS features of interest to system programmers. These features are difficult to use, and may negatively affect system performance and integrity if used incorrectly. For this reason, and also since they are used infrequently, these features are described here rather than in the XVM/DOS Users Manual.

Many of the features described in this chapter make use of the system communication table .SCOM. All .SCOM locations have mnemonics defined for them, and should only be referenced using these mnemonics. The programming examples included in this chapter follow this convention, first equating the mnemonic to the appropriate value and then using the mnemonic whenever the .SCOM location is accessed. This convention makes programs more readable and future alterations of .SCOM easier.

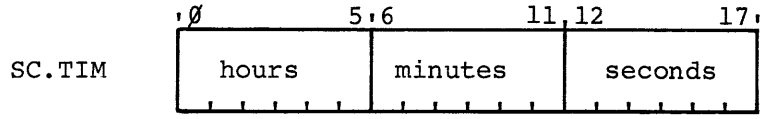
As stated previously, many of the features described herein may negatively affect system integrity if used incorrectly. The example coding sequences provided herein are especially sensitive in this regard. Many of the coding sequences contain timing and order dependencies, especially in their interactions with interrupt routines. For this reason the coding sequences presented here should be followed exactly, especially with respect to the order in which operations are performed. The programmer must be especially cautious when, as is normally the case, he does not completely understand the internal operation of the particular feature he is using.

2.1 TIMING FEATURES

XVM/DOS provides several timing mechanisms for user programs. .TIMER is capable of scheduling routines for execution at a specified time. (See Section 2.2.) Job time limits are provided for both BOSS and non-BOSS modes of operations. BOSS time limits are part of the $JOB card (described in BOSS XVM User's Manual), whereas non-BOSS time limits are accomplished by the TIMEST command (described in XVM/DOS User's Manual). All other timing features are described below.

2.1.1 Time of Day (SC.TIM)

The current time of day is maintained in .SCOM location SC.TIM. Its format is:



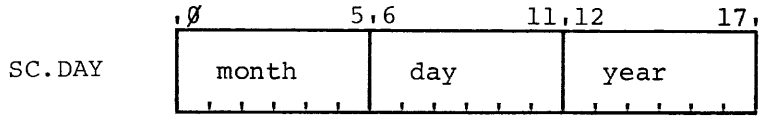
Time is kept according to a 24 hour clock.

SC.TIM should always be accessed using the mnemonic SC.TIM, after first equating SC.TIM to 150_8. SC.TIM is location 150 in memory.

The time of day is set to 00:00:00 (midnight) following a cold-start. The time may subsequently be reset with the TIME command (described in XVM/DOS User's Manual).

2.1.2 Calendar Date (SC.DAY)

The current date is maintained in .SCOM location SC.DAY. Its format is:



The year is stored as years since 1970; thus, zero corresponds to 1970. The month is stored as the calendar month; one corresponds to January. The day is merely the date within the month. Note that neither the month nor the day field may legitimately be zero, except in the case described below.

SC.DAY should always be accessed using the mnemonic SC.DAY, after first equating SC.DAY to 147_8. SC.DAY is location 147 in memory.

Following a cold-start, SC.DAY is cleared to zero, indicating the current date is unknown. If the current date is unknown, the non-resident monitor will, upon being invoked, ask the user for the date. The date is also set unknown (SC.DAY cleared) whenever midnight is reached twice in succession without the non-resident monitor having been invoked in between.

2.1.3 Interval Timing (SC.ETT)

.SCOM location SC.ETT is provided for interval timing by user programs. It is incremented either 50 or 60 times per second, depending upon the line frequency. This location is not used by any system program. The user may modify this location freely.

SC.ETT should always be accessed using the mnemonic SC.ETT, after first equating SC.ETT to 151_8. SC.ETT is location 151 in memory.

2.2 TIMER INTERRUPT ROUTINES

The .TIMER CAL allows the user to schedule .TIMER interrupt routines for execution at a specified time. This section describes the coding conventions which must be followed when writing a .TIMER interrupt routine. Use of the .TIMER CAL itself and its associated system macro are described in the XVM/DOS User's Manual.

The XVM/DOS monitor invokes .TIMER interrupt routines by simulating a JMS instruction. The high three bits (the operating mode bits) of the return address may differ, however, from the bits which would be stored by an actual JMS. When the user program is in XVM mode (17 bit addressing mode) these high three bits will always be zero. When the user program is not in XVM mode, the contents of these bits are undefined.

When the .TIMER interrupt routine is entered, the state of the user program AC, Link, Bank/Page mode status, and XVM mode status are preserved so a return to the interrupted code is possible. However, modifying one of the registers or states and returning (from the .TIMER routine) will not always alter the corresponding register or state for the interrupted user program.

There are three forms which a .TIMER interrupt routine might take. These forms are:

1. Routines which do not return to the interrupted code. This form should be avoided whenever API might not be used.
2. Routines which return to the interrupted code without scheduling a new .TIMER interrupt.
3. Routines which return to the interrupted code after scheduling a new .TIMER interrupt.

The three sections which follow describe the coding conventions appropriate to each form of .TIMER interrupt routine. Following this is a section covering the hazards of using .TIMER without also using API.

2.2.1 Routines Which Do Not Return

A .TIMER interrupt routine which will not return to the interrupted code should be coded as follows:

```
timint  Ø
        JMP label
```

No state information need be preserved since the interrupted code will not be resumed. However, any I/O in progress may need to be aborted.

WARNING

System integrity cannot be guaranteed if this form is used without API. In particular, disk file structures may be corrupted. It is strongly recommended that all .TIMER interrupt routines return to the interrupted code (possibly after setting appropriate flags) if the program might ever be used without API.

2.2.2 Routines Which Return

A .TIMER interrupt routine which will return to the interrupted code should be coded as follows:

```
timint  Ø
        DAC saveac      / Save AC
        RAR             / and Link.
        DAC savlnk
         .
         .
         .
        LAC savlnk      / Restore Link
        RAL
        LAC saveac      / and AC
        JMP* timint     / and return.
```

The user is responsible for saving and restoring the AC and Link. The user must also preserve all other registers and states, including the index and limit registers, EAE registers, auto-increment registers, FPP registers, and XVM mode (17 bit addressing mode) status. The .TIMER interrupt routine must not call any non-re-entrant user routines nor use any CAL's.

These restrictions may be relaxed somewhat if the .TIMER interrupt routine will only be used when API is also in use. If this assumption holds, the interrupt routine may issue CAL's freely. However, it is necessary to ensure that the mainstream program and the interrupt routine do not interfere with each other's .DAT slots. If API will be used, the interrupt routine may also make free use of any registers which are not used by the mainstream program. See Section 2.2.4 for a more detailed discussion of .TIMER restrictions when API might not be used.

2.2.3 Routines Which Re-Schedule and Return

A .TIMER interrupt routine which will schedule a new .TIMER interrupt and then return to the interrupted code should be coded as follows:

```
.INH=705522
.ENB=705521
SC.TMA=161
SC.TMT=160
timint  Ø
        DAC saveac      / Save AC ...
        RAR             / ... and Link.
        DAC savlnk
         .
         .
         .
        LAC (address)   / Put address of next .TIMER
        DAC* (SC.TMA)   / Interrupt routine in SC.TMA.
        .INH            /// Disable interrupts to avoid
        IOF             /// Possible reentrancy problems.
```

```
LAC (-ticks)    /// Put the two's complement
DAC* (SC.TMT)   /// of ticks until next inter-
                /// rupt in SC.TMT.
LAC savlnk      /// Restore Link ...
RAL             ///
LAC saveac      /// ... and AC.
ION             /// Enable interrupts ...
.ENB            ///
JMP* timint     /// ... after return.
```

The user is responsible for saving and restoring the AC and Link plus all other registers and states. See Section 2.2.2 for a more detailed discussion.

Interrupts must be disabled when the next .TIMER interrupt routine is scheduled to avoid possible re-entrancy problems. In the most general case, a .INH and IOF in combination are required to disable interrupts and an ION and .ENB in combination to enable interrupts. The IOF/ION pair is only required on machines which do not have API hardware installed[1]; whether or not API will be used does not matter. Similarly, the .INH/.ENB pair is only needed if API will be used. If the routine will always be run without API they may be deleted.

Usually the same .TIMER interrupt routine will be used for the next interrupt. SC.TMA need not be updated if this is the case. SC.TMT must always be updated because it has been cleared to deactivate .TIMER.

2.2.4 Using .TIMER Without API

When .TIMER is used with a system which is also using API, there are essentially no restrictions on its use. There are certain guidelines which the programmer should follow, such as preserving registers and the like, but the programmer is free to disobey these guidelines if he so chooses. Doing so will almost always result in a malfunctioning user program, but should not affect system integrity. The reason for this is as follows.

When API is in use, the XVM/DOS monitor can detect, at any instant in time, whether or not either a CAL routine or an interrupt routine is active. When the .TIMER time interval expires and the .TIMER interrupt occurs, the monitor checks to see if either a CAL routine or an interrupt routine is active. If so, the actual occurrence of the .TIMER interrupt

[1] The re-entrancy ECO package (ECO #KP15-49 through KP15-52) must also be installed. XVM/DOS cannot use API unless this ECO package is present.

is delayed until all such routines terminate and the monitor is quiescent. The net effect of this is that the interrupted program is always the user program. Whatever actions the .TIMER interrupt routine performs cannot disturb the integrity of the various monitor routines.

All this no longer applies when API is not in use. The monitor is unable to determine whether or not a CAL routine or an interrupt routine is active. Therefore, when the .TIMER time interval expires, the interrupt occurs immediately. It is possible that some portion of the monitor will be interrupted, which implies that the .TIMER interrupt routine must follow certain programming conventions to preserve system integrity. System integrity may be violated if these conventions are not followed, possibly causing disk file structure corruption and other undesirable results.

The conventions necessary to preserve system integrity are described below. All of these are conventions which must be followed by .TIMER interrupt routines.

1. The .TIMER interrupt routine must return to the interrupted code.
2. All registers must be preserved. This specifically includes the AC, Link, MQ, SC, XR, LR, and auto-increment registers.
3. The routine must not issue any CAL's nor modify any buffers or other locations being used by I/O handlers.

These conventions ensure that the monitor routines will be able to complete successfully.

While the user is expected to follow these conventions whenever possible, there are some occasions where this cannot be done. The most common of these will be .TIMER interrupt routines which do not return to the interrupted code. On such occasions the user is advised to terminate I/O using the procedures outlined in Section 2.4. However, we must stress that successful operation cannot be guaranteed and must depend upon luck and the user's own knowledge. Any problems which occur will be timing dependent, and thus probably inconsistent and non-reproducible.

There is one other oddity which occurs when .TIMER is used without API. When API is in use, the return address stored at the .TIMER interrupt routine's entry point is an address within the user program. Thus, this address may be used to gather statistics about how much time is spent in various portions of the program. When API is not in use, on the other hand, this does not apply. Without API the return address is actually a constant, and thus useless for this purpose.

2.2.5 Deactivating .TIMER Interrupts

A pending .TIMER interrupt request may be deactivated by issuing the following CAL:

```
.TIMER Ø,Ø
```

Within an interrupt routine the same effect should be achieved with:

```
SC.TMT=16Ø
        DZM* (SC.TMT)
```

2.3 CONTROL CHARACTER ROUTINES

XVM/DOS provides several control characters which interrupt the current program and transfer control to a specified routine. These control characters are ↑C, ↑P, ↑S, ↑T, and ↑Q.[1] All of these have standardized meanings, described in the XVM/DOS User's Manual. The XVM/DOS monitor provides a default routine for each of the control characters (except ↑P) so as to implement the standardized meaning. A default routine cannot be supplied for ↑P, since its standardized meaning is to restart the current program from the beginning. The ↑P control character routine must be specified by the current program, as only the program knows how to re-initialize itself.

Although default routines are supplied, the programmer is provided the capability to change them. The only exception is ↑Q, which is dedicated

[1] ↑R is recognized and treated specially by the IOPS error processor. However, it does not perform the same function as the control characters described here, i.e., interrupting the program and transferring control to a control character routine.

to its task. The remaining four control character routines (↑C, ↑P, ↑S, and ↑T) may be respecified; however, the standardized meanings should be adhered to whenever possible so as to retain a reasonably simple and consistent human interface.

There are two methods of specifying control routines, summarized in Table 2-1. The ↑S routine is straightforward, as the routine address is in .SCOM location SC.UST. The other three control character routines are specified by issuing a .INIT to TTA., the console terminal I/O handler. The third argument of the .INIT is the routine address, with the high two bits of the word indicating which control character routine is to be changed. The .INIT may be issued to any .DAT slot assigned to TTA., although the examples use .DAT slot -3. For all four control characters, a routine address of zero causes that control character to be ignored.

When API is in use, no special precautions need be observed in writing control character routines. There are certain hazards present, however, when API is not in use. The essential problem is the possibility that the control character interrupt will occur when an I/O handler or some other portion of the monitor is executing. Interrupting the monitor at the wrong place can violate system integrity and cause file structure corruption. The monitor can avoid this problem when it is allowed to use API. See Section 2.2.4 for a detailed discussion of the analogous problem with .TIMER.

This problem can be alleviated somewhat by having the control character routine return to the interrupted code. This causes several additional problems, however, which are described below. A .SCOM location, SC.TTP, is provided to allow a control character routine to return to the interrupted code. Its use is typified by the following example control character routine:

```
SC.TTP=116
routine DAC saveac      / Save AC
        LAC* (SC.TTP)   / and return address
        DAC savepc
        RAR             / and Link
        DAC savlnk
         .              / Set a flag to indicate
         .              / that the control character
         .              / has been typed.
        LAC savlnk      / Restore Link
        RAL
        LAC saveac      / and AC
        JMP* savepc     / then return.
```

The control character routine should follow the guidelines for .TIMER routines outlined in Section 2.2.4.

Although having the control character routine return to the interrupted code does solve the system integrity violation and consequent file structure corruption problems, it also introduces two new problems. The first of these is a re-entrancy problem. If a second control character is typed too quickly following a first, the contents of SC.TTP will be lost and an indefinite loop may result. This problem is inherently insoluble. The second problem results from IOPS errors. If the control character was struck in response to an IOPS error, the very concept of returning from the control character routine is meaningless. The IOPS error processor checks for this and issues an IOPS35 error if a control character routine returns to it.

The only real solution to the entire issue is to always use API. When using API, control character routines should never return. They may be coded to return anyway, but doing so will introduce the two problems described above.

Table 2-1
Specifying Control Character Routines

Control Character	Code (Octal)	Method
↑S	n/a	SC.UST=1Ø6 LAC (routine) DAC* (SC.UST)
↑C ↑P ↑T	2ØØØØØ Ø 4ØØØØØ	.INIT -3,d,code+routine
↑Q	n/a	None

2.4 ABORTING I/O

There are certain occasions in which the programmer may wish to abort I/O transfers which he has requested previously. This is particularly likely following a .TIMER or control character interrupt. The means of doing this is described below.

The primary means of aborting I/O in progress is to issue a .INIT to the busy .DAT slot. This will abort most XVM/DOS I/O handlers.

One exception to this is the console terminal device handler, TTA. The following code sequence is necessary to abort console terminal I/O:

```
SC.CTT=135
        XCT* (SC.CTT)
```

This sequence must be executed before any further CAL's (including .INIT) are issued to TTA.

2.5 THE IOPS ERROR PROCESSOR

XVM/DOS provides two processors for IOPS errors. .MED is used to print a message including the IOPS error code followed by zero, one, or two octal values. The number of octal values is determined by the error code. The expanded error processor prints the same IOPS error message as does .MED, then follows it with an arbitrary string of text. Most XVM/DOS I/O handlers use the expanded error processor; the remainder use .MED.

The IOPS error processors are intended for use by I/O handlers. The error processors, therefore, should only be called from CAL routines and interrupt routines[1]; they should not be called from mainstream.

The expanded error processor obtains information for the error message from the five sources listed below. The .MED error processor obtains its information from the first three of these sources. The sources are:

1. The AC contains the IOPS error code in the low 13 bits. The high 5 bits are ignored; thus the LAW instruction may be used. The expanded error processor, however, expects the AC to normally be negative and attaches a special meaning to a positive AC (see Section 2.5.2). Valid error codes are Ø through 777 (octal) inclusive.
2. For all errors except IOPS4, location .MED contains a value which is printed, in octal, following the IOPS error code. The CAL handler dispatch routine initializes .MED to the address of the CAL instruction. Thus, IOPS errors which print the CAL instruction address need not modify .MED. Note that this only works for CAL routines, not interrupt routines.

[1]This does not include .TIMER interrupt routines and control character routines.

3. If the error is an IOPS2Ø or IOPS21, the contents of SC.BBN is printed, in octal, following the contents of .MED.

4. A string of .SIXBT text follows calls to the expanded error processor. This text is printed to terminate the error message.

5. The Link specifies how nulls (character code ØØ) within the .SIXBT text are to be treated. If the Link is set, nulls will be printed as spaces. If the Link is clear, nulls will be ignored.

The expanded error processor may also be called in such a manner that only the .SIXBT text will be printed; the first three items listed above will be ignored. Detailed specifications of the various calling sequences are given in the sections which follow.

Errors may be either recoverable or non-recoverable. If the error is recoverable, and the operator types a ↑R, the error processor returns to the handler which called it. All registers except the AC and Link will be preserved. If the error is non-recoverable and the operator types a ↑R, the IOPS error message will be repeated. If the operator types any other control character (↑P, ↑T, ↑C, ↑Q, or ↑S) the operation will be aborted and the error processor will not return to the I/O handler. Therefore, the programmer must ensure that the I/O handler and its tables are in a consistent state before calling an error processor.

2.5.1 .MED

When using the .MED error processor, only the IOPS4 error is recoverable. The following code sequence invokes an IOPS4 error message:

```
.MED=3
        LAW 4
        JMS* (.MED)
```

This causes the following message to be printed:

```
IOPS4
```

If the operator responds with a ↑R, the error processor will return following the JMS. The AC and Link will be changed; all other registers will be preserved. In all other cases, the error processor will not return.

All other errors are non-recoverable, and are invoked by the following sequence:

```
.MED=3
        LAC (mmmmmm)
        DAC* (.MED)
        LAW nnn
        JMP* (.MED+1)
```

which causes the following message to be printed:

```
IOPSnnn mmmmmm
```

where "nnn" is the IOPS error code number. The meaning of "mmmmmm" varies with different error codes. The error processor will never return.

The only exceptions to this last calling sequence are the IOPS2Ø and IOPS21 errors. For these errors the sequence is modified to the following:

```
.MED=3
SC.BBN=132
        LAC (bbbbbb)
        DAC* (SC.BBN)
        LAC (mmmmmm)
        DAC* (.MED)
        LAW nnn
        JMP* (.MED+1)
```

which causes the following message to be printed:

```
IOPSnnn mmmmmm bbbbbb
```

where "nnn" is either 2Ø or 21. By convention "bbbbbb" is the particular disk or DECtape block number in which the error occurred. Otherwise this is identical to the previous variation.

2.5.2 The Expanded Error Processor

The expanded error processor is invoked by a code sequence of the following form:

```
.MED=3
SC.BBN=132
SC.EEP=137
        LAC (mmmmmm)
        DAC* (.MED)
        LAC (bbbbbb)
        DAC* (SC.BBN)
        CLL or STL
        LAW nnn
        JMS* (SC.EEP)
        JMP recover
        -count
        .SIXBT "text"
```

This causes one of the following four forms of messages to be printed:

```
IOPS4 text
IOPSnnn mmmmmm text
IOPSnnn mmmmmm bbbbbb text
text
```

The expanded error processor selects one of the first three forms based upon the IOPS error code number "nnn". An error code of 4 results in the first form listed. The codes 2Ø and 21 result in the third form. All other codes result in the second form. The fourth form, in which the IOPS portion of the message has been suppressed, is selected via a mechanism described below.

As with the .MED error processor, the meaning of "mmmmmm" varies with the error code number. By convention, "bbbbbb" is the particular disk or DECtape block number in which the error occurred. If the form being used does not include either or these values, the location containing it (.MED for "mmmmmm" or SC.BBN for "bbbbbb") need not be set up, allowing a portion of the code sequence above to be omitted.

The first three message forms above (the IOPS forms) all require that the AC be negative when the error processor is called. This will happen quite naturally if the LAW instruction is used. If, however, the AC is positive, the fourth message form results. The IOPS portion is omitted and only "text" is printed. When using this form, the AC should contain

zero; AC values greater than zero are undefined. Locations .MED and SC.BBN need not be set up.

In the code sequence above "text" stands for an arbitrary .SIXBT text string. This text string is printed following the IOPS portion of the message (if present). Null characters within the text string will either be ignored or converted to spaces, depending upon the value of the Link. If the Link is clear, nulls will be ignored; if it is set, they will be converted to spaces. A null is the character code ØØ. Nulls may be inserted in a .SIXBT text string by specifying the "@" character. This feature allows an I/O handler to easily format an error message, by concatenating various segments aligned on word boundaries.

The length, in words, of the .SIXBT text string is represented by "count" in the code sequence above. The two's complement of the length should be provided. If the value provided is positive or zero, the text string will be omitted.

The expanded error processor considers all errors to be potentially recoverable. The error processor returns following the JMS if the operator types a ↑R. If the error is in fact recoverable, this location should be a JMP to a recovery routine. If the error is non-recoverable, this should be a JMP, which loops back to re-call the error processor. Note that the AC and Link have been modified and must be restored before the error processor is re-called.

2.6 .MTRAN

XVM/DOS provides an interface to the system bootstrap, implementing transfers to and from the system disk. This interface is via the .MTRAN (Monitor TRAN) CAL function. .MTRAN is a system macro; its expansion is given in Appendix E.

.MTRAN should be invoked with a code sequence similar to the following:

```
LAC (prmblk)
CLL or STL
.MTRAN
```

The .MTRAN CAL does not return in the normal fashion; when the transfer completes the program continues in a manner described below. The state

of the Link indicates the transfer direction. If the Link is clear, information is read from the system disk into core. If the Link is set, information is written from core onto the system disk. The location and length of the transfer is determined by a four word parameter block. The address of the first word of the parameter block is conveyed in the AC. The parameter block takes the following form:

```
prmblk  blknum          / first disk block.
        coradr-1        / starting core address
        -length         / two's complement transfer length
        restart         / program restart address
```

The first word contains the number of the first disk block to be transferred. The second word contains one less than the core address at which the transfer is to start. The third word contains the two's complement of the number of words to transfer. The fourth word contains the address at which the program should continue when the transfer is complete. .MTRAN does not return in the normal fashion, to the location following the CAL expansion. Rather, it returns to the address specified in the fourth word of the parameter block. Upon return from a .MTRAN, all registers have been altered. All that is preserved is the current addressing mode - both bank vs. page mode and XVM mode enable/disable. Note that the parameter block must reside entirely within the first 32K words of core.

Although XVM/DOS provides this interface to the system bootstrap, its use is discouraged. Using the bootstrap requires that the entire interrupt system be shut down. The programmer is encouraged to use the .TRAN function of a conventional disk handler whenever possible.

If a disk error is encountered while performing a .MTRAN, the following message will result:

```
IOPS4
```

This message results for all errors, not just drive-not-ready errors. Typing a ↑R will cause the transfer attempt to be repeated.

2.7 XVM MODE

Unlike bank and page modes, the programmer may dynamically turn XVM mode (17-bit addressing mode) on and off within his program.[1] This is

[1] Only the system loader may switch between bank and page modes, as the XVM/DOS monitor and associated I/O handlers must be reconfigured.

accomplished with the .XVMON and .XVMOFF CAL functions. These functions are provided as system macros.

In order to use these CAL's, the programmer must ensure that XVM mode has been enabled. This is accomplished by issuing the XVM ON command to the non-resident monitor. If XVM mode is disabled (XVM OFF), the .XVMON CAL will result in an IOPSØ error. The .XVMOFF CAL will still execute successfully, however.

It should not be necessary for the user to issue either of these CAL's when the system is used normally. All DEC supplied system programs run with XVM mode turned off. Accordingly, the system loader always leaves XVM mode off. The Linking Loader and Execute, both of which load user programs, turn on XVM mode (issue a .XVMON) if XVM mode has been enabled.

2.8 ADVANCED .GET/.PUT

There are a number of options included in the .GET and .PUT CAL functions which are not described in the XVM/DOS Users Manual. These options are controlled by various flags in the first argument (the function argument) to the .PUT and .GET system macros.

Every function performed by .PUT or .GET results from three operations performed in sequence. These operations may be selectively omitted or altered by various flags within the function code and by whether the CAL is a .PUT or a .GET. The meanings and octal values of the flags are summarized in Table 2-2.

The three operations comprising .PUT and .GET are as follows:

1. Core dump. The monitor first closes all open files. Then, if the CAL is a .PUT, it transfers the core image into the ↑Q area on the system disk.
2. File transfer. The ↑Q area is transferred to or from the named disk file, or else no operation takes place. If the flag SC.QNF is set in the function code, this operation is skipped and no file transfer operation takes place. If the flag is clear, the transfer operation will be performed in a direction determined by the flag SC.QPUT.

If SC.QPUT is clear, the file named by the .GET CAL is copied into the ↑Q area. If SC.QPUT is set, the ↑Q area is copied to the file. Note, however, that the flag SC.QPUT is complemented when the CAL is a .PUT. Thus, the meanings of this flag become reversed from what is described above.

3. Exit. If the flag SC.QNRM is set, the system exits to the non-resident monitor. If SC.QNRM is clear, the ↑Q area is brought into core and control is returned to the user program. The manner in which control is returned is determined by the return code portion of the function code, described in the XVM/DOS Users Manual.

In the course of these operations numerous error checks are performed.

Table 2-2
.GET/.PUT Special Function Flags

Flag	Octal Value	Meaning
SC.QNF	2ØØØ	Skip file transfer operation
SC.QPUT	4ØØ	Determine direction of file transfer. Meaning when .GET: set => ↑Q area→ file clear => File→ ↑Q area These meanings are reversed when used with .PUT.
SC.QNRM	1ØØØ	Exit to non-resident monitor when finished, rather than restoring core image

The core image saved and restored by .PUT and .GET does not include all of core. Locations Ø to 4 are never included. This should be of no consequence to the user, as these locations are reserved for the monitor. It is also possible that the image will not extend to the top of memory. If the ↑Q area is shorter than the current system memory size, the image will be truncated to the ↑Q area length. Note that memory above the bootstrap may be omitted from the saved (and subsequently restored) image without a warning message ever appearing.

Example 1:

The non-resident monitor command line:

```
$PUT␣FILNAM␣EXT
```

might be translated to the following code sequence:

```
        .GET SC.QNRM!SC.QPUT,filblk
          .
          .
          .
filblk  .SIXBT "FILNAM"
        .SIXBT "EXT"
```

Example 2:

The action performed by a ↑Q is closely approximated by the following code sequence:

```
SC.QNRM=1ØØØ
SC.QNF=2ØØØ
        .PUT SC.QNRM!SC.QNF,Ø
```

The monitor actually uses this code sequence to implement ↑Q, after first modifying the .GET CAL function routine. See Chapter 3 for more details.

2.9 TASK CONTROL BLOCKS

Communication between the XVM processor and the UNICHANNEL (PDP-11) processor is performed through blocks of information called Task Control Blocks. These blocks are resident in the first 4K words of XVM memory where they can be accessed by both processors. The TCB (Task Control Block) contains all the information necessary for the processing of a UNICHANNEL request. For more details refer to the XVM UNICHANNEL Software Manual.

Space for Task Control Blocks is allocated whenever the UC15 option is enabled. This option is enabled with the UC15 ON command and disabled with the UC15 OFF command. The UC15 OFF command will reclaim (for other purposes) the space allocated to the Task Control Blocks.

Task Control Blocks are accessed by means of .SCOM location SC.TCB. If the UNICHANNEL is disabled (UC15 OFF), the TCB area is not allocated and SC.TCB contains zero. Otherwise, SC.TCB points to a table of transfer vectors, containing one location for each TCB.

Each location in this table is fixed, corresponding to a particular TCB. This table is currently nine words long, as there are currently nine TCB's. Existing table locations will not be changed in the future; changes will be implemented by adding additional table locations.

Each potential Task Control Block has a dedicated location in this transfer vector table. This location will be zero if the associated TCB has not be allocated. Otherwise, it will be the base address of the TCB. The length of the TCB is determined when the resident monitor is assembled. Each TCB has a standard default length. While users may change the length, they should do so with caution, as all software which may reference that TCB must be modified to reflect the change.

The various Task Control Blocks, the assembly parameters which control their generation, and their transfer vector locations within the table pointed to by SC.TCB are all summarized in Table 2-3. Which TCB's are allocated, and the length of each TCB when it is allocated, is determined by several assembly parameters to the resident monitor. Each TCB has its corresponding assembly parameter. If the parameter is left undefined, the TCB will not be allocated. If the parameter is defined equal to zero, the default length will be used. If the parameter is defined with a non-zero value, that value is used for the TCB's length and the default length overridden.

There are two exceptions to the above rule. Both the disk Task Control Block (RKTCB) and the line printer Task Control Block (LPTCB) will always be generated. This occurs even if the assembly parameters are left undefined, as these two TCB's are required by various pieces of system software.

We wish to stress that the foregoing discussion of TCB assembly parameters only applies when the UNICHANNEL is enabled. When the UNICHANNEL is disabled (UC15 OFF), no TCB's are allocated, regardless of what assembly parameters are used. The assembly parameters determine which TCB's will be allocated when the UNICHANNEL is enabled (UC15 ON).

Table 2-3
Task Control Blocks

Offset[1]	Assembly Parameter[2]	Default Length[3]	Use
Ø	RKTCB*	21	Cartridge disk TCB
1	LPTCB*	117	Line Printer TCB and Buffer area
2	CDTCB	65	Card Reader TCB and Buffer area
3	PLTCB	117	XY (incremental) Plotter TCB and Buffer area.
4	TCB1	24	Spare TCB and Buffer area #1.
5	TCB2	120	Spare TCB and Buffer area #2.
6	TCB3	170	Spare TCB and Buffer area #3.
7	LVTCB	120	Printer/Plotter (electrostatic) TCB and Buffer area.
10	DLTCB	54	Communications TCB and Buffer area.

[1]The offset, shown in octal, determines the location of the TCB's transfer vector within the table pointed to by SC.TCB. The following code sequence loads the AC with the base address of a TCB:

```
SC.TCB=2ØØ

        LAC* (SC.TCB)
        SNA
        JMP tcb not allocated
        AAC offset
        DAC temp
        LAC* temp
        SNA
        JMP tcb not allocated
```

[2]The two TCB's flagged with an asterisk are always generated. Leaving either RKTCB or LPTCB undefined is equivalent to setting it equal to zero.

[3]The default lengths are expressed in octal.

2.10 MINI-ODT

NOTE

Mini-ODT is an UNSUPPORTED feature of XVM/DOS. It is described here as a convenience to customers who may wish to use it in spite of this restriction.

Mini-ODT is an optional feature of the XVM/DOS resident monitor. It is included in the monitor by defining the following assembly parameter when assembling RESMON:

```
%ODT=Ø
```

If present, Mini-ODT will be invoked on all IOPS errors. The ↑T control character routine will, by default, be Mini-ODT. Thus, typing a ↑T will invoke Mini-ODT unless the user program respecifies the ↑T control character routine or DDT, Batch or BOSS is in use. All interaction with Mini-ODT occurs via the console terminal.

When an IOPS error occurs, Mini-ODT prompts for a command on the line following the normal IOPS error message. When Mini-ODT is invoked via ↑T, it types a line of information and prompts for a command on the following line. The line of information is of the form:

```
ODT > PC=pppppp   AC=aaaaaa
```

This line describes the contents of the PC and AC when the ↑T interrupt occurs.

Mini-ODT prompts for commands by printing the following sequence at the left margin:

```
ODT >
```

Note that every line printed by Mini-ODT begins with this same sequence. Legitimate commands are listed in Table 2-4. If an invalid command is entered, Mini-ODT types two question marks, closes any open location, and prompts for a new command.

Table 2-4

Mini-ODT Commands

Command	Action
nnnnnn/	Close any open location, then open location nnnnnn and print its contents. Always legal.
↵	Close the currently open location, then open the next location in sequence and print its contents. Illegal if no location is open.
nnnnnn↵	Store nnnnnn in the currently open location and close it, then open the next location in sequence and print its contents. Illegal if no location is open.
$	Close any open location. Always legal.
nnnnnn$	Store nnnnnn in the currently open location and close it. Illegal if no location is open.
↑P, ↑C, ↑T, ↑S, ↑Q	Same meaning as with error processor without Mini-ODT. Exits error processor and Mini-ODT and transfers to control character routine.
R	Return from error processor and Mini-ODT. If Mini-ODT was invoked by an IOPS error, the handler retrys the I/O transfer. If invoked by ↑T, resume the interrupted program.

Key:

nnnnnn	Any octal number
↵	Carriage return
$	Alt Mode. A dollar sign will not be echoed.

2.11 RESIDENT MONITOR PATCHING FACILITIES

There are two distinct types of patching available with the XVM/DOS resident monitor. The first of these provides what is normally meant by a patch area. This method is appropriate for altering the resident monitor. The second method, although using what is termed the resident patch area, is really a means of communication between different core loads.

To add code (presumably a patch) to the resident monitor, the following procedure should be followed. Using the Patch XVM utility program, examine location 1Ø1 within the disk image of RESMON. This location's contents is the address of the first free location above the resident monitor. The code and data should be inserted beginning at this address, and the contents of location 1Ø1 updated to reflect the monitor's increased length.

The second patching method uses the aforementioned resident patch area. The resident patch area is allocated by SGEN XVM when a non-zero response is given to the question:

RESIDENT PATCH AREA SIZE [Ø]

The response to this question determines the length (in words) of the resident patch area. The base address of the resident patch area is contained in location 1Ø1 of the disk image of RESMON. Note that patching the resident monitor via method 1 above will relocate the resident patch area.

The resident patch area gets its name from the fact that it is truly resident. It is initialized from the monitor's disk image upon a manual bootstrap load (cold-start). Following this it is left undisturbed, from core load to core load, unlike the rest of memory which is constantly refreshed. Thus it is a convenient method of communication between core loads.

It should be noted that references to location 1Ø1 apply only to the disk image of RESMON. This location is changed when the monitor is brought into core and initialized. The programmer must also be aware that an upper limit is placed on the combined sizes of the two patch areas when the resident monitor is assembled. The patch areas should be restricted to a total combined length of no more than 1ØØØ (octal) words.

2.12 API SOFTWARE INTERRUPTS

This section presents the coding conventions which should be followed when a particular API software interrupt level is shared among several uses. The conventions must be followed by the user if he uses level 4, as the XVM/DOS monitor and I/O handlers use level 4 also. They need not be followed with levels 5, 6, and 7 as these levels are reserved exclusively for the user. However, it is still recommended that these conventions be followed, to allow the levels to be used for additional purposes.

The routine which requests a level 4 software interrupt should be coded as follows:

```
API.R4=4Ø4ØØØ
SC.LV4=112
.INH=7Ø5522
.ENB=7Ø5521

        .INH                ///
        RPL                 /// Save previous level
        AND (API.R4)        /// 4 request.
        DAC svrqst          ///
        LAC* (SC.LV4)       /// And the interrupt
        DAC svvctr          /// vector.
        LAC (API.R4)        ///
        ISA                 /// Issue a request
        LAC (routine)       /// for our routine.
        .ENB                ///
        DAC* (SC.LV4)       ///
```

This coding sequence saves a current software interrupt request (if one is present) so that it may be restored later. Interrupts are disabled so that an interrupt routine, itself trying to schedule a software interrupt, won't find things in an inconsistent state. An analogous coding sequence should be followed with software levels 5, 6, and 7, substituting the appropriate mnemonics and values from Table 2-5 for API.R4 and SC.LV4.

The software interrupt routine must be coded to complement the requesting routine above. The interrupt routine should be coded as follows:

```
SC.LV4=112
.INH=7Ø5522
.ENB=7Ø5521

routine Ø               /// Software interrupt
        .INH            /// routine entry point.
        DAC saveac      /// save AC.
        LAC svrqst      /// Restore previous
        ISA             /// level 4 request.
        LAC svvctr      /// And its interrupt
        .ENB            /// vector.
        DAC* (SC.LV4)   ///
         .
         .
         .
        LAC saveac      / Restore the AC
        DBR             / and operating modes
        JMP* routine    / and return.
```

This routine restores the previous software interrupt request which was saved by the scheduling routine. It is essential that no interrupts be allowed after the moment the software interrupt is granted until the previous request has been completely restored. Routines for levels 5, 6, and 7 should be coded in a like manner, substituting the appropriate mnemonic and value from Table 2-5 for SC.LV4.

The programmer should note that XVM mode is turned off when the software interrupt occurs. A .XVMON CAL should be issued if XVM mode is required within the software interrupt routine. The DBR instruction used to return from the software interrupt routine causes XVM mode to be restored to the proper state.

The programmer must also remember that software interrupts may only be used when XVM/DOS has been conditioned to use API. This conditioning occurs when the API ON command is given. Enabling the API hardware when the monitor is not so conditioned (API OFF) is erroneous.

Table 2-5

API Software Interrupt Vectors and Request Flags

Priority Level	Interrupt Transfer Vector		Request Flag	
	Mnemonic	Octal Value	Mnemonic	Octal Value
4	SC.LV4	112	API.R4	404000
5	SC.LV5	113	API.R5	402000
6	SC.LV6	114	API.R6	401000
7	SC.LV7	115	API.R7	400400

CHAPTER 3
THE RESIDENT MONITOR

This chapter contains information concerning the internal operation of the resident monitor for those who may wish to alter it. These alterations, however, are not supported by Digital Equipment.

A fresh copy of the resident monitor is built for each core load and becomes a permanent part of the core load. The .SCOM table, which contains absolute locations used to communicate between XVM/DOS components, is an important part of the resident monitor. The resident monitor also contains the CAL handler and associated CAL function routines, the console terminal I/O handler (TTA.), plus various pieces of optional, specialized code for BOSS, Batch, VT ON, and other features.

3.1 RESIDENT MONITOR INITIALIZATION

A fresh copy of the resident monitor is built for every core load by reading it from disk into core and transferring to the initialization code. Usually this process is used by the monitor to rebuild itself for a new core load. A less common use of this process is for a manual bootstrap or cold start.

Manual bootstrap loads read the resident monitor into bank zero of core. The initialization code copies the monitor to the memory bank in which the bootstrap resides and transfers control to that bank. Next .SCOM is cleared and selected locations initialized. The contents of SGNBLK are the primary information sources for initializing .SCOM.

These steps construct an approximation to an operational core image, allowing the completion of the initialization process using the more common monitor rebuild procedure. This procedure is described below.

Whenever a new core load is constructed, the resident monitor is read into the memory bank in which the bootstrap resides. The monitor initialization code is executed, rebuilding the monitor in bank zero.

Upon completion of this build procedure the initialization code overlays itself with the system loader (.SYSLD). The system loader completes the core load by loading the appropriate system program and requested I/O handlers. The rest of this section describes the initialization code in more detail.

Immediately upon being entered the initialization code (executing in the bootstrap's bank) saves certain key tables from the previous core load's monitor. At this time the previous core load's monitor is still in bank zero. The tables which are saved include the old .SCOM, the .DAT and .UFD tables, and the VT ON display buffer (if present). After these tables are saved, low memory -- in particular .SCOM -- is zeroed.

At this point the build procedure begins. The new .SCOM is built using the old .SCOM and SGNBLK as primary data sources. Entries in .SCOM are checked for validity in terms of the current hardware configuration, and altered if necessary. The monitor is copied into bank zero and configured for the current operating modes. Other system tables are set up and initialized. These tables include the skip chain and the .DAT, .UFD, and TCB tables. Finally the interrupt system is turned on and the system loader brought into core.

Monitor appendages play an important role in configuring the resident monitor for various operating modes. A monitor appendage is a section of code which may easily be included as part of the monitor or left out and the space recovered for other purposes. Monitor appendages are used to implement many optional features, such as Batch and BOSS modes and VT ON.

A monitor appendage is divided into two sections. The first section is instructions, the second absolute constants. The monitor initialization code checks if the appropriate option is enabled, and calls a routine (IN.MOV) to add the appendage if appropriate. This routine relocates all memory reference instructions and copies the rest of the appendage, including the absolute constant section, without modification. The necessary registers are modified to reflect the increased size of the monitor. Note that no provision is made for relocatable constants -- any such must be handled separately.

3.2 THE CAL HANDLER

The CAL handler is a basic dispatch routine. There are two varieties of CAL's, each handled separately. If the CAL is an I/O CAL, the .DAT slot number is determined from the CAL instruction and the I/O handler address obtained from the .DAT table. The CAL handler transfers to the I/O handler via a JMP instruction, with the CAL instruction address in the AC.

Non-I/O CAL's are performed by CAL function routines internal to the resident monitor. The CAL handler obtains parameters from the CAL expansion and stores them in places convenient for the CAL function routines. The CAL handler then invokes the appropriate CAL function routine via a JMS instruction. When the CAL function routine returns, the CAL handler returns to the user program. A more detailed description is provided by the accompanying flow diagrams (Figure 3-1) and by comments included in the resident monitor.

The above description of the CAL handler is greatly simplified, as it completely ignores a major problem which may occur. This problem centers upon the possibility of certain interrupts occurring during monitor processing. The interrupts in question are control character interrupts from the console terminal keyboard and clock interrupts which cause a .TIMER routine to be invoked. The troublesome characteristic of these interrupts is that they cause an asychronous transfer of control in the user program. If the problem were ignored, and an I/O handler or the monitor were interrupted, various internal tables could be left in an inconsistent state. Such a violation of system integrity can have consequences as far reaching as disk file structure corruption.

The solution to this problem is to provide a way to determine whether or not the monitor is active -- i.e. whether the monitor or the user program is interrupted. If the monitor is inactive, the control character or .TIMER interrupt takes effect immediately. If the monitor is active, the control character or .TIMER interrupt should be delayed until monitor activity terminates, so that only the user program is affected. This solution is in fact achieved when API can be used. Without API, however, the problem still exists.

The API solution makes extensive use of level 4 software interrupts, and the fact that CAL instructions implicitly raise the processor to level 4. The CAL handler uses these techniques to maintain a CAL level

counter, indicating the nesting depth of currently active CAL's. Every time a CAL is issued, the CAL handler increments the CAL level counter and requests a level 4 software interrupt. When the CAL function routine completes, it returns, simultaneously dropping below API level 4. The level 4 software interrupt occurs, allowing the CAL level counter to be decremented. If the counter reaches zero, all CAL's have completed and the user (mainstream) program is active, so the handler drops back to mainstream and returns from the software interrupt. If the counter hasn't reached zero, a CAL function routine (which has just itself issued a CAL) is still active, so the handler remains at level 4 when it returns from the software interrupt. These operations are further described in Figure 3-1.

The above procedure maintains an indication of whether or not a CAL routine is active. Note that this works equally well for both I/O CAL's processed by I/O handlers and non-I/O CAL's processed internally. There must also be a way to detect if an interrupt routine is active. This is quite easily achieved, however, as all interrupt routines operate at API levels 4 and above.

This information about CAL and interrupt routine activity is used by the monitor when processing control character and .TIMER interrupts. The logic to handle this is contained in a monitor subroutine, RQ.LV4. RQ.LV4 is called by an interrupt routine to schedule an auxiliary routine for execution at a later time. An auxiliary routine is identical to a level 4 software interrupt routine, with two exceptions. The important difference is that the "software interrupt" which invokes the auxiliary routine is guaranteed to interrupt the user program, rather than any portion of the monitor. Thus when the auxiliary routine receives control, the monitor is inactive and all CAL and interrupt routines have completed. The second difference between an auxiliary routine and a level 4 software interrupt routine is that an auxiliary routine need not restore a previous software interrupt request (see Section 2.12).

When RQ.LV4 is called to schedule an auxiliary routine, it firsts checks the CAL level counter to see if a CAL function routine is active. If no CAL function routines are active, it requests a level 4 software interrupt and invokes the auxiliary routine when the software interrupt is granted. If a CAL function routine is active, however, it arranges for the level 4 software interrupt to be requested when all active CAL

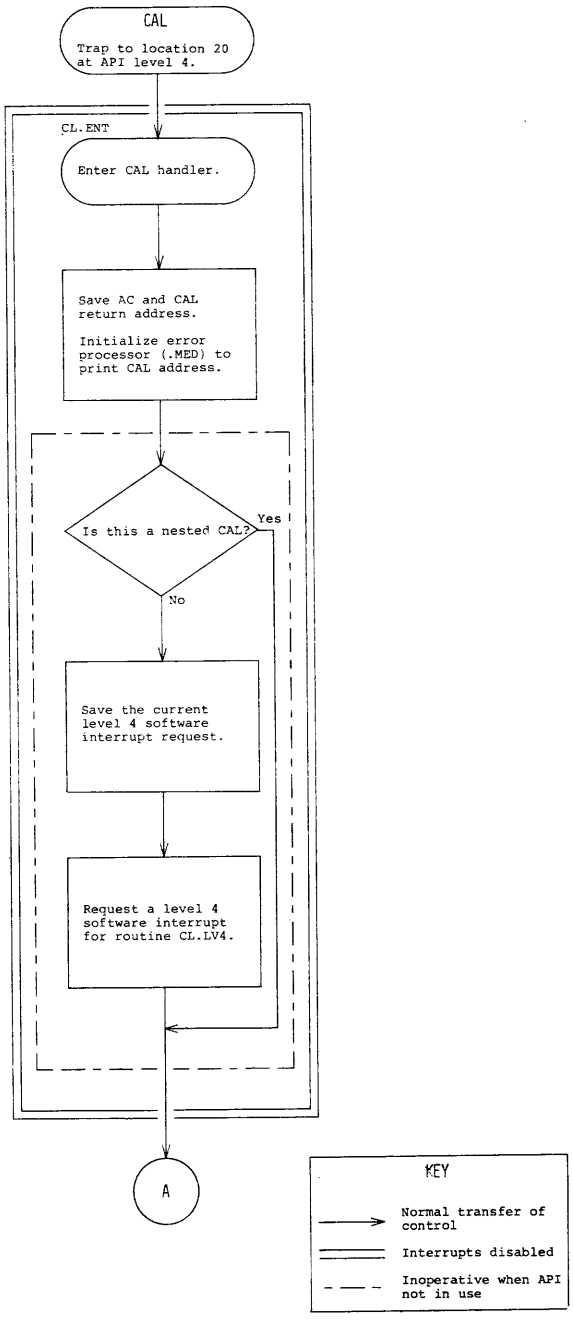


Figure 3-1
The CAL Handler

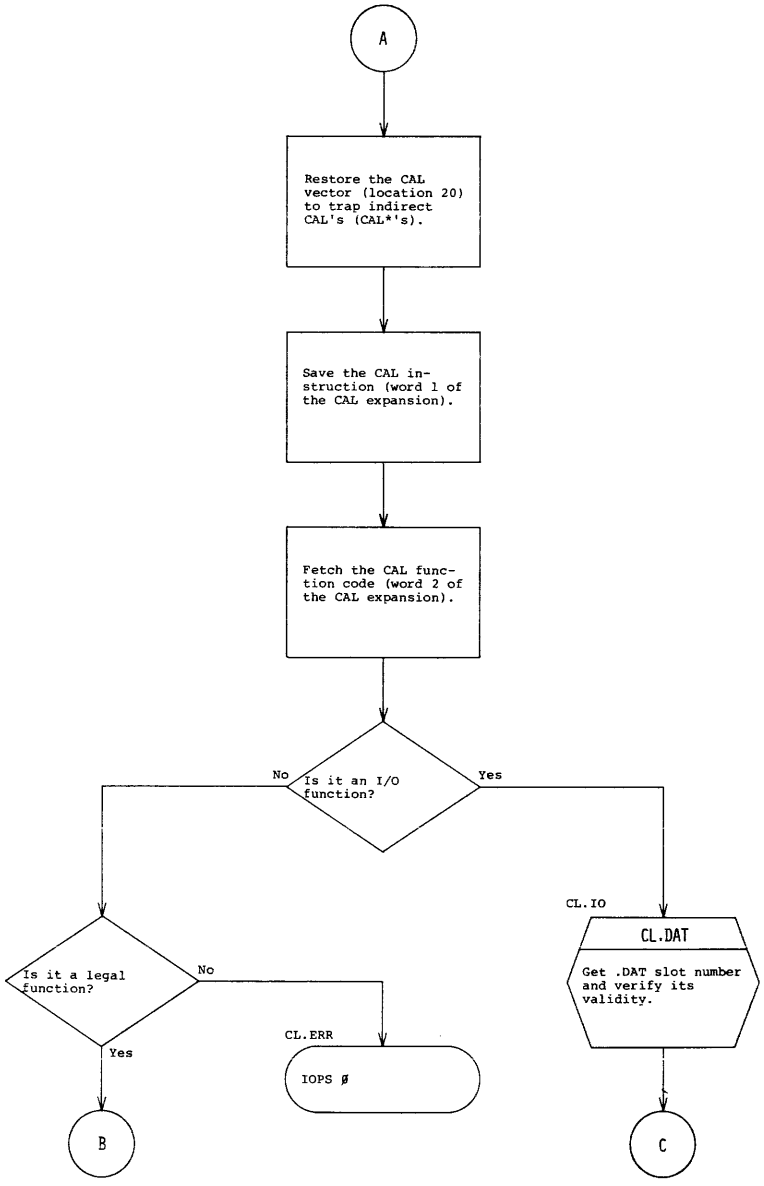


Figure 3-1 (cont)
The CAL Handler

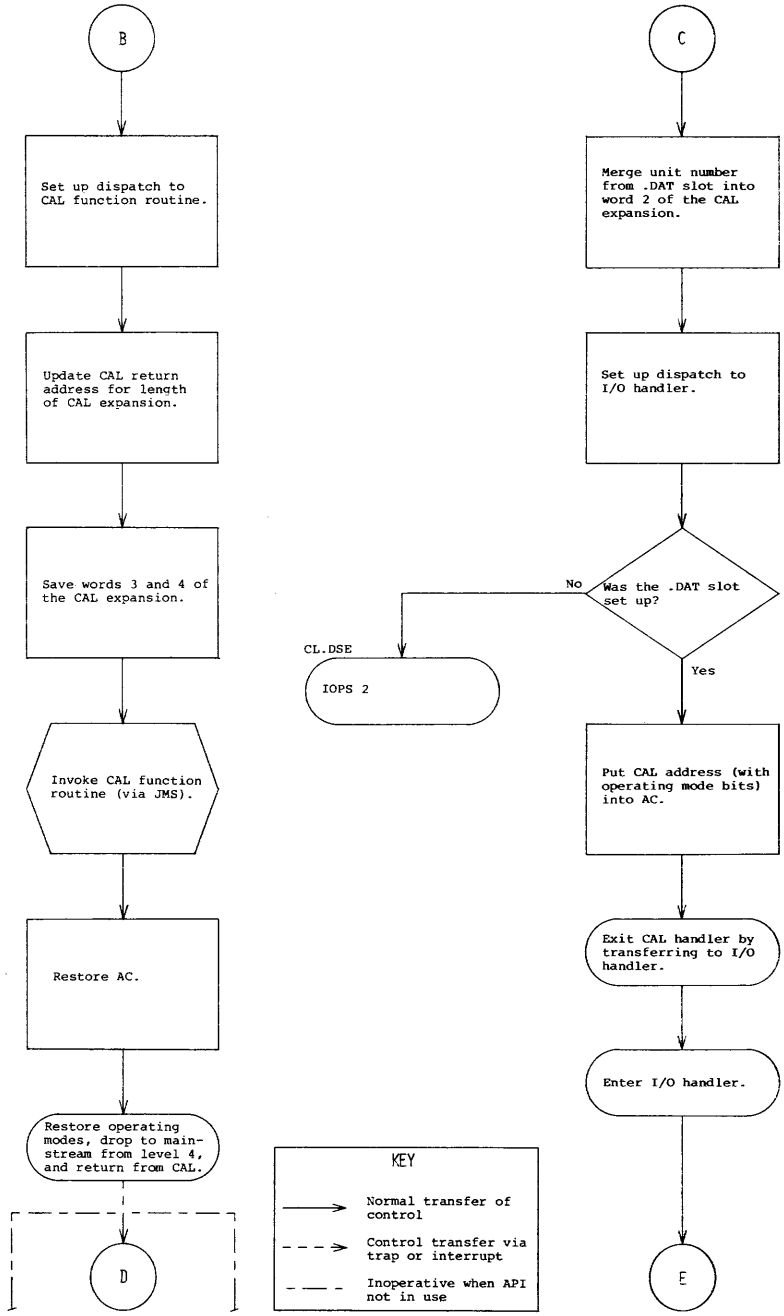


Figure 3-1 (cont)
The CAL Handler

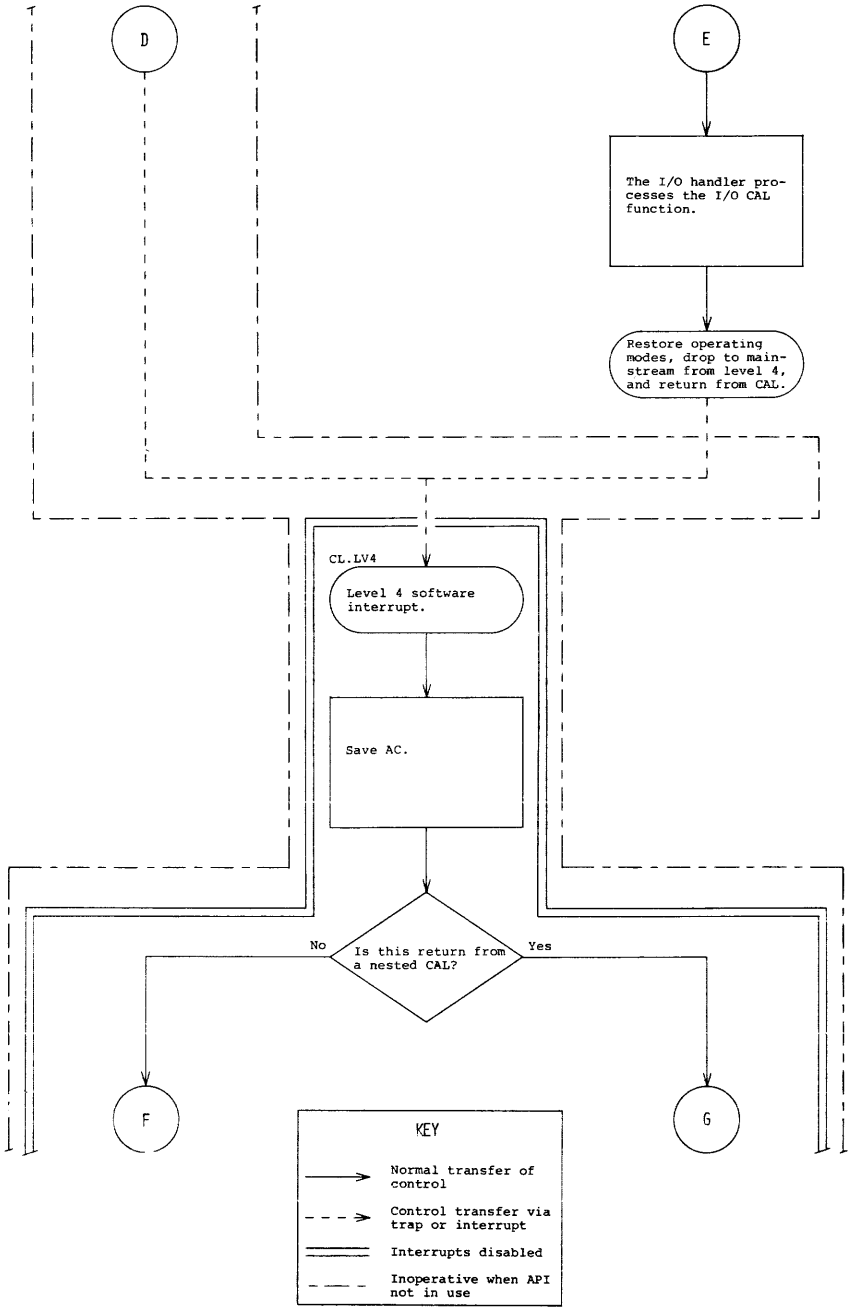


Figure 3-1 (cont)
The CAL Handler

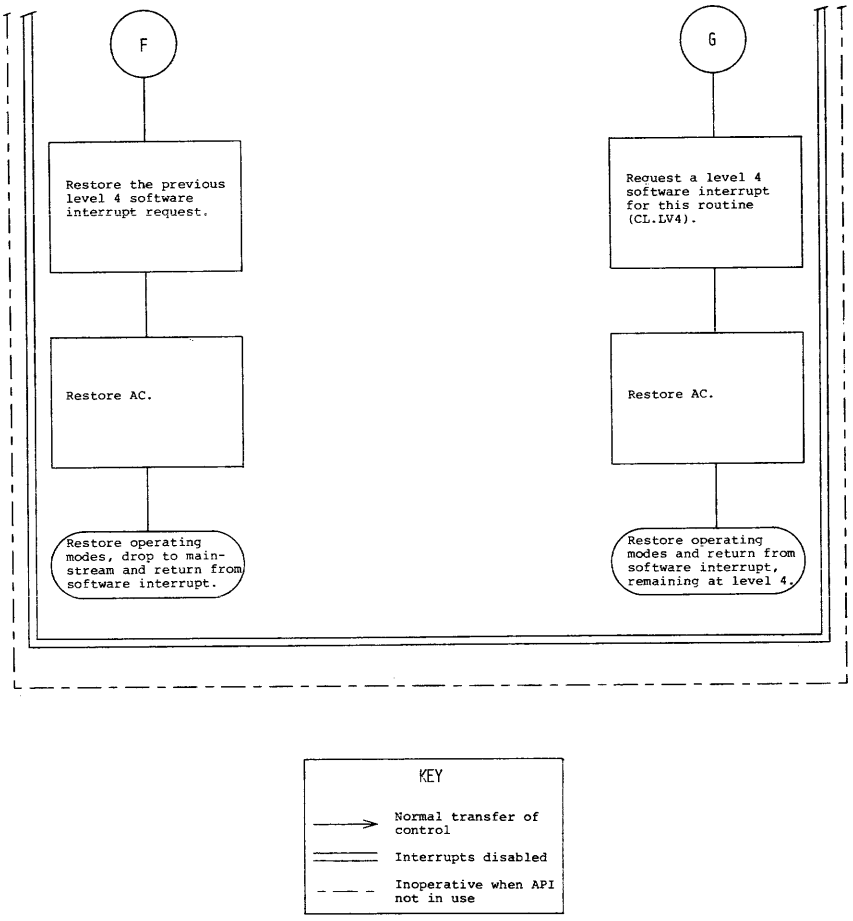


Figure 3-1 (cont)
The CAL Handler

function routines terminate. This logic is illustrated by Figure 3-2. The interaction with the CAL level counter ensures that all CAL function routines have completed when the auxiliary routine is invoked; the level 4 software interrupt request ensures that all interrupt routines are complete.

The use of RQ.LV4 is well illustrated by the .TIMER interrupt logic. When the clock interrupt routine detects that the .TIMER time interval has expired, it calls RQ.LV4 to schedule an auxiliary routine. Following this, the clock interrupt routine continues operation and completes normally. Ultimately the auxiliary routine is invoked via the procedure described above. The auxiliary routine, in turn, actually invokes the user's .TIMER interrupt routine.

The foregoing discussion has described a solution to the system integrity problem which centers upon control character and .TIMER interrupts. As stated previously, this solution uses API extensively -- particulary level 4 software interrupts. When API cannot be used (either because it isn't present or because its use has been disabled), this solution is inapplicable; the problem becomes insoluble and it most frequently manifests itself as disk file structure corruption.

An example of how this problem might occur is as follows. Suppose a user were to use the PIP XVM utility program to delete a disk file. Furthermore suppose that the filename was typed incorrectly, but that the error was not discovered until after the command line was terminated with a carriage return, so that the wrong file would be deleted. Upon discovering this error, most users would immediately try to abort the deletion with a ↑C or ↑P which would be too late to save the file. However, unless the file were extremely short, this would cause disk corruption. Specifically, some of the blocks which had been allocated to the file would be lost from the disk's bit maps and unavailable for further use.

Even though this problem is inherently insoluble without API, we cannot just ignore it. We must exert some effort to preserve the correct status of XVM or user mode (see Section 3.4). A user program may execute in either user mode or exec mode, whereas the monitor always executes in exec mode. This requires attention in conjunction with .TIMER and other user interrupts which might return to the interrupted code.

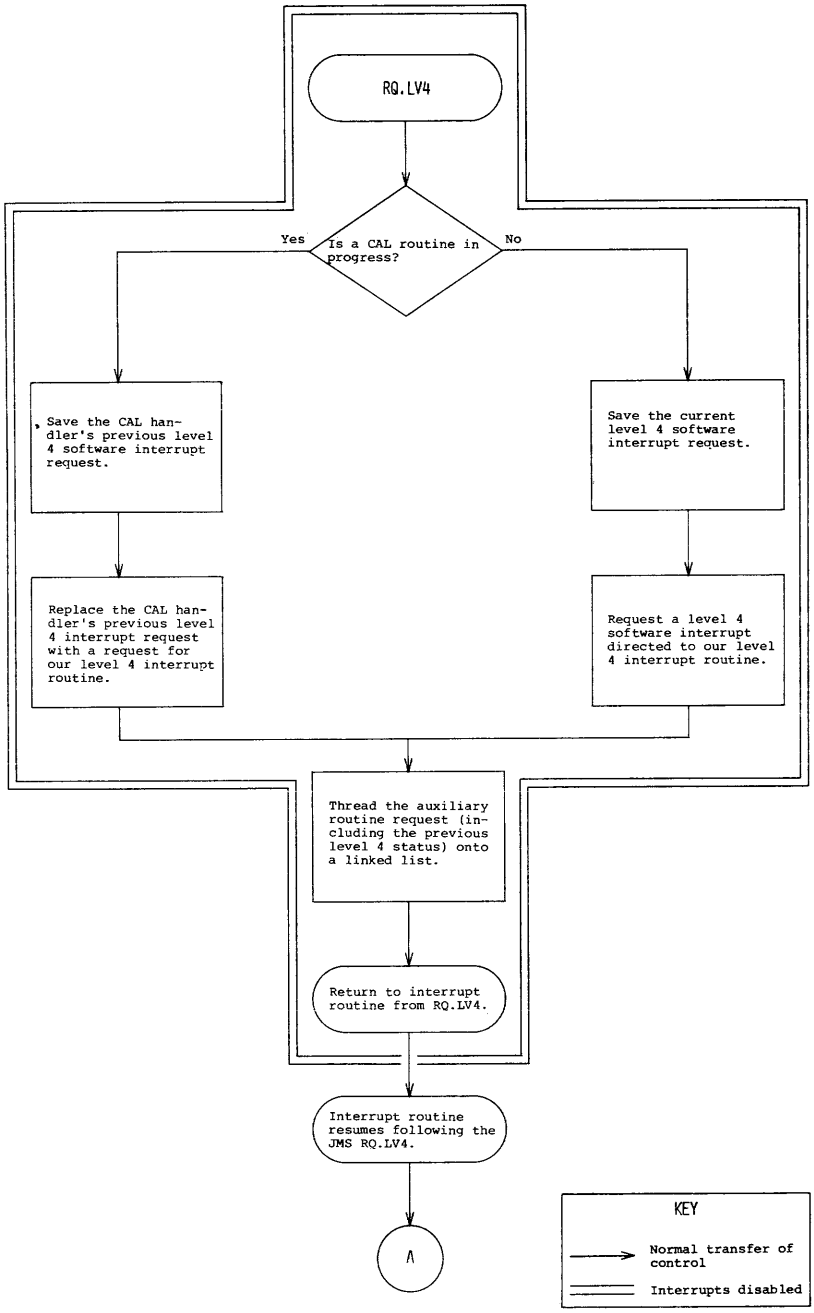


Figure 3-2
The Auxiliary Routine Scheduler, API Version

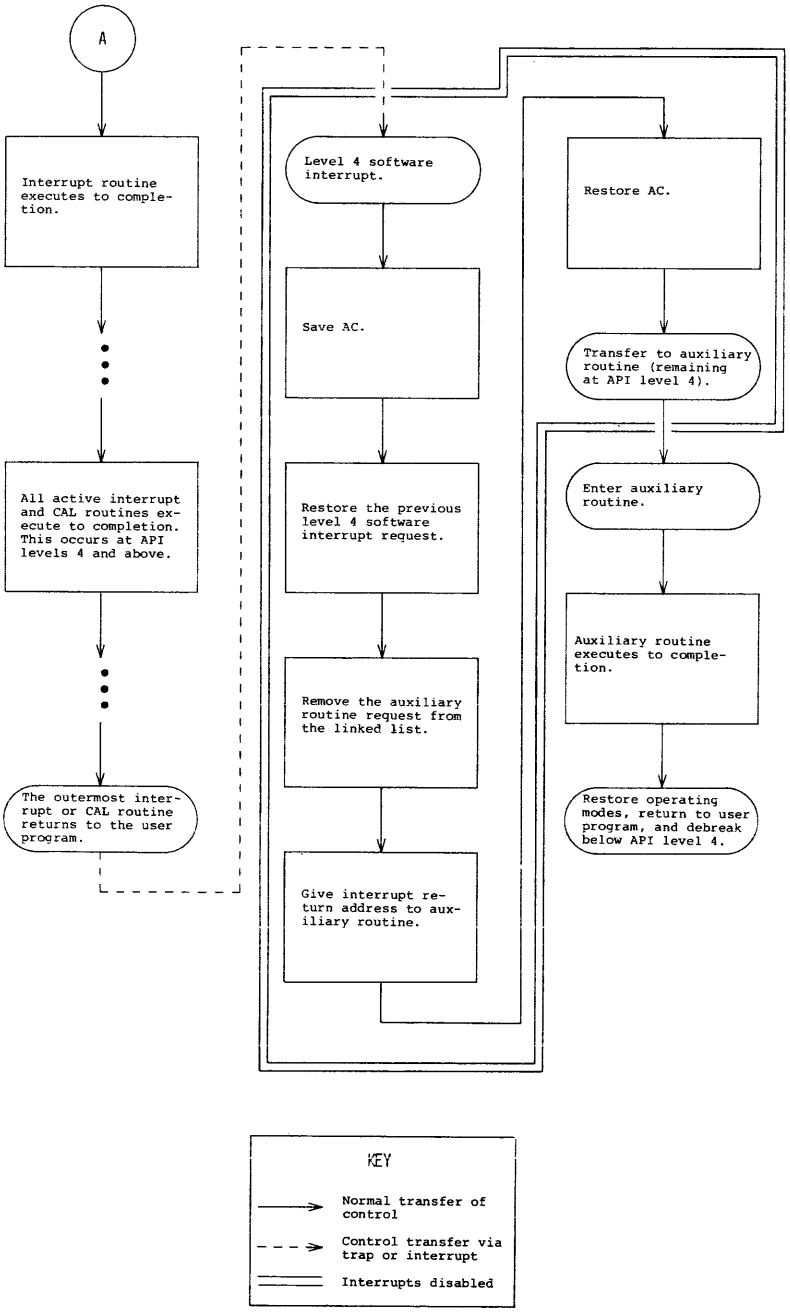


Figure 3-2 (cont)
The Auxiliary Routine Scheduler, API Version

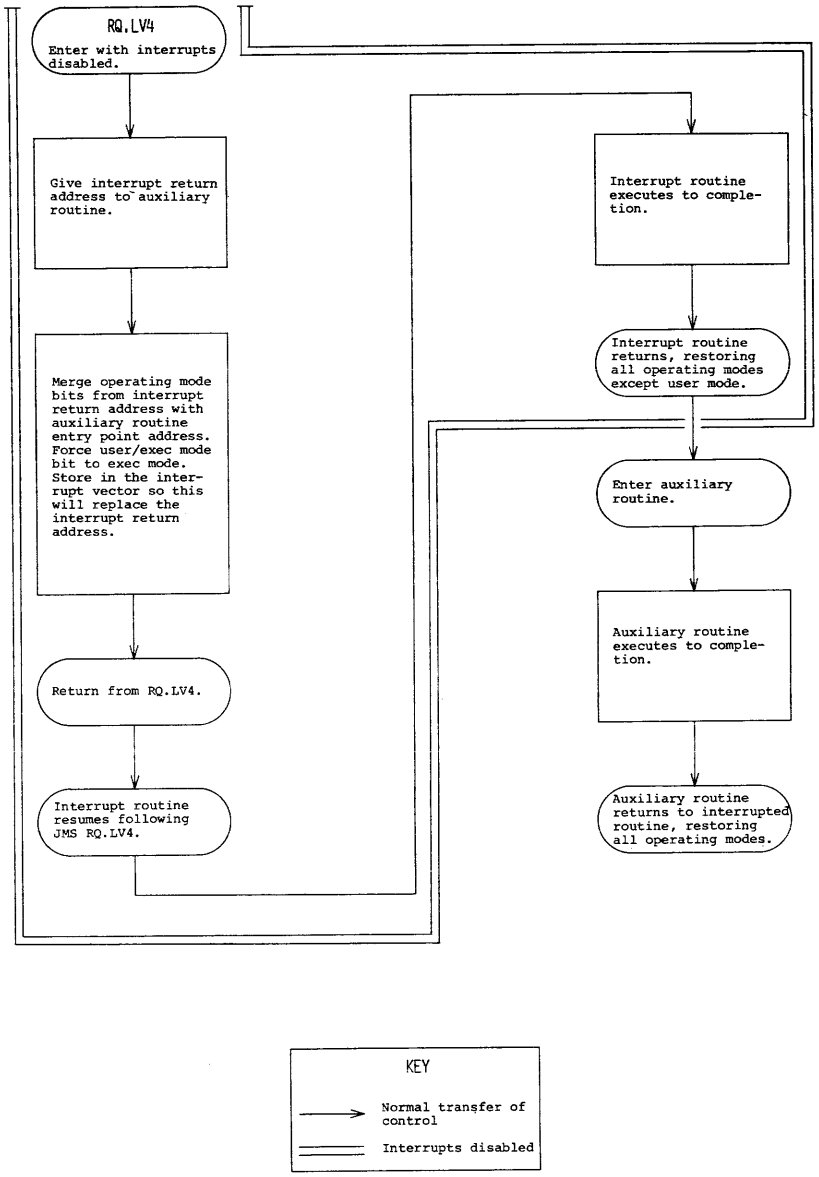


Figure 3-3
The Auxiliary Routine Scheduler, Non-API Version

Suppose the monitor is active when a .TIMER interrupt occurs. We must leave exec mode and enter user mode before calling the user's .TIMER interrupt routine. Correspondingly, when the .TIMER interrupt routine returns, we must restore exec mode before returning to the monitor. These mode transitions are handled by the auxiliary routine. The non-API version of the auxiliary routine enters user mode (if appropriate) and invokes the user's .TIMER interrupt routine via a JMS instruction. If the user's .TIMER interrupt routine returns from the JMS, the auxiliary routine will return to the interrupted code with a DBR instruction, so that all operating modes will be restored correctly.

This raises one final point. Although auxiliary routines and RQ.LV4 are primarily useful with API, they are also used without API as a coding convenience. Recall that RQ.LV4 may only be called from an interrupt routine. When API is not in use, RQ.LV4 will schedule the auxiliary routine for execution immediately following termination of the interrupt routine which called RQ.LV4. The non-API variation of RQ.LV4 is depicted by Figure 3-3.

3.3 REAL TIME CLOCK OPERATION

The XVM/DOS resident monitor uses the KW15 real time clock to maintain timing information and to schedule various routines. The clock interrupt routine, CK.INT, is depicted in Figure 3-4 as is the clock startup routine, CK.STRT. The clock startup routine is used to restart the clock following a CAF instruction.

Several .SCOM locations are used in conjunction with the clock routines for timing purposes. These .SCOM locations are described in Table 3-1. Note that .SCOM locations should only be referenced by equating their mnemonic to their location, and then using the mnemonic as the actual address. Following this convention facilitates readability and future alteration to .SCOM.

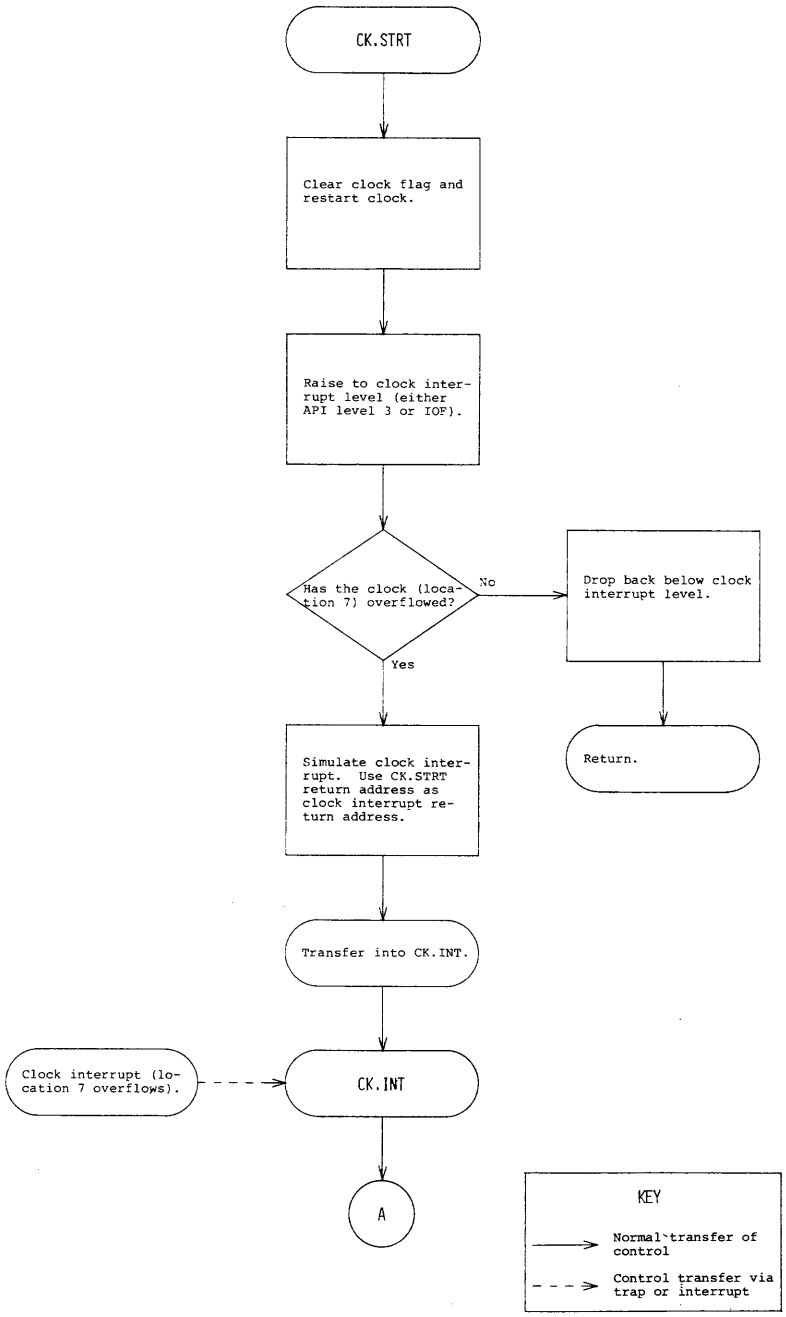


Figure 3-4
Real Time Clock Routines

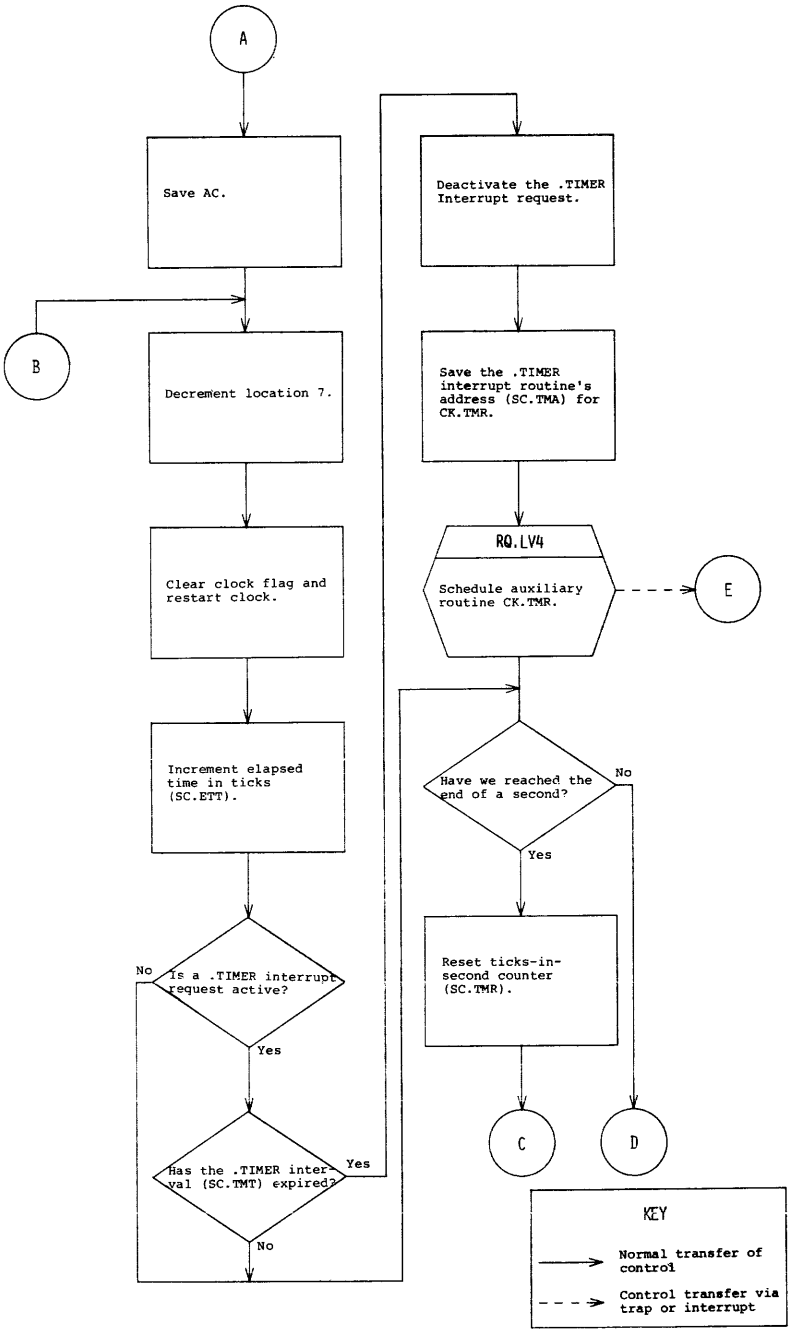


Figure 3-4 (cont)
Real Time Clock Routines

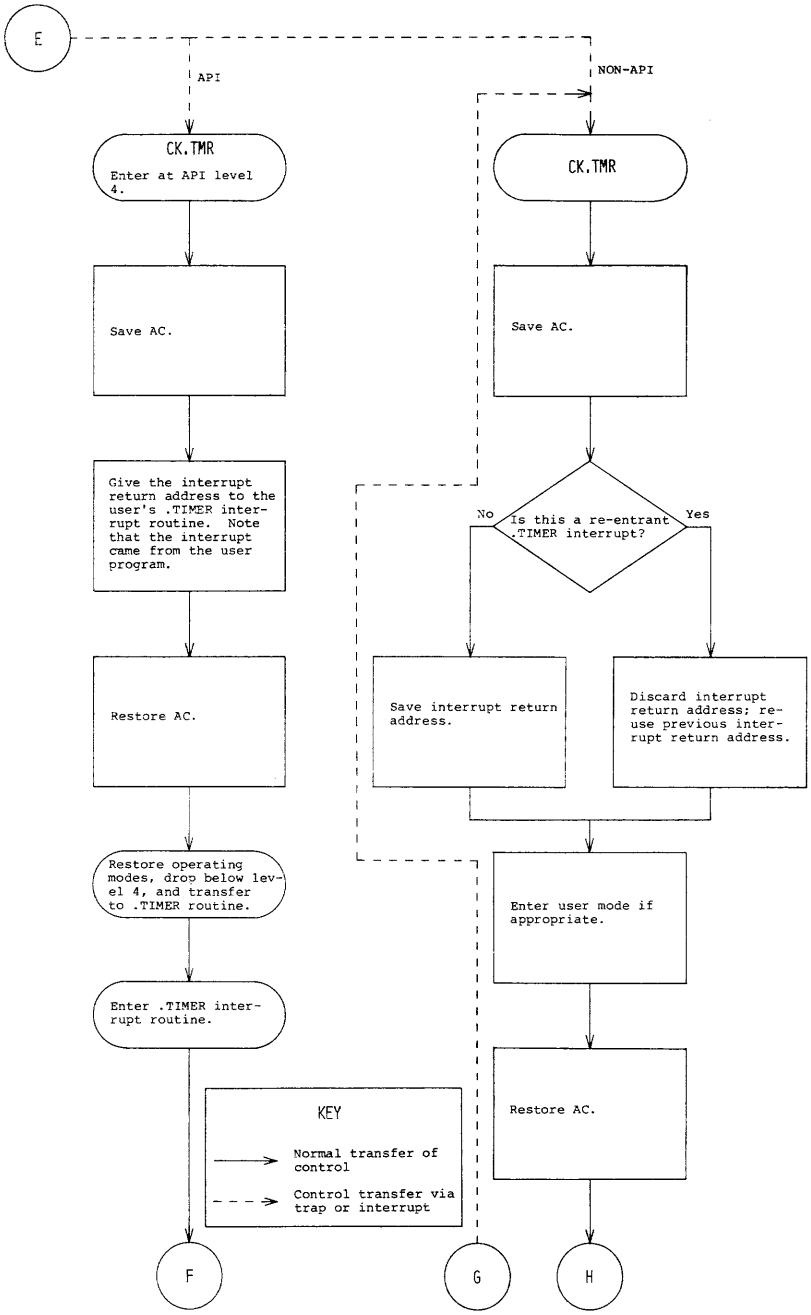


Figure 3-4 (cont)
Real Time Clock Routines

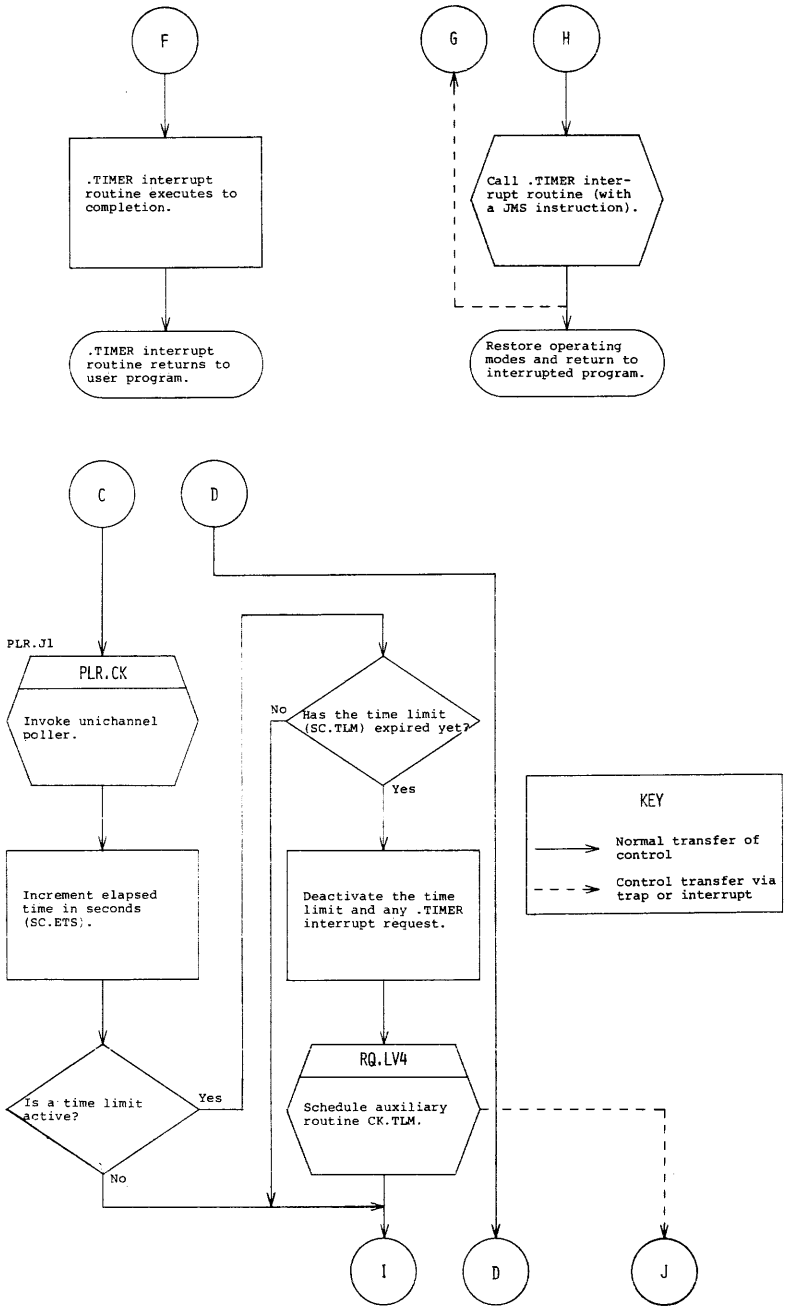


Figure 3-4 (cont)
Real Time Clock Routines

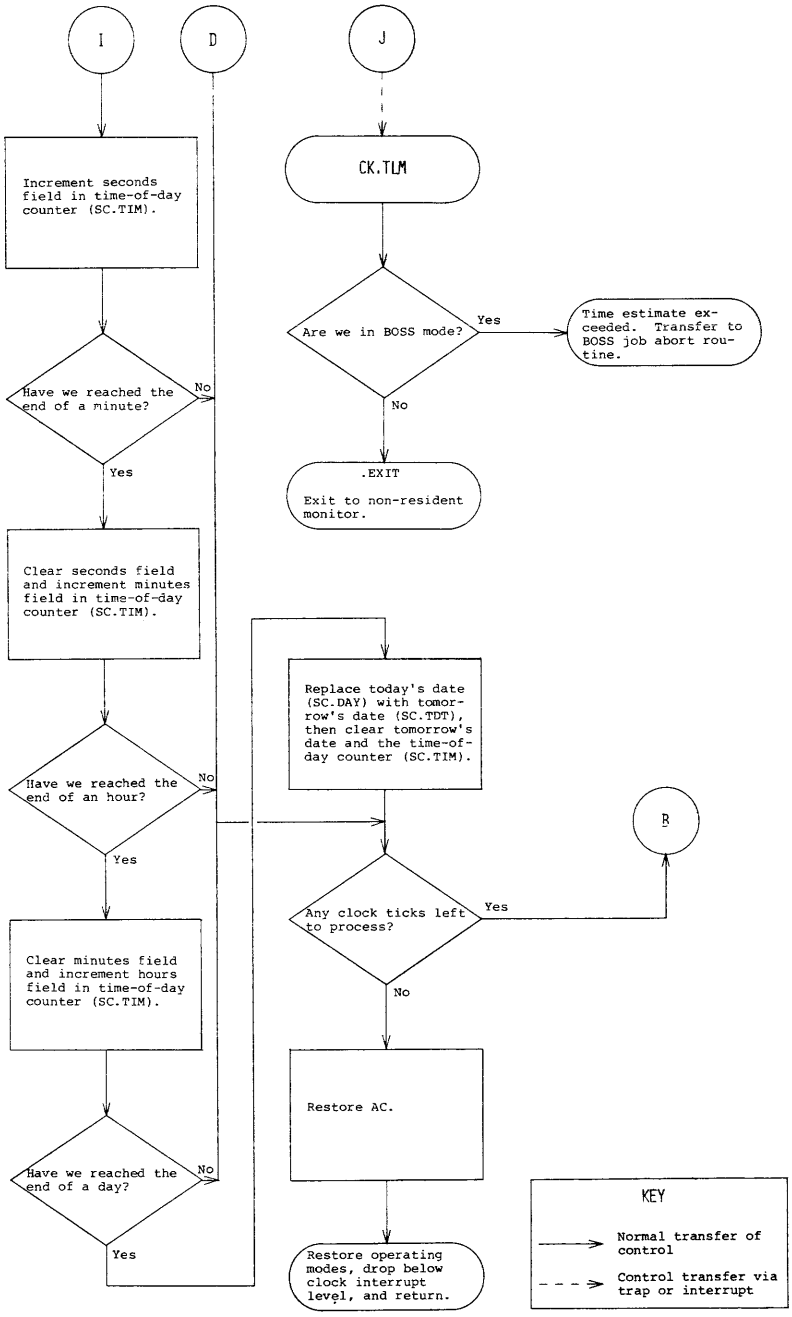


Figure 3-4 (cont)
Real Time Clock Routines

Table 3-1

Clock Oriented .SCOM Locations

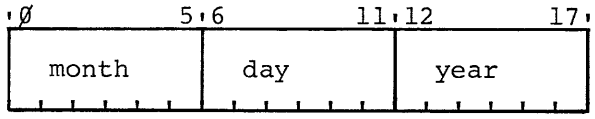
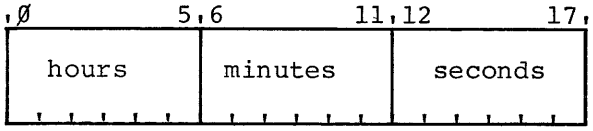
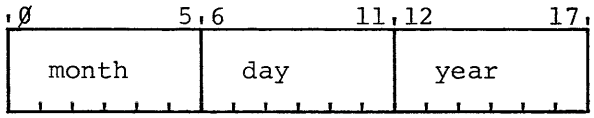
Location	Mnemonic	Description
134	SC.ETS	Elapsed time in seconds. Incremented once per second. Used in BOSS mode for job run time accounting.
147	SC.DAY	Today's date, or zero if the date is unknown. Format:  where month is the calendar month (one corresponds to January), day is the date within the month, and year is years since 1970 (one corresponds to 1971).
150	SC.TIM	The current time of day, kept as a 24 hour clock. Format:  where hours, minutes, and seconds express the time since midnight. Midnight is denoted by all fields containing zero. Note that this location must be initialized via the TIME command for the time of day to be accurate.
151	SC.ETT	Elapsed time in ticks. Incremented at the power line frequency. This location may be modified by the user, as it is not used by system programs.
156	SC.TLM	Two's complement of the number of seconds remaining in the current time limit, or zero if no time limit in effect. Time limits are requested with the TIMEST command or as part of BOSS's $JOB card.
160	SC.TMT	Two's complement of the number of clock ticks (at the power line frequency) remaining until the current .TIMER interrupt occurs, or zero if no .TIMER interrupt request is active.

Table 3-1 (cont.)
Clock Oriented .SCOM Locations

Location	Mnemonic	Description
161	SC.TMA	The address of the .TIMER interrupt routine associated with the current .TIMER interrupt request.
171	SC.TDT	Tomorrow's date, or zero if tomorrow's date is either unknown or not yet determined. Format:  where month is the calendar month (one corresponds to January), day is the date within the month, and year is years since 1970 (one corresponds to 1971). This location is cleared (i.e. tomorrow's date becomes undetermined) at midnight and is reinitialized the next time the non-resident monitor is loaded.
173	SC.TMR	Two's complement of the number of clock ticks (at the power line frequency) left in this second.
174	SC.LFR	Two's complement of the power line frequency -- i.e. the number of clock ticks in each second.

3.4 THE UNICHANNEL POLLER

The UNICHANNEL poller provides a mechanism for reporting to the user/operator any error status conditions which may arise in the UNICHANNEL processor. Most of these are asychronous conditions not directly related to any active XVM I/O requests. The poller, driven by the clock, periodically polls the UNICHANNEL with an error status report software directive to discover if any of these errors have occurred.

The poller is triggered once per second by the clock interrupt routine. Barring the occurrence of certain delays, this means that the UNICHANNEL will be polled every second. One delay which can occur is slow response from the UNICHANNEL. Within reasonable limits this causes no problem. However, if the delay exceeds a preset timeout a separate error (described below) is reported. The other delay which might occur involves the state of the console terminal. UNICHANNEL poller errors are reported on the console terminal printer. If the console terminal printer is active, the poller will be deferred until it becomes idle. The

printer is considered active when either output is in progress or input is in progress and one or more characters have been typed. The printer becomes idle when the current .WRITE or .READ completes, which is usually at the end of the current line.

When operating normally, the poller always has an error status report software directive request outstanding with the UNICHANNEL. When triggered, the poller checks if the directive has completed and exits (via a timeout check) if the directive is still in progress. Assuming the directive has completed, the poller scans for and reports any error conditions. Subsequently the poller requests a new error status report software directive for the next time it is invoked.

Throughout its operation the poller places certain timeouts upon its UNICHANNEL interactions. Although not available as assembly parameters, there are several assembly constants which determine the lengths of the timeouts. These constants should be revised if the timeouts must be changed for any reason.

The poller's primary timeout limits the time the UNICHANNEL has to respond to the error status report software directive. This timeout, controlled by the constant PLR.WT, is normally set at one second. If this timeout expires, the poller will report an IOPSUC PRX 4 error. This error usually implies that the UNICHANNEL processor has halted or is otherwise locked out and unable to respond.

The conditions which cause this timeout to take effect are frequently relatively long lasting. Therefore a frequency counter is used to avoid excessive repetition of the IOPSUC PRX 4 error message. This frequency counter is controlled by the constant PLR.FR, and will normally cause the IOPSUC PRX 4 message to repeat every five minutes. Note that the frequency counter only has effect if the UNICHANNEL processor lock out condition is continuously maintained. If the condition occurs, clears itself, and then re-occurs, the IOPSUC PRX 4 message will be immediately re-issued.

In addition to the primary timeout described above, the poller implements a second timeout. If the UNICHANNEL is unprepared to accept a TCB request when the poller attempts to issue an error status report software directive, an IOPSUC PRX 4 error is again reported. The poller implements this by neglecting to issue the request to the UNICHANNEL, so that eventually the primary timeout above expires. When the poller

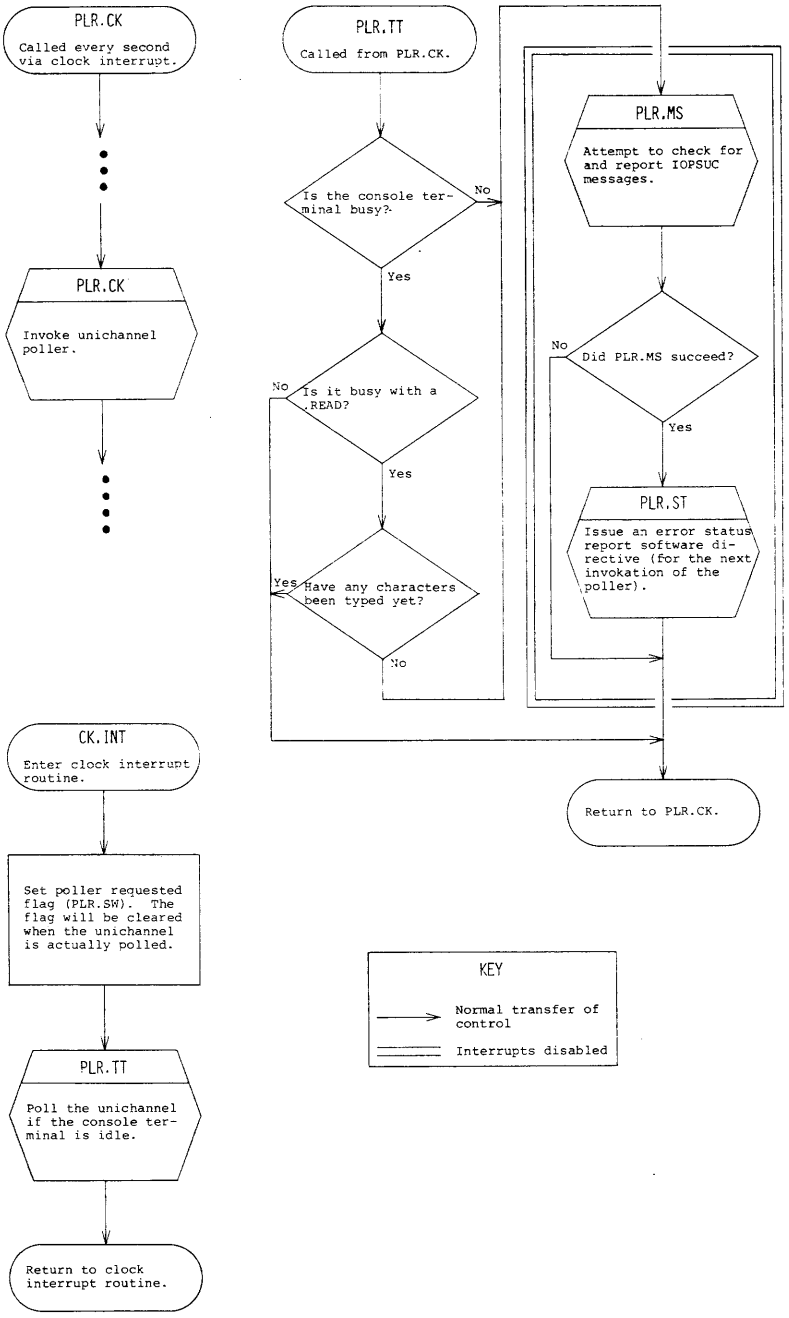


Figure 3-5
The Unichannel Poller

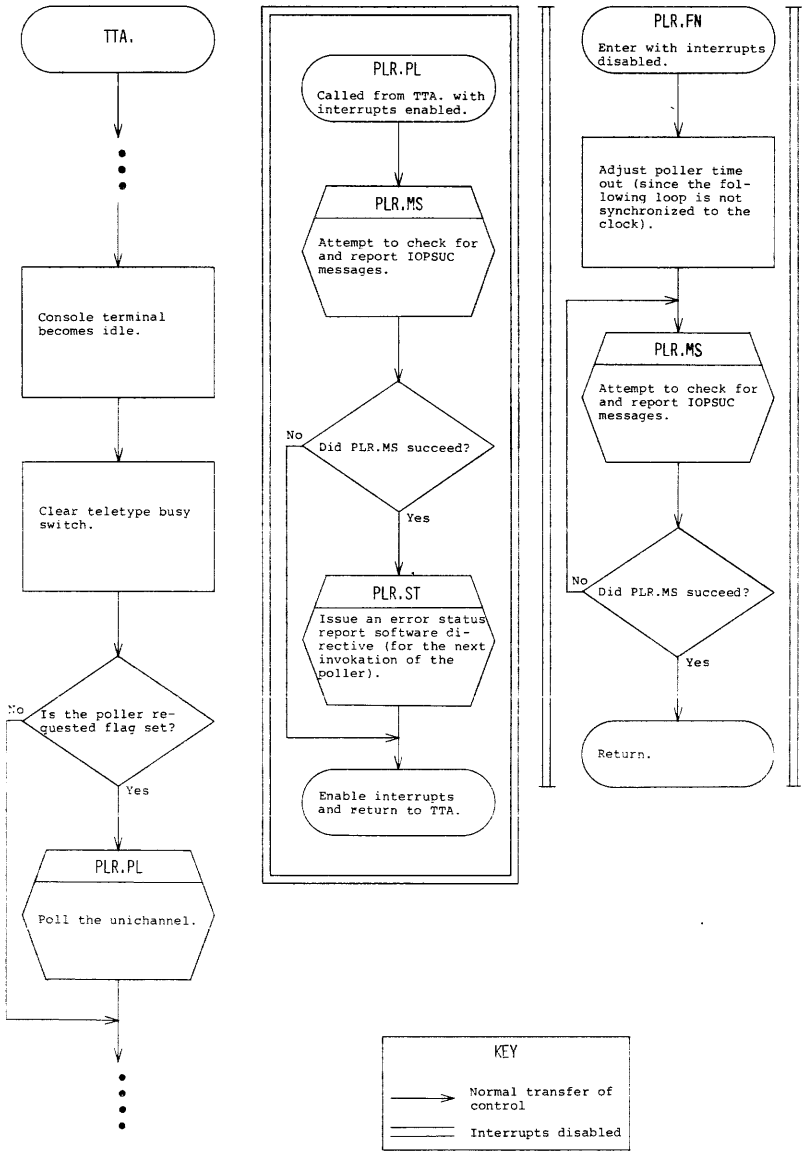


Figure 3-5 (cont)
The Unichannel Poller

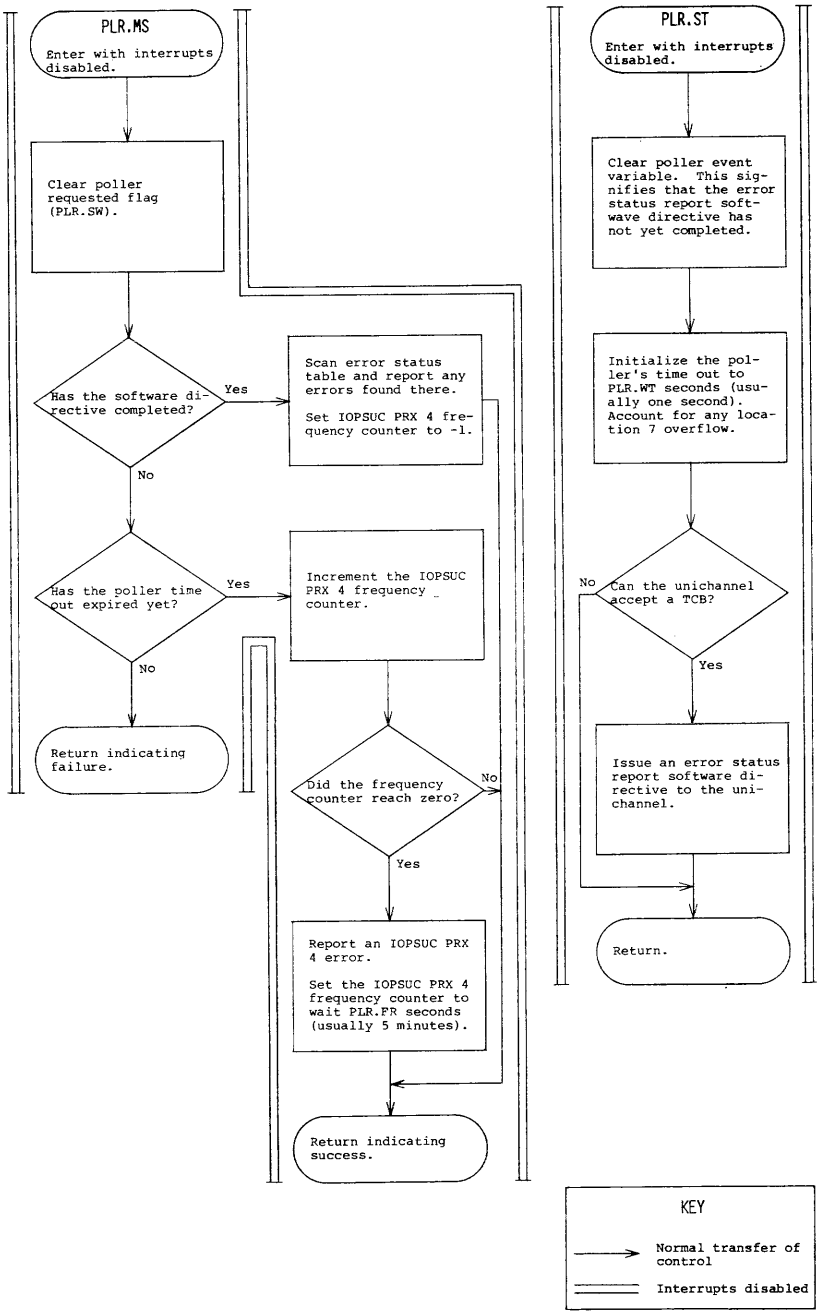


Figure 3-5 (cont)
The Unichannel Poller

issues its request, it has been running for some time with interrupts disabled, so that the UNICHANNEL processor has had a reasonable length of time to accept any earlier requests.

The poller is implemented as a monitor appendage, so that it will only be included in the monitor when its operation has been enabled. It consists of the routines depicted in Figure 3-5. PLR.CK is called once per second from the clock interrupt routine. PLR.TT checks to see if the console terminal printer is active or idle. If the printer is active, PLR.PL will be called from TTA. when the printer next becomes idle. PLR.MS and PLR.ST together perform the polling function proper. PLR.ST is also used to initiate poller operation. PLR.FN is used to terminate poller operation. The poller must not be operating whenever a CAF instruction is executed. Note that the poller interacts with many other parts of the monitor, and thus much of the poller executes with interrupts disabled.

3.5 XVM MODE

The XVM/DOS resident monitor implements XVM mode (17-bit indirect addressing mode) by equating it to user mode. When running on an XVM, the monitor initialization code sets the relocate disable and 17-bit addressing flags. These flags are in the XM15 hardware option's MM register. The flags are also set whenever the monitor executes a CAF instruction.

With these flags set, the monitor turns XVM mode on and off by switching between user and exec modes. Thus the .XVMON and .XVMOFF CAL's (Section 2.7) merely enter and leave user mode. User programs should not make use of this fact, however, as the detailed implementation of XVM mode may change with subsequent releases of XVM/DOS.

CHAPTER 4
THE NONRESIDENT MONITOR

4.1 INTRODUCTION

The System Loader brings the Nonresident Monitor into core after a hardware readin, a manual restart, a CTRL C, or a .EXIT. SGNBLK, SYSBLK and COMBLK are always coresident with the Nonresident Monitor. This gives the Nonresident Monitor access to all important system parameters.

The Nonresident Monitor announces its presence by typing XVM/DOS Vnxnnn on the teleprinter. It remains in core until the operator requests another system program, or until the operator's command implies a refreshed configuration of the Resident Monitor is necessary.

The Nonresident Monitor's actions are limited to (1) decoding commands, (2) manipulating or examining bits and registers in .SCOM, .DAT, .UFDT, SYSBLK, COMBLK, and SGNBLK, and (3) calling the System Loader, when necessary. The Nonresident Monitor has only one entry, which starts an initialization section. Figure 4-1, Nonresident Monitor Initialization, describes that logic. Every time the System Loader brings in the Nonresident Monitor, it passes control to the initialization section. After initialization, and after all commands that do not require the System Loader, the Nonresident Monitor types a $ and awaits an input line, terminated by a Carriage RETURN or an ALT MODE. It then examines the first six characters (or those up to the first blank) and tries to find an entry in the Nonresident Monitor's Command Table. If a match is found, control passes to the appropriate routine, and thence to the next command or the System Loader. If the typed command does not correspond to an entry in the command table, the Nonresident Monitor temporarily assumes the operator wishes a new core-image system program and checks COMBLK for a corresponding entry. If there is no corresponding entry in COMBLK, the Nonresident Monitor will type an error message and await the next command. If COMBLK contains a matching entry, the Nonresident Monitor composes a .OVRLA and passes control to the System Loader via that .OVRLA.

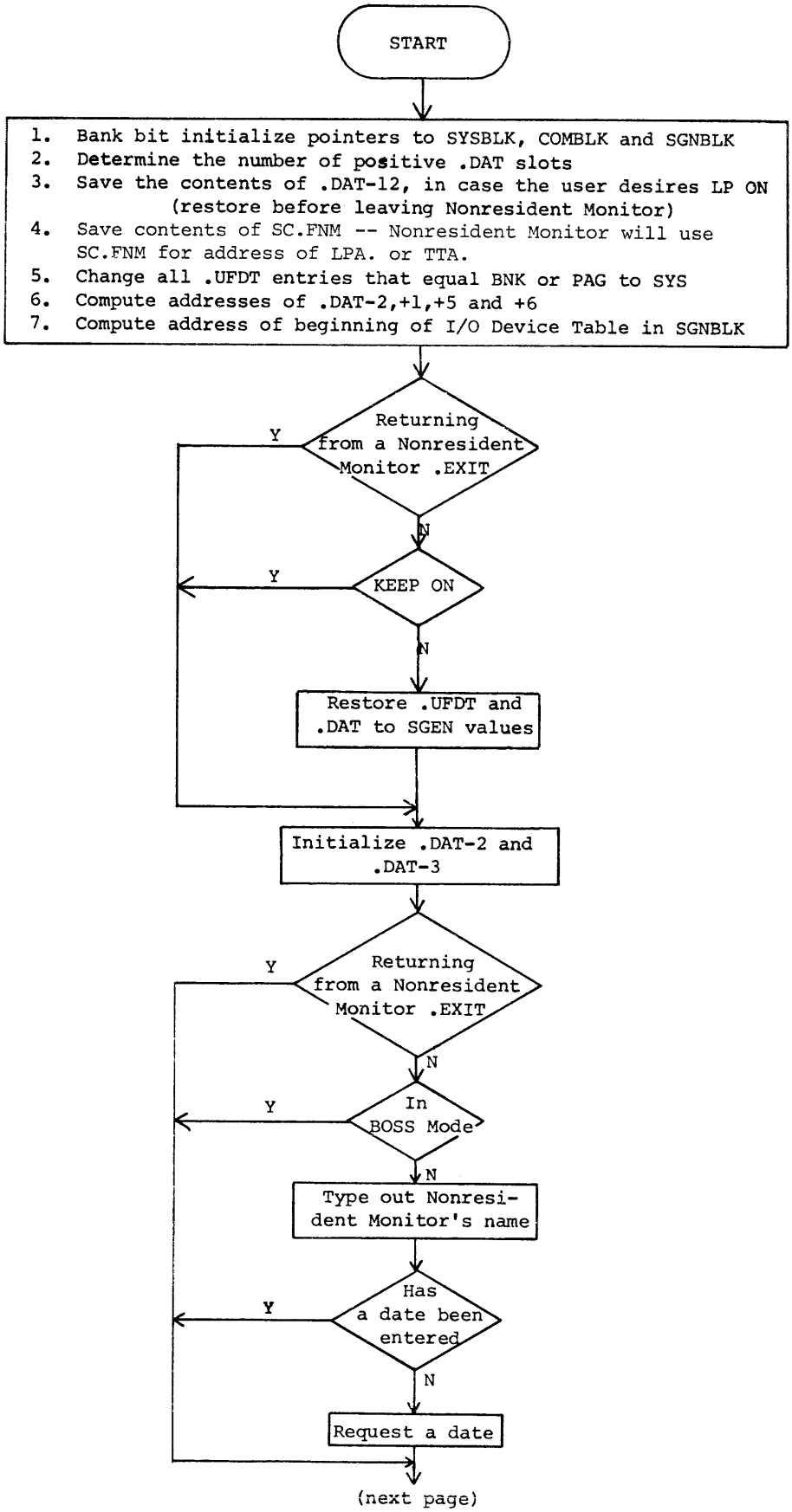
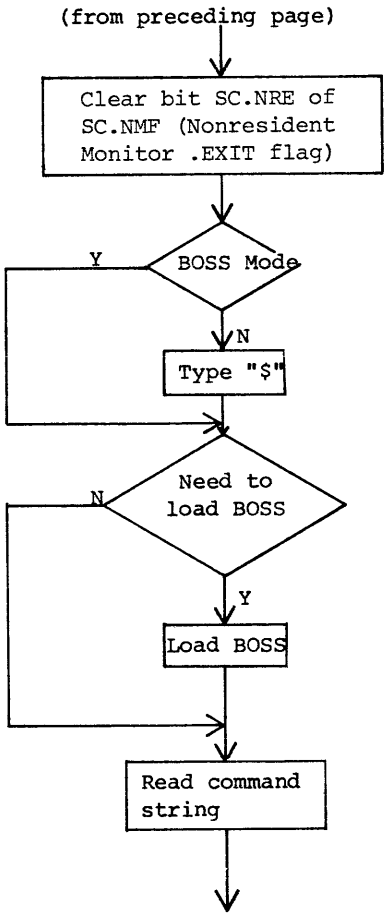


Figure 4-1
Nonresident Monitor Initialization



(Continue to Command Decoder)

Figure 4-J (Cont.)
Nonresident Monitor Initialization

4.2 COMMANDS TO THE NONRESIDENT MONITOR

This paragraph discusses legal commands listed in the Nonresident Monitor's Command Table. Table 4-1, Effects and Exits for Nonresident Monitor Commands, describes all commands that do not request a new program.

There are five entries in the Command Table that load relocatable system programs. They are INSTRUCT, DDT, EXECUTE, GLOAD and LOAD. The Nonresident Monitor treats these commands separately, because SYSBLK does not list them. All information necessary for loading these programs resides in the Nonresident Monitor itself.

4.3 CONSIDERATIONS FOR ADDITIONS TO THE NONRESIDENT MONITOR

Programmers should not attempt to add commands to the Nonresident Monitor unless they have access to a copy of the source code. The source code is available as an integral part of the XVM/DOS release. They should then use the EDITOR program to put in the indicated changes, and reassemble.

New additions to the Nonresident Monitor require the following actions:

1. Update the Nonresident Monitor's Command Table.
   The Command Table is in two parts:
   a) The .SIXBT names of the commands
   b) The corresponding transfer vector
2. Write the code for the command.
3. Consider the kind of exit the command will take:
   a) Commands that end with a request for a new command should end with JMP KLCOM
   b) Commands that re-configure the Nonresident Monitor should end with JMP NRMEX1.
4. Determine if the command should result in an automatic MODE typeout. If an automatic mode typeout is desired, a JMS SETMD should be included in the code for the command and a JMP NRMEXI exit should be used.

Table 4-1

Effects and Exits
for Nonresident Monitor Commands[1]

COMMAND	MODIFIER	ACTION TAKEN	EXIT
API	ON OFF	Set bit SC.API of SC.MOD Clear bit SC.API of SC.MOD	.EXIT .EXIT
ASSIGN	handler (and/or) uic	Check whether handler is available. if yes, load .DAT slot with proper handler code. (The proper loader will load the handler, and insert its starting address into the .DAT slot. Load proper slot via a .USER	Next Command Next Command
BANK	ON OFF	Set bit SC.BNK of SC.MOD Clear bit SC.BNK of SC.MOD	Next Command
BATCH	SY RK DP DK PR CD MT	See Chapter 9.	
BUFFS	number	Put number indicated into SC.BNM, and set Nonresident Monitor Initialization to leave SC.BNM alone.	Next Command
CHANNEL	7 9	Clear bit SC.9CH of SC.MOD Set bit SC.9CH of SC.MOD	Next Command
DATE	date no date	Enter date into SC.DAY Print date from SC.DAY	Next Command
FILL	ON OFF	Set bit SC.FIL of SC.MOD Clear bit SC.FIL of SC.MOD	.EXIT .EXIT
GET GETP GETS GETT		See Section 2.8.	
HALF	ON OFF	Set bit SC.HFN of SC.VTF Clear bit SC.HFN of SC.VTF	.EXIT .EXIT
HALT		Set bit SC.HLT of SC.NMF	Next Command

[1]This table assumes error-free input

Table 4-1 (cont)

Effects and Exits
for Nonresident Monitor Commands

COMMAND	MODIFIER	ACTION TAKEN	EXIT
INSTRUCT	none ERRORS	Print INSALL SRC } By loading Print INSERR SRC } INSTRC BIN	.EXIT Command
KEEP	ON OFF	Set bit SC.KPN of SC.NMF Clear bit SC.KPN of .SC.NMF. Initialize to SGEN default values all entries in .DAT and .UFDT, except change SCR default values to current UIC.	Next Command
LOG		Output five spaces after Carriage RETURNs. After ALT MODE, go to next command.	Next Command (after ALT MODE)
LOGIN	uic	Make specified UIC current (SC.UIC). *Then set up .UFD entries; set .DAT entries and system parameters (SC.MOD, SC.MSZ, SC.BNM, SC.VTF) to system default values; clear SC.NMF and SC.TLM.	.EXIT
LOGOUT		Set current UIC to SCR. Then same as LOGIN (above) from *.	.EXIT
LOGW		For BOSS-15, print message. In all cases, after a Carriage RETURN, output five spaces. After ALT MODE, type four bells ↑P, and await CTRL P. After CTRL P, go to next command.	Next command (after ALT MODE)
LP	ON OFF	Set bit SC.LPON of SC.NMF. Clear bit SC.LPON of SC.NMF.	.EXIT .EXIT
MEMSIZ	nnnk	Check nnn for multiple of 8 and range $24 \leq nnn \leq 128$, if valid (nnn* 1024)-1 is stored in SC.MSZ.	.EXIT
MICLOG	mic	Check mic with SGNBLK. If correct set bit SC.MIC of SC.NMF and make 'SYS' the current UIC. Then same as LOGIN (above) from * (except SC.NMF not cleared). If incorrect, ignore command.	.EXIT
MODE		Information pertaining to available hardware (SC.NMF) is compared with requested parameters and the resulting parameter settings are typed to the console for those parameters related to the existing hardware.	Next Command
PAGE	ON OFF	Clear bit SC.BNK of SC.MOD. Set bit SC.BNK of SC.MOD.	.EXIT

Table 4-1 (cont)

Effects and Exits
for Nonresident Monitor Commands

COMMAND	MODIFIER	ACTION TAKEN	EXIT
POLLER	ON OFF	Clear bit SC.PLR of SC.MOD. Set bit SC.PLR of SC.MOD.	.EXIT
PROTECT	n	If n is between Ø and 7, inclusive, enter it into SC.PRC.	Next Command
PUT		See Section 2.8.	
QDUMP		Set bit SC.DMP of SC.NMF.	Next Command
REQUEST	none USER prog	Print the current assignments for .DAT and .UFDT. Print the current assignments for all positive .DAT and .UFDT slots. Print required .DAT and .UFDT slots, and the assignments and use for each.	Next Command
SCOM		Print the information for the current system.	Next Command
TAB	ON OFF	Clear bit SC.TAB of SC.MOD. Set bit SC.TAB of SC.MOD.	.EXIT .EXIT
TIME	time none	Enter time into SC.TIM. Print time from SC.TIM.	Next Command
UC15	ON OFF	Bit SC.UC15 of SC.MOD is set indicating a request for UNICHANNEL interaction capability. (Note: the resident monitor verifies the validity of this request.) Bit SC.UC15 of SC.MOD is cleared indicating that no UNICHANNEL interaction is desired. (The resident monitor will not allow UC15 OFF to succeed on an RK based system.)	.EXIT .EXIT
VT	ON OFF	Set bit SC.VTN of SC.VTF. Clear bit SC.VTN of SC.VTF.	.EXIT .EXIT
XVM	ON OFF	Bit SC.XVM of SC.MOD is set indicating a request for wide addressing mode. (Note: the resident monitor verifies the validity of this request.) Bit SC.XVM of SC.MOD is cleared indicating that 15-bit indirect addressing mode is desired.	.EXIT .EXIT

5. After assembly, the programmer must call PATCH, in order to make his relocatable binary program absolute. Commands to PATCH should be as follows:

```
>DOS15
>READR 16Ø77 DOSNRM BIN
```

16Ø77 indicates the highest location the new monitor can occupy. (SYSBLK begins at 161ØØ.) DOSNRM BIN happens to be the file name used by program development. The programmer may, of course, substitute his own file name. More information may be found in the PATCH manual.

CHAPTER 5
THE SYSTEM LOADER

The System Loader is the third major part of the XVM/DOS operating system. The other two are the Resident and Nonresident monitors. The Resident and Nonresident Monitors communicate with the System Loader by manipulating certain .SCOM registers. When commands to either part imply a new configuration is needed, that part sets up the appropriate .SCOM registers and passes control to the resident monitor via a .EXIT or .OVRLA. The resident monitor rebuilds itself, loads the System Loader into high core, and gives it control.

The System Loader examines the .SCOM registers then loads the desired system program and all handlers required by the new configuration. In addition, it will allocate all required buffers. The Nonresident Monitor is treated like any other core-image system program.

The System Loader never loads user programs. It only loads core-image system programs, the INSTRUCT command processing program, the Linking Loader and Execute. The latter two load user programs.

The System Loader uses two device handlers to interface with the disk: the System Bootstrap, and the System Loader Disk Handler (DKL./DPL./RKL.). xxL. arrives in core as part of the loader itself. The Bootstrap loads core-image programs only. The xxL handler takes care of relocatable programs and any handlers loaded by the System Loader. Those include all handlers for core-image system programs, the Linking Loader's own handlers, and any needed by the Execute file. The Linking Loader loads handlers needed by user programs it links.

5.1 LOADING SYSTEM PROGRAMS

The System Loader gets control in the highest bank. The System Loader loads handlers from the lowest part of free core up. Core-image system programs are usually loaded just beneath the Bootstrap. Such core images must be wholely within the bank in which the Bootstrap resides, and above register 17 of that bank. Figure 5-1 illustrates the core maps for system programs.

Whenever the Linking Loader is loaded (LOAD, GLOAD, DDT, and DDTNS), the System Loader loads all handlers for .DAT slots -1, -4, and -5, and then loads the Linking Loader itself. (DDT is loaded by the Linking Loader.) Whenever INSTRC (the INSTRUCT command processing program) is loaded, the handler assigned to .DAT slot -12 is also loaded. Figure 5-2 illustrates the core maps for the Linking Loader and INSTRC.

For EXECUTE, the System Loader loads EXECUTE's handler, and reads the EXECUTE file, in order to determine the active .DAT slots. The System Loader then loads all the handlers required and sets up the .DAT slots. Figure 5-3 illustrates core maps for EXECUTE.

BOSS XVM mode operation requires the system "A" handler be assigned to .DAT-7. This requires a slight of hand on the part of the System Loader, which needs the "L" handler on .DAT-7. It therefore loads the "A" handler as if it were assigned to .DAT+Ø, and transfers the set up .DAT+Ø contents to .DAT-7 before transferring control to the program being loaded. .DAT+Ø is then restored to its original status.

5.2 TABLES AND INFORMATION BLOCKS USED AND BUILT BY LOADERS

The System Loader uses SYSBLK, COMBLK, SGNBLK, .SCOM, the Mass Storage Busy Table, the File Buffers Transfer Vector Table, the Overlay Table, .DAT, and .UFD. Tables 5-1 and 5-2 describe how the Loaders use these blocks and tables.

5.3 .DAT SLOT MANIPULATION BY THE SYSTEM LOADER

When loading core-image system programs, the System Loader determines the active .DAT slots by examining COMBLK. When loading EXECUT, the System Loader sets up .DAT-4, and any active slots indicated by the Execute file itself. When loading the Linking Loader, the System Loader sets up .DAT-1, -4, and -5 and also .DAT-12, if loading INSTRC. The Linking Loader will set up other active .DAT slots according to the .IODEV commands in the assembly of the program units being loaded.

Both the System Loader and the Linking Loader set up .DAT slots in this manner: (In the following procedure, "loader" refers to either one.)

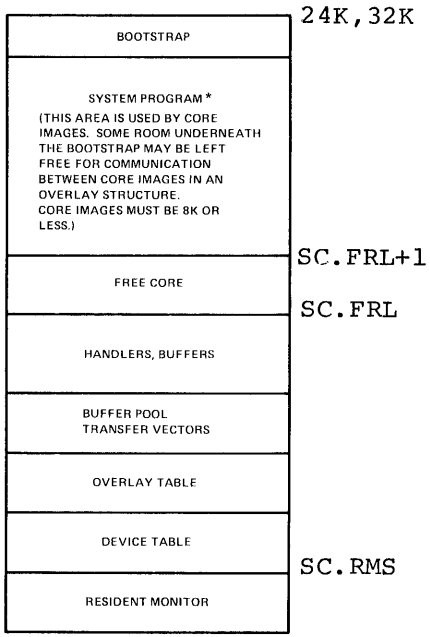
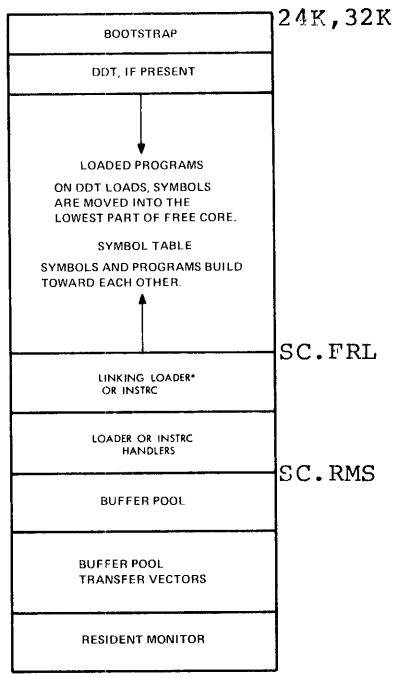


Figure 5-1

System
Program Load

*All system programs except MAC11, which is always loaded in Bank 1.



Placement of SC.FRL depends on relative positions of the Linking Loader and its handlers. When control is transferred to loaded program SC.FRL and SC.FRL+1 bracket free core.

Figure 5-2

Linking Loader

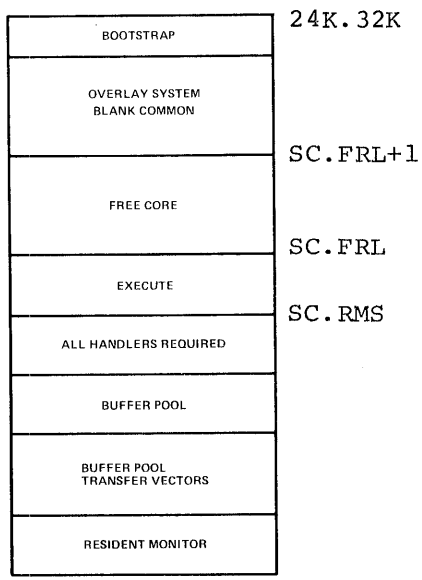


Figure 5-3

Execute

Table 5-1

Tables and Blocks Used by the Loaders

NAME	USE	LOCATION
SYSBLK	The System Loader obtains Monitor TRAN parameters from SYSBLK when it builds	165ØØ of .SYSLD's bank
COMBLK	Indicates number of buffers required, the active .DAT slots, and the names	171ØØ down, in .SYSLD's bank
SGNBLK	Number of words per buffer and handler information.	161ØØ of .SYSLD's bank
.SCOM Table	See Table 5-2	1ØØ of 1st bank
Mass Storage Busy Table	Built by the System Loader itself.	Pointed to by SC.BTA
File Buffers Transfer Vector Table	Built by the System Loader itself. System Loader proper initializes for core-image programs.	Pointed to by SC.BTB
Overlay Table	Built by the System Loader itself.	Pointed to by SC.OTB
.DAT and .UFDT	The System Loader loads all handlers for core-image programs and and EXECUTE Files, and sets up the appropriate .DAT slots. The System Loader also loads handlers assigned to .DAT-1, -4, and -5 when loading the Linking Loader, and .DAT-7 and +6 for BOSS XVM.	Pointed to by SC.DAT and SC.UFD

Table 5-2

.SCOM Registers Used by the System Loader

Location	Description of Use by the System Loader
SC.FRL	.SYSLD continually updates this indication of the first free location as it moves code and builds tables.
SC.FRL+1	Updated as with SC.FRL. Last free location in core below the Bootstrap.
SC.SST	.SYSLD uses SC.SST for the starting address when loading EXECUT or LOAD. The Bootstrap transfers to the address in SC.SST after all its operations.
SC.UST	Stores codes for DDT, DDTNS, LOAD and GLOAD.
SC.FNM	.SYSLD saves contents of .DAT-1 in SC.FNM, when loading the Linking Loader. When loading EXECUT, SC.FNM contains the first three-characters of the Exedute file's name. Contains .DAT-12 when loading Nonresident Monitor.
SC.FNM+1	.SYSLD saves contents of .DAT-4 in SC.FNM+1, when loading the Linking Loader. When loading EXECUT, SC.FNM+1 contains the second three characters of the Execute file's name.
SC.FNM+2	.SYSLD saves contents of .DAT-5 in SC.FNM+2, when loading the Linking Loader. When loading EXECUT, SC.FNM+2 contains the extension of the Execute file's name.
SC.BNM	When the Nonresident Monitor was the last program, the System Loader allocates the number of buffers indicated by the contents of SC.BNM. If the Nonresident Monitor was not the last program, the System Loader restores SC.BNM to the default value if program to be loaded is core image. Otherwise, untouched.
SC.BLN	The number of words per file buffer.
SC.BTB	Pointer to the File Buffer Transfer Vector Table.
SC.OTB	When loading a core-image program, .SYSLD loads SC.OTB with the pointer to the Overlay Table, or with zero, if there is none.
SC.SPN	Contains name of the program to be loaded.
SC.ACT	System Loader loads with the number of entries in the Mass Storage Busy Table.
SC.BTA	System Loader loads with the address of the first entry in the Mass Storage Busy Table.

1. Each .DAT slot will contain a handler number -- either the system default, or one inserted via an ASSIGN command to the Nonresident Monitor. This handler number is the relative index of the handler name in the handler table portion of SGNBLK.

2. For each active .DAT slot, the loader uses the handler number in that slot to find the name in the handler table, and converts the name to .SIXBT.

3. If the handler is already in core, the loader simply inserts the starting address of the handler into the .DAT slot.

4. If the handler is not yet in core, the loader does a .SEEK to <IOS> UIC for the handler, reads it into core, relocates it, and places the starting address of the handler into the .DAT slot.

The System Loader always sets up .DAT-2 and -3. (It reserves .DAT-7 for its own use.) When not in non-BOSS Batch Mode, -2 is assigned to TTA. In non-BOSS Batch Mode, the batch input device goes to -2. If loading the Nonresident Monitor and bit SC.LPON of SC.NMF is set, the System Loader will set up .DAT-12 for LPA, if it is in the system, or else for TTA. If in BOSS mode, the Nonresident Monitor assigns LPA. to .DAT+6, and the System Loader assigns .DAT-7 to the system device "A" handler. The System Loader then ensures that both handlers are in core. The Resident BOSS setup routine subsequently routes all .DAT slots connected to TTA. to Resident BOSS.

5.4 BUFFER ALLOCATION BY THE SYSTEM LOADER

The System Loader allocates space for buffers equal to the contents of SC.BNM times the contents of SC.BLN. The first time initialization routine sets SC.BLN to the standard number of locations per buffer. Before the Nonresident Monitor does an .OVRLA to a software system program, it checks whether a BUFFS command has been issued. If so, it leaves SC.BNM as is. If not, it uses the default number of buffers for that program, as shown in SYSBLK.

CHAPTER 6
SYSTEM INFORMATION BLOCKS AND TABLES

6.1 SYSTEM COMMUNICATION (.SCOM) TABLE

The system communication or .SCOM table is at a fixed location in memory. It is the primary means of communication between the components of XVM/DOS. Individual .SCOM locations may either be reconstructed for each core load or preserved across successive core loads. Table 6-1 describes the .SCOM table.

The accompanying table defines mnemonics for each .SCOM location, and separate mnemonics for any bit flags. These mnemonics should be used for all references to .SCOM. The mnemonics should be equated to the appropriate values and then used whenever .SCOM is accessed to make the program easier to read and .SCOM easier to alter in the future.

6.2 DISK-RESIDENT UNCHANGING BLOCKS: SYSBLK, COMBLK AND SGNBLK

SYSBLK, COMBLK and SGNBLK occupy blocks 34, 35, and 36 (octal) on the system device(unit zero). SYSBLK and COMBLK (blocks 34 and 35) contain the parameters for loading all core image system programs. SGNBLK contains all the other information needed to run DOS. All three arrive in core along with the Resident Monitor and start at location 161ØØ of the highest bank. The Nonresident Monitor and System Loader use them; SGEN XVM and PATCH XVM modify them when necessary.

6.2.1 SYSBLK

SYSBLK contains the parameters required for implementation of .OVRLA to any system program, or any of the system program overlays.

The order of entries in SYSBLK is unimportant, except for the first three permanent entries: RESMON, .SYSLD, and ↑QAREA. The first word of SYSBLK contains the block address (the unrelocated address) of the first free word after itself. Figure 6-1 describes SYSBLK.

Table 6-1

System Communication (.SCOM) Table

Location	Mnemonic	Description
100	SC.COD	First free register below bootstrap; the highest location useable for code.
101	SC.RMS	Resident Monitor size; first free register above Resident Monitor.
102 103	SC.FRL SC.FRL+1	Free memory in low core (below the bootstrap). SC.FRL contains the address of the first free location. SC.FRL+1 contains the address of the last free location.
104	SC.MOD	Operating mode bit register:
	SC.API = 400000_8	1 → API enabled
	SC.TAB = 100000_8	1 → Simulate tabs with spaces
	SC.NRM = 40000_8	1 → Non-resident monitor in core
	SC.UB2 = 20000_8	Reserved for customer use.
	SC.UB1 = 1000_8	Reserved for customer use.
	SC.9CH = 4000_8	1 → 9-channel magtapes assumed.
	SC.FIL = 200_8	1 → Insert fill characters.
	SC.BNK = 100_8	1 → Bank mode operation.
	SC.LPSZ = 60_8	Line printer line length (in characters): 0 → zero 40 → 120 20 → 80 60 → 132
	SC.PLR = 4_8	1 → Poller enabled.
	SC.UC15 = 2_8	1 → UC15 enabled.
	SC.XVM = 1_8	1 → XVM mode enabled.
105	SC.SST	Core image system program starting address.
106	SC.UST	User program starting address/↑S address.
	SC.DDT = 400000_8	1 → DDT in core.
	SC.GLD = 200000_8	1 → Linking Loader invoked via GLOAD.
	SC.DNS = 100000_8	1 → DDT invoked without symbol table (via DDTNS).
	Bits 3-17 contain the user program starting address, which is also the address of the ↑S control character routine.	

Table 6-1 (cont)

System Communication (.SCOM) Table

Location	Mnemonic	Description
107	SC.FNM	Execute Filename, .GET,.PUT Filename, or linking loader handler indices.
110	SC.FNM+1	
111	SC.FNM+2	
112	SC.LV4	API level 4 software interrupt transfer vector.
113	SC.LV5	API level 5 software interrupt transfer vector.
114	SC.LV6	API level 6 software interrupt transfer vector.
115	SC.LV7	API level 7 software interrupt transfer vector.
116	SC.TTP	Saved PC on control character interrupts.
117	SC.TTA	Reserved.
120	SC.MSZ	System memory size, as set via MEMSIZ command.
121	SC.MTS	Magtape status register.
122	SC.AMS	Actual (physical) memory size.
123	SC.DAT	Address of .DAT table.
124	SC.SLT	Number of positive .DAT slots.
125	SC.UFD	Address of .UFD table.
126	SC.BNM	Number of file buffers.
127	SC.BLN	Number of words per file buffer.
130	SC.BTB	Address of file buffer transfer vector table.
131	SC.OTB	Address of overlay table or zero.
132	SC.BBN	Bad block number for IOPS20 and IOPS21.

Table 6-1 (cont)
System Communication (.SCOM) Table

Location	Mnemonic	Description
133	SC.VTF	VT ON flag register;
	SC.HFN = 400000_8	1 → Half size VT15 buffer (HALF ON).
	SC.VTN = 100000_8	1 → VT ON issued.
	SC.DMN = 1_8	1 → Display mode on.
134	SC.ETS	Elapsed time in seconds.
135	SC.CTT	Instruction to clear teletype busy switch.
136	SC.ACT	Number of active .DAT slots; number of entries in mass storage busy table.
137	SC.EEP	Expanded error processor entry point.
140	SC.EEP+1	JMP to expanded error processor.
141	SC.UIC	Current UIC
142	SC.NMF	Non-resident monitor flag register:
	SC.MIC = 400000_8	1 → MICLOG successful
	SC.NRE = 200000_8	1 → Non-resident monitor .EXIT
	SC.NRO = 100000_8	1 → Non-resident monitor .OVRLA
	SC.LPON = 40000_8	1 → LP ON 0 → LP OFF
	SC.DMP = 20000_8	1 → ↑Q dump on IOPS errors.
	SC.HLT = 10000_8	1 → Halt on IOPS errors.
	SC.TMM = 4000_8	1 → Mode message should be typed.
	SC.PVT = 1000_8	1 → VT15 present.
	SC.PCLK = 400_8	1 → Real time clock present.
	SC.PAPI = 200_8	1 → API present.
	SC.PUC15 = 100_8	1 → UC15 present.
	SC.PXVM = 40_8	1 → XVM present.
	SC.DT6 = 10_8	1 → Set up .DAT+6 (for BOSS).
	SC.DT7 = 4_8	1 → Set up .DAT-7 (for BOSS or Batch)
	SC.KPN = 2_8	1 → KEEP ON 0 → KEEP OFF
	SC.BCH = 1_8	1 → Batch or BOSS mode active.

Table 6-1 (cont)

System Communication (.SCOM) Table

Location	Mnemonic	Description
143 144	SC.SPN SC.SPN+1	.SIXBT name of system program to to loaded.
145 146	SC.NMN SC.NMN+1	.SIXBT name of the non-resident monitor ("DOS15").
147	SC.DAY	Today's date, formatted as MMDDYY.
150	SC.TIM	The current time, formatted as HHMMSS.
151	SC.ETT	Elapsed time in ticks.
152	SC.BOS	BOSS bit register:
	SC.BMD = 400000_8	1 → BOSS mode active.
	SC.BCR = 200000_8	1 → Control card read by user.
	SC.BEOF = 100000_8	1 → EOF reached on run time file.
	SC.BTM = 40000_8	1 → Time estimate exceeded.
	SC.BTT = 20000_8	1 → I/O CAL to go to TTA.
	SC.BIO = 10000_8	1 → Terminal IOPS error.
	SC.BDMP = 4000_8	1 → Give user ↑Q dump on IOPS errors.
	SC.BOA = 2000_8	1 → Operator abort (↑T)
	SC.BJA = 1000_8	1 → Job active.
	SC.BXT = 400_8	1 → Exit from BOSS mode.
	SC.BPT = 200_8	1 → User tried to do a .PUT
	SC.BGT = 100_8	1 → User tried to do a .GET
	SC.BERR = 16_8	.SYSLD error number.
	SC.BAB = 1_8	1 → Job abort.
153	SC.VTR	VT ON display file restart address or zero if display file not set up.
154	SC.PRC	Default file protection code.
155	SC.TRN	Reserved.
156	SC.TLM	Two's complement of time limit in seconds or zero if no time limit.
157	SC.SDV	Handler index of system device for Linking Loader.

Table 6-1 (cont)

System Communication (.SCOM) Table

Location	Mnemonic	Description
160	SC.TMT	Two's complement of number of clock ticks until .TIMER interrupt.
161	SC.TMA	.TIMER interrupt routine entry point address.
162	SC.BTA	Address of mass storage busy table.
163	SC.BTL	Number of words per mass storage busy table entry.
164		Reserved
165	SC.CQF	.GET/.PUT flag register (for communication with QFILE):
	SC.QFLG = 400000_8	1 → Call QFILE 0 → Return from .PUT
	SC.QNF = 2000_8	1 → Skip file transfer operation.
	SC.QNRM = 1000_8	1 → Exit to non-resident monitor when QFILE completes.
	SC.QPUT = 400_8	1 → Transfer ↑Q area to file 0 → Transfer file to ↑Q area
	SC.QRTN = 7_8	.PUT/.GET return code.
166	SC.CQB	.MTRAN parameter block for ↑Q area contains:
167	SC.CQB+1	First block of ↑Q area First address -1 in core Two's complement transfer length
170	SC.CQB+2	The transfer length is the minimum of the ↑Q area size and the current system memory size.
171	SC.TDT	Tomorrow's date, formatted MMDDYY.
172		Reserved.
173	SC.TMR	Two's complement of the number of ticks left in this second.
174	SC.LFR	Two's complement of the number of ticks per second.
175	SC.RTF	Indicates the current position within the batch stream (batch mode) or the run time file (BOSS mode).

Table 6-1 (cont)
System Communication (.SCOM) Table

Location	Mnemonic	Description
176 177	SC.FRH SC.FRH+1	Free memory in high core (above the bootstrap). SC.FRH contains the address of the first free location. SC.FRH+1 contains the address of the last free location.
200	SC.TCB	Address of Task Control Block transfer vector table.
201	SC.U01	Reserved for customer use.
202	SC.U02	Reserved for customer use.
203	SC.U03	Reserved for customer use.
204	SC.U04	Reserved for customer use.
205	SC.U05	Reserved for customer use.
206 207	SC.BFNM SC.BFNM+1	.SIXBT batch stream filename or zero if batch device is non-file oriented.
210	SC.BFXT	.SIXBT batch stream file extension.
211	SC.BUIC	.SIXBT batch stream file UIC code.
212	SC.BDEV	.SIXBT batch stream device mnemonic.
213	SC.BUNT	Batch stream device unit number (high 3 bits).

	Word #		Value	Description
	Ø		ØØØnnn	Pointer to first free word after SYSBLK (There is one set of seven words/core image program.)
	.			
	.			
	7N+1		.SIXBT	Name of System Program or overlay
	7N+2		.SIXBT	
S	7N+3		nnnnnn	Number of first block on system device occupied by this program or overlay.
Y	7N+4		ØØØØnn	Number of blocks occupied by this program or overlay
S	7N+5		addres	Thirteen-bit first address for this program or overlay
B				
L	7N+6		Ønnnnn	Program size
↓ K	7N+7		addres	Thirteen-bit starting address for this program or overlay
	.		.	.
	.		.	.
	.		.	.
	nnn			
(free area)				
	mmm		ØØØØ1Ø	Number of words in this entry (in this case, 1Ø)
	mmm+1		.SIXBT	Name of this system program (left-justified and zero-filled)
	mmm+2	p	.SIXBT	
	mmm+3	r	.SIXBT	Name of an overlay (left-justified and zero-filled) -- overlays are optional
	mmm+4		.SIXBT	
	mmm+5	o	ØØØØØ2	Number of buffers required by this system program (Bits Ø-6 = Ø means the end of any overlay names. This is why program and overlay names must be left-justified.)
↑ C		g		
O		1		
M	mmm+6		.DAT&777	Active .DAT slot
B	mmm+7		.DAT&777	Active .DAT slot (Note: 777777 for a .DAT slot means all positive .DAT slots.)
L	mmm+10		ØØØØØ5	Number of words for this entry (in this case, 5)
K	mmm+11	p	.SIXBT	Name of this system program
	mmm+12	r	.SIXBT	
	mmm+13	0	ØØØØØ1	Number of buffers required by this program (Note that this program has no overlays.)
		g		
	mmm+14	2	.DAT&777	.DAT slot for this program
	.		.	.
	.		.	.
	.		.	.
	777		ØØØ5ØØ	Pointer to first word in COMBLK. The two contiguous blocks on the system device that hold SYSBLK and COMBLK are treated by the system as one large block.

Figure 6-1

SYSBLK and COMBLK

6.2.2 COMBLK

COMBLK contains information the System Loader and the Nonresident Monitor need to remember about the current core-image system programs. The last location in COMBLK (that is, location 377 of block 35) contains the block address of the first entry in COMBLK. The remainder of COMBLK consists of variable-length entries associated with the system programs. The Nonresident Monitor searches COMBLK when it finds no match for a typed command in its own Command Table. Figure 6-1 illustrates the organization of COMBLK. SGEN XVM adds names of core-image system programs by making them the new first entry. In this way, SYSBLK and COMBLK build toward the center.

6.2.3 SGNBLK

SGNBLK (block 36 on the system device) contains all the system parameters not directly associated with core-image system programs. The bulk of SGNBLK is concerned with I/O (.DAT slots, .UFDT slots, Skip Chain Order, Handlers, and skip IOT codes and mnemonics). The first few registers hold such important system information as the system device, SC.MOD contents, and so on. The very first word in SGNBLK points to the block address of the first free word after SGNBLK. The next entry is an offset word indicating the total length (including itself) of the miscellaneous system parameter table to follow. This table includes the size of the .DAT and the size of the skip chain. The end of the handler and skip IOT table is the first free entry of the block.

The .DAT slot table corresponds to the legal range of .DAT slots, with the maximum negative set to 15_8 and the maximum positive set to a number not to exceed 77_8. The .DAT slots are in the form in which they appear when the Nonresident Monitor is in core. That is, the unit number is in bits Ø-2, and the number of the handler right-justified in bits 3-17. The handler number for the first handler in the Device Handler-Skip IOT Table is zero, for the pseudo-handler NON, TTA. is one, and so on. The constant 1ØØØØØ indicates a fixed or illegal .DAT slot (such as -2, -3, and Ø). DOSGEN will not modify such slots.

The .UFD Table is in one-to-one correspondence with the .DAT slot Table. An entry of .SIXBT 'UIC' indicates that the logged in UIC is to be substituted for the name UIC in the table. An entry of .SIXBT 'SYS' indicates BNK or PAG is to be substituted, in accordance with the current

addressing mode. Otherwise, the contents of each location will be the .SIXBT representation of the corresponding .UFD slot.

The Skip Chain Table lists the system skip IOT's in order. A negative skip (one that skips on "off", not "on") is represented in one's complement. Not all skips in the handler Skip IOT Table (described below) need to be included in the Skip Chain Table.

The Device Handler/Skip IOT Table contains all the handler names and skip IOT numbers and mnemonics for each I/O device identified to the system. Every such device has an entry in the table. A handler name must be exactly three characters in length, with the last character not an octal digit. The device code for a device is exactly two characters. The first two characters of each handler name for a device must be the device code. This fact is essential for understanding the format of a device entry, since the device code is never stored as such in an entry, but is inferred from the device handler name. The typical entry for a device is the following:

1. The first words of an entry contain the handler names for a device in .SIXBT. Each handler name is different, and the end of the list of handlers is determined by a word with zeros in bits Ø-5 (the first character position).
2. The word that terminated the list of handler names contains the number of skip IOT's for the device. For each skip IOT, there are three words in the table: two for the skip mnemonic and one for the actual code.

The next device entry follows the last skip for the previous device. Handlers may be entered without any skips, and skips may be included for handlerless devices. Figure 6-2 illustrates the organization of SGNBLK.

6.3 DISK-RESIDENT CHANGING BLOCKS

The System Loader uses block 37 of the system device to store an image of .DAT and .UFDT. Other disk-resident changing blocks are the storage Allocation Table and the Bad Allocation Table. These tables are described in Chapter 7.

Location	Value	Description
Ø	ØØØnnn	Pointer to first free entry in SGNBLK
1	ØØØØ15	Number of miscellaneous parameters
2	ØØØnnn	Size of .DAT plus size of .UFD = (number of positive .DAT slots + 16_8)*2. (Initial value is $2Ø_8$ positive .DAT slots.)
3	ØØØnnn	Number of skips in Skip Chain
4	nnnnnn	System device code in .SIXBT
5	nnnnnn	Original contents of SC.MOD
6	nnnnnn	Original contents of SC.MSZ
7	nnnnnn	Number of words per buffer (SC.BLN)
1Ø	nnnnnn	Default number of buffers (SC.BNM)
11	.SIXBT	Monitor Identification Code
12	nnnnnn	Information on VT and CTRL X (SC.VTF)
13	ØØØØØn	Default files protection code (SC.PRC)
14	ØØnnnn	Size of the Resident Monitor Patch Area
15	7777nn	Minus the number of clock ticks in a second (-74 for 60 hz, -62 for 5Ø hz)
.	ØØØnnn	Device assignments for the .DAT (made by handler numbers).
.	.	
.	.	
.	.	
.	ØØØnnn	
.	.SIXBT	UIC assignments for the UFD.
.	.	
.	.	
.	.	
.	.SIXBT	
.	nnnnnn	Skip Chain Table (Negative skips in one's complement).
.	.	
.	.	
.	.	
.	nnnnnn	
.	.SIXBT	The last part of the SGNBLK is the Device Handler-Skip IOT Table. Each entry starts with the .SIXBT representations of all handlers for a particular device. (First two characters equal device code, for all handlers.) Zeroes in the first six bits of a word indicates the end of the handler names, and says that the rest of the word contains the number of skips for this entry's device. The skip IOT's follow immediately. As above, one's complement skips indicate negative skips. Note, however, the confusing fact that a one's complement of a skip IOT is a positive number. Thus, 7Ønnnn complemented is Ø7nnnn.
.	.	
.	.	
.	.	
.	.	
.	.	
.	.	
.	.SIXBT	
.	ØØØØØ3	
.	nnnnnn	
.	nnnnnn	
.	nnnnnn	
.	.SIXBT	
.	ØØØØØ1	
.	nnnnnn	
.	.	
.	.	
nnn	.	

Figure 6-2
SGNBLK

6.4 TEMPORARY TABLES BUILT FROM DISK-RESIDENT TABLES

6.4.1 The Overlay Table

The System Loader builds the Overlay Table from the entries in SYSBLK referenced by a core-image system program entry in COMBLK. That is, the Overlay Table contains an entry for the system program itself, and one for each of its overlays. Table 6-2 illustrates the format of an entry in the Overlay Table. The first entry in the Overlay Table is pointed to by SC.OTB. SC.OTB will contain zero, if there are no entries in the Overlay Table. This will occur during Linking Loader or EXECUTE loads. The overlay table is terminated by a word containing zero.

.OVRLA is the only Monitor function that looks at the Overlay Table. If the .OVRLA processor finds a match to the .OVRLA argument in the Overlay Table, it uses the parameters listed in the table to bring it in via a Monitor TRAN. Note that this bypasses the System Loader, and does not change the handler load. Thus, the overlay must use only those .DAT slots required by the original program, the one listed in COMBLK.

If the .OVRLA processor does not find a match in the Overlay Table, it calls in the System Loader, which searches COMBLK for the requested program. This type of overlay request does not require that .DAT slot assignments be the same. On the other hand, the System Loader refreshes all of core except .SCOM, etc. Thus, communication between overlays is more difficult. The resident patch area, however, can be used for this purpose.

6.4.2 The Device Table

The Device Table is built by the System Loader interface whenever PIP is being loaded, or when PIP is listed in COMBLK among the overlays for a program. It is located just above the register pointed to by SC.RMS, and has an entry for each positive .DAT slot plus an additional entry indicating the system device code. If a slot has an assigned device, the low-order twelve bits of the corresponding entry in the Device Table will contain the device's code, in .SIXBT. Bit 2 is set when the slot is busy. If no device is assigned to a slot, the corresponding entry in the Device Table will contain zero.

6.4.3 The Input/Output Communication (IOC) Table

The System Loader builds the IOC Table and locates it just below the first register of the System Loader. It contains an entry for each handler in the system, in the order that they appear in SGNBLK. The entries themselves contain the handler name in Radix 5Ø. The System Loader and the Linking Loader use the handler number supplied by the Nonresident Monitor to index down the IOC Table. They use the contents of the entry for a .SEEK to the IOS UIC.

6.4.4 The Device Assignment Table (.DAT)

The Device Assignment Table makes the association between logical and physical devices. The Monitor knows its location by the contents of SC.DAT, which points to entry zero in the Table. Specific slots are found by indexing on the contents of SC.DAT. The number of negative slots is fixed at 15_8. The number of positive slots is specified by SC.SLT, and may be any positive number less than $1ØØ_8$. It is specified with SGEN XVM.

The Nonresident Monitor places the handler number in the low order bits and the unit number in the high order bits. It derives the handler number from SGNBLK. As mentioned above, the System Loader and the Linking Loader subsequently use the IOC Table to determine the handler name. After either loader has loaded and relocated a handler, it places the handler's starting address in the .DAT slot that references that handler. The unit number remains in the high-order three bits. Slots with no handler (NON) contain zero. Active .DAT slots are designated by COMBLK, for core-image system programs, and by .IODEV pseudo-ops for the Linking Loader and EXECUTE.

6.4.5 The User File Directory Table (.UFD)

.UFD+Ø is offset from .DAT+Ø (pointed to by SC.DAT) by the sum of the positive and negative .DAT slots. Each .DAT slot has a corresponding .UFD slot. UIC's in the .UFD are packed in .SIXBT. The address of .UFD+Ø is stored in SC.UFD.

6.4.6 The Skip Chain

Register 1 of Bank Ø contains a jump to the beginning of the Skip Chain. The Skip Chain is defined with SGEN XVM, is located in SGNBLK, and is

rebuilt every time the System Loader is called in. The SGEN XVM Utility Manual describes considerations for constructing the Skip Chain.

6.5 TEMPORARY TABLES BUILT FROM SCRATCH

6.5.1 File Buffer Transfer Vector Table

The System Loader allocates space for the buffer pool, and creates the File Buffer Transfer Vector Table. SC.BTB points to the first entry in the table, and the number of entries is specified by SC.BNM. Each entry in the table contains the address of a buffer, or its one's complement. Negative address indicate a busy buffer. Since references to buffers must be indirect anyway, buffers are allocated without regard to bank boundaries.

6.5.2 The Mass Storage Busy Table

Entries in this table are allocated by the System Loader or the Linking Loader. The Mass Storage Busy Table is pointed to by SC.BTA. SC.BTL contains the number of words per entry in the table, and SC.ACT contains the current number of entries. Generally speaking, there are as many entries in the Busy Table as there are active .DAT slots, although the disk handlers are the only ones that currently refer to the Busy Table.

The .INIT command to a disk handler establishes a Busy Table entry. The .CLOSE command (or the Rewind .MTAPE command) deletes the corresponding entry. Table 6-3 illustrates a typical Busy Table Entry.

The first word of an active entry in the Busy Table contains the .DAT slot in bits 9-17. The disk handlers save information about the UFD current for this .DAT slot in the Mass Storage Busy Table. They save information about the file current to the .DAT slot (if any) in the buffer pointed to by word 1 of the Busy Table Entry. More information on the disk handlers and file structure is contained in Chapter 7.

6.6 RESERVED WORD LOCATIONS

Word locations Ø through 77, are dedicated systems locations. The contents of these locations are described in Table 6-4.

Table 6-2

Overlay Table

Word #	Contents
N,N+1	.SIXBT name of Overlay or zero if end of table
N+2	First block number
N+3	First address, minus 1
N+4	Size, in two's complement
N+5	Fifteen-bit starting address

Table 6-3

Mass Storage Busy Table Entry

Word #	Contents
N	Device Type$_{\emptyset-2}$, Unit Number$_{3-5}$, Write Check$_{6}$,.DAT$_{9-17}$
N+1	Buffer Address, or $\emptyset$, if none allocated
N+2	Three-character UIC
N+3	First UFD block for this UIC
N+4	UFD Entry size for files in this UFD

Table 6-4

Reserved Address Locations

ADDRESS	USE
$\emptyset$	Stores the contents of the PC, link, user/exec mode status, and bank/page mode status during a program interrupt
1	JMP to Skip Chain
2	Reserved
3	.MED, entry to Monitor Error Diagnostic routine
4	JMS to error handler
5	Reserved
6	Reserved
7	Stores real time clock count
10-17	Autoindex registers
20	Stores the contents of the PC, link, user/exec mode status, and bank/page mode status on a CAL instruction.
21	JMS to CAL handler
22-37	Seven pairs of word count-current address registers for use with 3-cycle I/O device data channels.
40-77	Store unique entry instructions for each of 32_{10} automatic priority interrupt channels.

CHAPTER 7

FILE STRUCTURES

7-1 DECTAPE FILE ORGANIZATION

DECtape can be treated either as a directoried or non-directoried device.

7.1.1 Non-Directoried DECtape

A DECtape is said to be non-directoried when it is treated as magnetic tape by issuing the .MTAPE commands: REWIND or BACKSPACE, followed by .READ or .WRITE. No directory of identifying information of any kind is recorded on the tape. A block of data (255_{10} word maximum), exactly as presented by the user program, is transferred into the handler buffer and recorded at each .WRITE command. A .CLOSE terminates recording with a software end-of-file record consisting of two words: 001005, 776773

Because braking on DECtape allows for tape roll, staggered recording of blocks is employed in XVM/DOS to avoid constant turnaround or time-consuming back and forth motion of physically sequential block recording. When recorded as a non-directoried DECtape, block Ø is the first block recorded in the forward direction. Thereafter, every fifth block is recorded until the end of the tape is reached, at which time recording, also staggered, begins in the reverse direction. Five passes over the tape are required to record all 1100_8 blocks.

7.1.2 Directoried DECtape

Just as a REWIND or BACKSPACE command declares a DECtape to be non-directoried, a .SEEK or .ENTER implies that a DECtape is to be considered directoried. A directory listing of any such DECtape is available via the (L)ist command in PIP. A fresh directory may be recorded via the N or S switch in PIP.

The directory of all DECtapes occupies all $4ØØ_8$ words of block $1ØØ_8$. It is divided into two sections: (1) a $4Ø_8$ word Directory Bit Map and (2) a $34Ø_8$ word Directory Entry Section.

The Directory Bit Map defines block availability. One bit is allocated for each DECtape block (1100_8 bits = 40_8 words). When set to 1, the bit indicates that the DECtape block is occupied and may not be used to record new information.

The Directory Entry Section provides for a maximum of 56_{10} files on a DECtape. Each file on the DECtape has a four-word entry. Each entry includes the three-word file name and extension, a pointer to the first DECtape block of the file, and a file active or present bit. Figure 7-1 illustrates the DECtape directory.

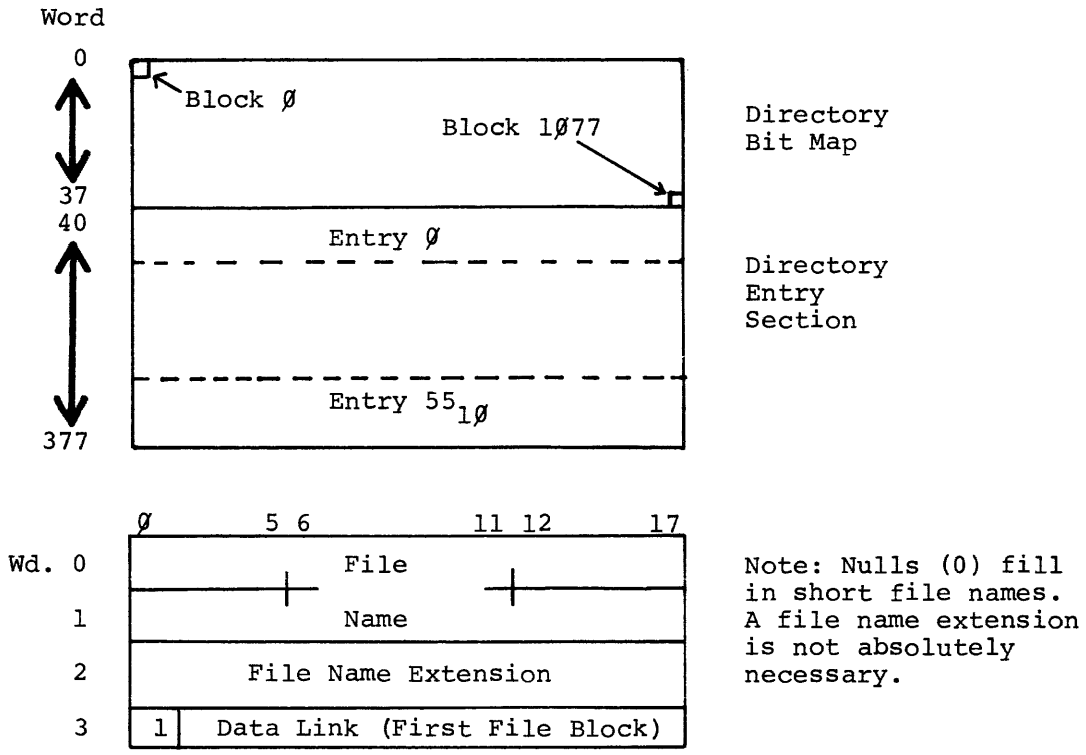


Figure 7-1

DECtape Directory

Additional file information is stored in blocks 71 through 77 of every directoried DECtape. These are the File Bit Map Blocks. For each file in the directory, a 40_8-word File Bit Map is reserved in block 71 through 77. The bit maps are contiguous, and the N^{th} file uses the

N^{th} bit map. Each block is divided into eight File Bit Map Blocks. A File Bit Map specifies the blocks occupied by that particular file and provides a rapid, convenient method to perform DECtape storage retrieval for deleted or replaced files. Note that a file is never deleted until the new one of the same name is completely recorded on the .CLOSE of the new file. When a fresh directory is written on DECtape, blocks 71 through 100 are always indicated in the Directory Bit Map as occupied. Figure 7-2 illustrates DECtape file bit maps.

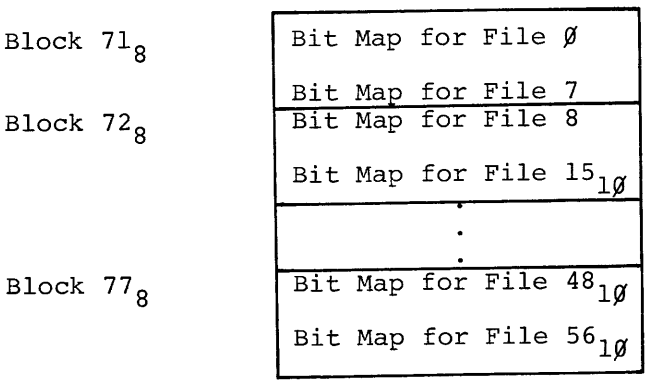


Figure 7-2

DECtape File Bit Map Blocks

Staggered recording (at least every fifth block) is used on directoried DECtapes, where the first block to be recorded is determined by examination of the Directory Bit Map for a free block. The first block is always recorded in the forward direction; thereafter, free blocks are chosen which are at least five beyond the last one recorded. The last word of each data block recorded contains a data link or pointer to the next block in the file. When turnaround is necessary, recording proceeds in the same manner in the opposite direction. When reading, turnaround is determined by examining the data link. If reading has been in the forward direction, and the data link is smaller than the last block read, turnaround is required. If reverse, a block number greater than the last block read implies turnaround.

A software end-of-file record (001005, 776773) terminates every file. The data link of the final block is 777777.

Data organization for each I/O medium is a function of the data modes. On directoried DECtape there are two forms in which data is recorded: (1) packed lines - IOPS ASCII, IOPS Binary, Image Alphanumeric, and Image Binary, and (2) dump mode data - Dump Mode.

In IOPS or Image Modes, each line (including header) is packed into the DECtape buffer. In IOPS Binary, a 2's complement checksum is computed and stored in the second word of the header. When a .WRITE which will exceed the remaining buffer capacity is encountered, the buffer is output, after which the new record is placed in the empty buffer. No record may exceed 254_{10} words, including header, because of the data link and even word requirement of the header word pair count. An end-of-file is recorded on a .CLOSE. It is packed in the same manner as any other line.

In Dump Mode, the word count is always taken from the I/O macro. If a word count is specified which is greater than 255_{10} (note that space for the data link must be allowed for again), the DECtape handler will transfer 255_{10} word increments into the DECtape buffer and from there to DECtape. If some number of words less than 255_{10} remain as the final element of the Dump Mode .WRITE, they will be stored in the DECtape buffer, which will then be filled on the next .WRITE, or with an EOF if the next command is .CLOSE. DECtape storage is thus optimized in Dump Mode since data is stored back-to-back. See Appendix A.

7.2 MAGNETIC TAPE

DOS provides for industry-compatible magnetic tape as either a directoried or non-directoried medium. The magnetic tape handlers communicate with a single TC-59D Tape Control Unit (TCU). Up to eight magnetic tape transports may be associated with one TCU; these may include any combination of transports TU-10A or B and TU-20A or B.

There are a number of major differences between magnetic tape and DECtape or Disk; these differences affect the operation of the device handlers. Magnetic tape is well suited for handling data records of variable length. Such records, however, must be treated in serial fashion. The physical position of any record may be defined only in relation to the preceding record. Three techniques available in I/O operations to block-addressable devices are not honored by the magnetic tape handlers:

a. The user cannot specify physical block numbers for transfer. In processing I/O requests that have block numbers in their argument lists (i.e., .TRAN) the handler ignores the block-number specification.

b. The only area open for output transfers in the directoried environment is that following the logical end of tape.

c. Only a single file may be open for transfers (either input or output) at any time on a single physical unit.

7.2.1 Non-directoried Data Recording (MTF)

MTF is intended to satisfy the requirements of the FORTRAN programmer while still providing the assembly language programmer maximum freedom on the design of his tape format. MTF writes out a record to the tape each time the main program issues a .WRITE. The length of the record is always two times the word pair count in the header word pair. FIOPS records are always as long as the buffer size returned on a .INIT (up to 256_{10} words). MTF returns a standard buffer size of 377_8, after a .INIT. The FORTRAN user may dynamically change this size, however, via the following instructions

Example:

```
(FORTRAN STATEMENTS)                    (MACRO STATEMENTS)
.                                       .TITLE SETMTB
.                                       .GLOBL .DA, MTBSIZ, SETMTB
.                          SETMTB       Ø
CALL SETMTB (IBFSIZ)                    JMS*    .DA
.                                       JMP     START
.                          BUFSIZ       Ø
                           START        LAC*    BUFSIZ    (any buffer size)
                                        DAC*    MTBSIZ
                                        JMP*    SETMTB
                                        .END
```

7.2.2 Directoried Data Recording (MTA., MTC.)

The programmer can make the fullest possible use of those features peculiar to magnetic tape by using MTF. On the other hand, MTF does not offer the powerful file-manipulation facilities available in the system. Directoried I/O allows device independence, and extensive use of the storage medium with a minimum of effort.

MTA. and MTC. do not support non-directoried data recording.

Every block recorded by MTA. (with the exception of end-of-file markers, which are hardware-recorded) includes a two-word Block Control Pair and not more than 255_{10} words of data. The data will contain the records from one or more .WRITE's.

The Block Control Pair serves three functions: it specifies the character of the block (label, data, etc.), provides a word count for the block, and gives an 18-bit block checksum. The Block Control Pair has the following format:

Word 1:

Bits 0 through 5: Block Identifier (BI). This 6-bit byte specifies the block type. Values of BI may range from 0 to 77_8. Current legal values of BI, for all user files, are as follows:

BI Value	Block Type Specified
00	User-File Header Label
10	User-File Trailer Label
20	User-File Data Block

Bits 6 through 17: Block Word Count (BWC). This 12-bit byte holds the 2's complement of the total number of words in the block (including the Block Control Pair). Legal values of BWC range from -3 to -401_8.

Word 2:

Bits 0 through 16: Block Checksum. The Block Checksum is the full-word, unsigned, 2's complement sum of all the data words in the block and word 1 of the Block Control Pair.

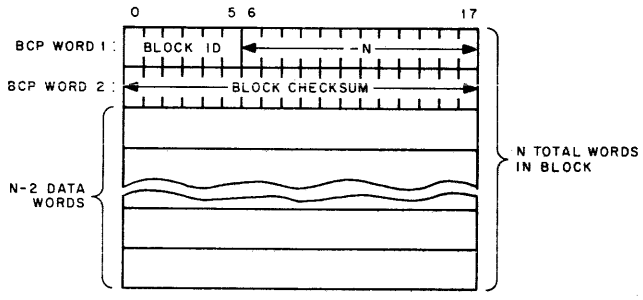


Figure 7-3

Block Format, File-Structured Mode

One of the main file functions of MTA. and MTC. is that of identifying and locating referenced files. This is carried out by two means: first, names of files recorded are stored in a file directory at the beginning of the tape; and second, file names are contained in the file's header and trailer labels.

7.2.2.1 Magnetic Tape File Directory

The directory, a single-block file (and the only unlabeled file on any file-structured tape), consists of the first recorded data block on the tape. It is a 257_{10} word block with the following characteristics:

a. Block Control Pair (words 1 and 2)

   Word 1

   Block Identifier = 74_8 = File Directory Data Block
   Block Word Count = $-401_8 = 7377_8$.

   Word 2:

   Block Checksum: As described above.

b. Active File Count (Word 3, Bits 9 through 17) 9-bit one's complement count of the active file names present in the File Name Entry Section (described below).

c. Total File Count (Word 3, Bits 0 through 8) 9-bit one's complement count of all files recorded on the tape, including both active and inactive files, but not the file directory block.

d. File Accessibility Map (Words 4 through 17): The File Accessibility Map is an array of 252_{10} contiguous bits beginning at bit 0 of word 4 and ending as bit 17 of word 17. Each of the bits in the Accessibility Map refers to a single file recorded on tape. The bits are assigned relative to the zeroth file recorded; that is, bit 0 of word 4 refers to the first file recorded; bit 1, word 4, to the second file recorded; bit 0, word 6, to the 37_{10}^{th} file recorded; and so on, for a possible total of 252_{10} files physically present.

   A file is only accessible for reading if its bit in the Accessibility Map is set to one. A file is made inaccessible for reading (corresponding bit = 0) by a .DLETE of the file, by a .CLOSE (output) of another file of the same name, or by a .CLEAR. A file is made accessible for reading (corresponding bit = 1) by a .CLOSE (output) of that file. Operations other than those specified above have no effect on the File Accessibility Map.

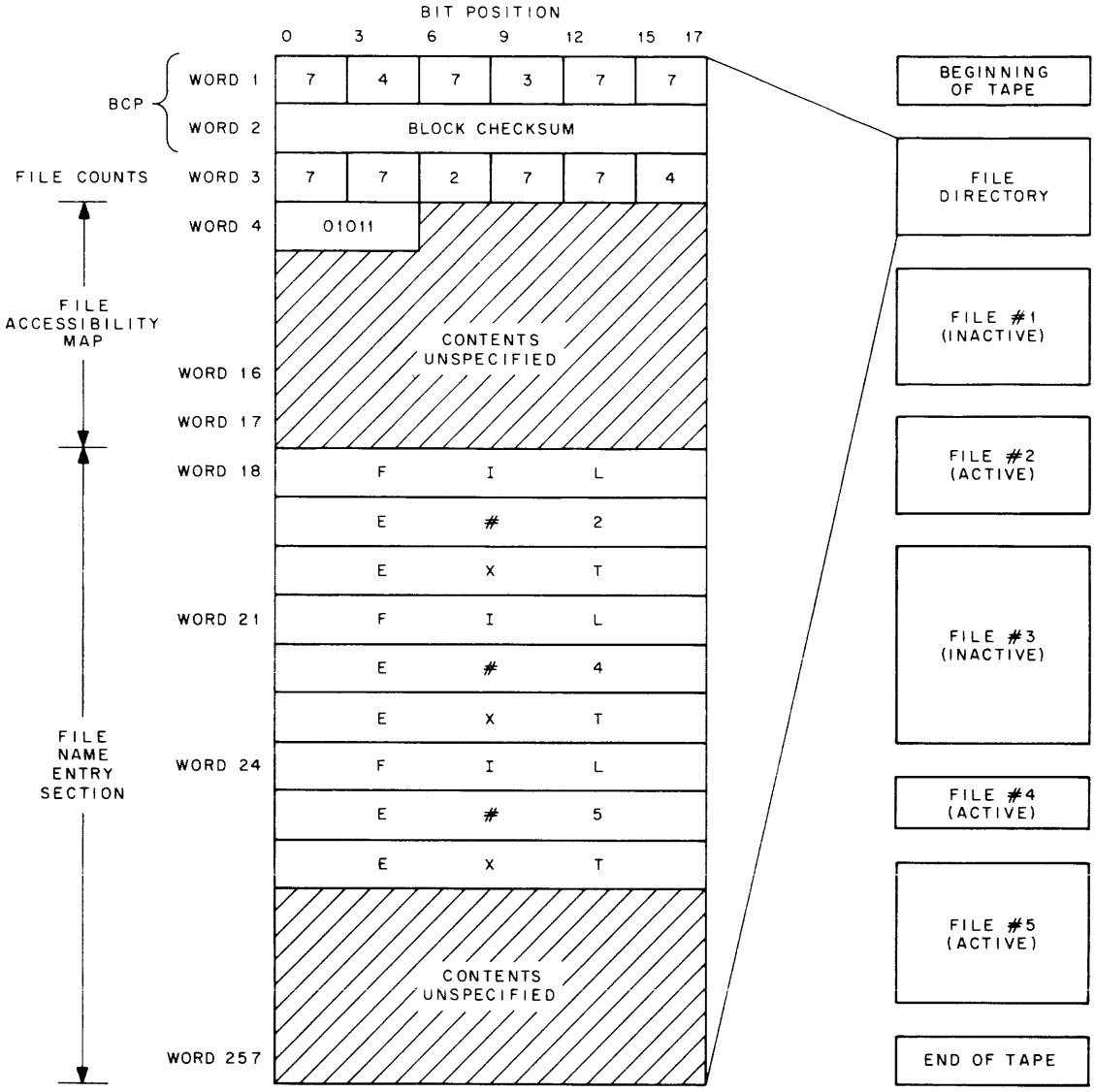


Figure 7-4a. Format of the File Directory Data Block, showing relationship of active and inactive files to file name entries and to Accessibility Map

Figure 7-4b. Format of file-structured tape, showing directory block and data files.

Figure 7-4

Magtape File Structure

e. File Name Entry Section (Words 18 through 257): The File Name Entry Section, beginning at word 18 of the directory block, includes successive 3-word file name entries for a possible maximum of 80 entries. Each accessible file on the tape has an entry in this section. Entries consist of the current name and extension of the referenced file in .SIXBT (left-adjusted and, if necessary, zero-filled).

The position of a file name entry relative to the beginning of the section reflects the position of its accessibility bit in the map. That bit, in turn, defines the position of the referenced file on tape with respect to other (active or inactive) files physically present. Only active file names appear in the entry section, and accessibility bits for all inactive files on the tape are always set to zero; accessibility bits for all active files are set to one.

To locate a file on the tape having a name that occupies the second entry group in the File Name Entry Section, the handler must (a) scan the Accessibility Map for the second appearance of a 1-bit, then (b) determine that bit's location relative to the start of the map. That location specifies the position of the referenced file relative to the beginning of the tape. The interaction of the File Name Entry Section and the Accessibility Map are shown in Figure 7-4.

7.2.2.2 User-File Labels

Associated with each file on tape is one label, the header label. It precedes the first data block of the file. Each label is 27_{10} words in length. Label format is shown in Figure 7-5.

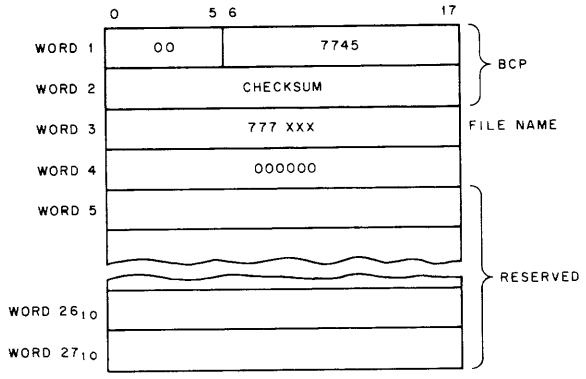


Figure 7-5

User File Header Label Format

7.2.2.3 File-Names in Labels

The handler will supply the contents of the file-name fields (Word 3) in labels. These are used only for control purposes during the execution of .SEEK's. The name consists simply of the two's complement of the position of the recorded file's bit in the Accessibility Map; the "name" of the first file on tape is 777777, that of the third file is 777775, and so on. A unique name is thus provided for each file physically present on the tape. Since there may be a maximum of 252_{10} files present, legal file-name values lie in the range 777777 to 777404.

7.2.3 Continuous Operation

Under certain circumstances, it is possible to perform successive I/O transfers without incurring the shut-down delay that normally takes place between blocks. The handler stacks transfer requests, and thus ensures continued tape motion, under the following conditions:

a. The I/O request must be received by the CAL handler before a previously-initiated I/O transfer has been completed.

b. The unit number must be identical to that of the previously initiated I/O transfer.

c. The I/O request must be one of those listed below to ensure successful completion. The handler in processing requests in continuous mode depends on receiving control at the CAL level in order to respond to I/O errors. The functions for which continuous operation is attempted include only the following:

1.	.MTAPE	3.	.WRITE
2.	.READ	4.	.TRAN

d. With MTA, more than one logical record may be in a physical block, so tape motion may stop if fewer successive .READ's or .WRITE's are issued than there are records in a block.

e. The previously-requested transfer must be completed without error. In general, successive error-free READ's (WRITE's) to the same transport will achieve non-stop operation. The following examples illustrate this principle.

Example 1: Successful Continued Operation

```
SLOT = 1
INPUT = 0
BLOKNO = 0
READ1          .TRAN SLOT, INPUT, BLOCKNO, BUFF1, 257
READ2          .TRAN SLOT, INPUT, BLOCKNO, BUFF2, 257
RETURN         JMP READ1
```

The program segment in Example 1 will most probably keep the referenced transport (.DAT slot 1) up to speed. The probability decreases as more time elapses between READ1 and READ2, and between READ2 and RETURN. Each .TRAN request causes an implicit .WAIT until its operation is completed.

Example 2: Unsuccessful Continued Operation

```
SLOT = 1
INPUT = 0
BLOKNO = 0
READ            .TRAN SLOT, INPUT, BLOKNO, BUFF, 257
STOP            .WAIT SLOT
RETURN          JMP READ
```

The program segment in Example 2 will not keep the tape moving because the .WAIT at location STOP prevents control from returning to location READ until the transfer first initiated at READ has been completed.

Example 3: Unsuccessful Continued Operation

```
SLOT1 = 1
SLOT2 = 2
INPUT = 0
BLOKNO = 0
READ1           .TRAN SLOT1, INPUT, BLOKNO, BUFF1, 257
READ2           .TRAN SLOT2, INPUT, BLOKNO, BUFF2, 257
RETURN          JMP READ1
```

This program segment will not provide non-stop operation because of the differing unit specification at READ1 and READ2.

7.2.4 Storage Retrieval on File-Structured Magnetic Tape

The use of a file accessibility map as well as block identifiers in Magtape file directories makes it almost impossible to retrieve the area of a deleted file from a magnetic tape. The execution of the deletion command (i.e., .DLETE) removes the name of the object file from the file directory, and clears the corresponding bit in the File Accessibility Map.

The only circumstance under which a file area may be easily retrieved is when the deleted file is also the last file physically on the tape. Under these conditions, the handler can retrieve the area occupied by the deleted file when the next .ENTER - .WRITE - .CLOSE sequence is executed. Users may also copy the active files to another device, renew the directory, and recopy the files.

7.3 DISK FILE STRUCTURE

7.3.1 Introduction

The XVM/DOS disk file structure is in some ways analogous to DECtape file structure. Ordinarily, each disk user has a directory which points to named files, just as each DECtape has a directory. The DECtape has only one directory, but the disk has as many directories as users have cared to establish. A single user's disk directory might correspond to a single DECtape directory. A single disk file's size is also limited only by the available space, as is true with DECtape. Although DECtape directories may reference a maximum of 56_{10} files, the number of files associated with any one directory on the disk is limited only by the available disk space.

The DECtape directory is in a known location -- at block 100. Since the disk may have a variable number of directories, the Monitor must know how to find each user's directory. It therefore maintains a Master File Directory (MFD) at a known location[1], and the Master File Directory points to each User File Directory (UFD). XVM/DOS allows only those users who know the Master Identification Code to have access to any protected UFD's within the MFD. Figure 7-6 illustrates the MFD. Appendix B is a flowchart of the Disk "A" Handlers.

7.3.2 User Identification Codes (UIC)

The Monitor finds User File Directories by seeking associated User Identification Codes (UIC's), which are all listed in the Master File Directory. The UIC is a three-character code that is necessary for all non-.TRAN I/O to the disk. .TRAN macros use no directory references. A programmer may operate under as many UIC's as he wishes, provided all are unique and none is reserved[2]. He may establish a new User File Directory by (1) logging in his new UIC to the Monitor via the LOGIN command, (2) calling PIP, and (3) issuing an N␣␣DK command. This establishes a new User File Directory, or refreshes (wipes clean) an old directory under that UIC. (.ENTER will also create a new MFD entry and/or a UFD, if none exists.) Figure 7-7, User File Directory, illustrates the organization of a UFD.

---

[1]On the RF and RK disk, the first block of the MFD is 1777 octal. On the RP disk, the first block of the MFD is 47Ø4Ø octal.
[2]The following are reserved UIC's: @@@, ???, PAG, BNK, SYS, IOS, CTP.

Word #	Contents	Description
Ø	777777	Dummy UIC used by system.
1	nnnnnn	Bad Allocation Table's first block number, or 777777, if there is none.
2	nnnnnn	SYSBLK's first block number, or -1, if there is none.
3	$4_{Ø-2}$+blknum	MFD entry size in bits Ø-2, plus the block number of the first submap
⋮	⋮	⋮
4N	.SIXBT	UIC for this UFD
4N+1	nnnnnn	Block number for the first block of this UFD or 777777, if no UFD exists (as after PIP's N␣DK))
4N+2	$P_{Ø}$+M	Protection code in bit Ø, plus the UFD entry size for each file
4N+3	spare	Unused at this writing
⋮	⋮	⋮
374^{1}	nnnnnn	Spooler disk area size
375^{1}	nnnnnn	Spooler disk area starting block
376	nnnnnn	Pointer to previous MFD block, or 777777 if none.
377	nnnnnn	Pointer to next MFD block, or 777777 if none.

Figure 7-6

Master File Directory

Word #	Contents	Description
⋮	⋮	⋮
8N	.SIXBT	Name of this file
8N+1	.SIXBT	and its
8N+2	.SIXBT	extension
8N+3	$T_{Ø}$+blknum	Truncation code in bit Ø, plus the number of the first block of the file
8N+4	nnnnnn	Number of blocks in this file
8N+5	ribptr	Pointer to the first block of the Retrieval Information Block
8N+6	$P_{Ø-1}$+ribwrd	Protection code in bits Ø-1, plus the first word in ribptr used by the RIB-- if the last block of the file has room for the RIB, the handlers will put it there, and load word 8N+6 accordingly.
8N+7	crdate	Date of file's creation -mmddyy (yy modulo 7Ø)
⋮	⋮	⋮
376	nnnnnn	Pointer to previous block, or 777777 if none
377	nnnnnn	Pointer to next UFD block, or 777777 if none

Figure 7-7

User File Directory

[1]Bits Ø-1 of word 374 are concatenated with bits Ø-1 of word 375 to produce a 4-bit checksum for the 16-bit fields of words 374,375. Checksum = word 374 (2-17) + word 375 (2-17) + 1 (modulo $16_{1Ø}$).

7.3.3 Organization of Specific Files on Disk

The Disk Handlers write out files in almost the same way that a DECtape handler does. Disk file blocks, however, have a forward and backward link. (Non-dump records are therefore limited to lengths of 254_{10} words.) Further, upon receipt of a .CLOSE I/O macro, the disk handlers fill out a Retrieval Information Block (RIB). The RIB performs the same functions as the file bitmap on DECtape, and also associates the logical sequence of blocks in the file with the physical locations of the blocks on the disk. The disk handler uses the RIB to implement .RTRAN commands and to delete files. Figure 7-8, The Retrieval Information Block, illustrates a RIB.

After a user has created a disk file he can access logical records sequentially via .READ commands, just as with DECtape files. He can also access physical blocks of that file by referencing relative block numbers in the .RTRAN command. (The .RTRAN commands require the file be opened with the .RAND command.)

7.3.4 Buffers

The handlers break buffers from the pool into three parts: (1) File Information (about 40_8 words)*, (2) the Block List -- addresses of pre-allocated blocks (between 4 and 253_{10} addresses, inclusive), and (3) data buffer (256_{10} words). Figure 7-9, Disk Buffer, illustrates the breakdown of disk buffers.

7.3.4.1 Commands That Obtain And/or Return Buffers

The following commands obtain buffers from the pool, and return them immediately after execution:

.DLETE
.RENAM
.CLEAR

The following commands obtain a buffer from the pool and do not return it until a subsequent .CLOSE is performed:

.FSTAT
.ENTER
.SEEK
.RAND

---

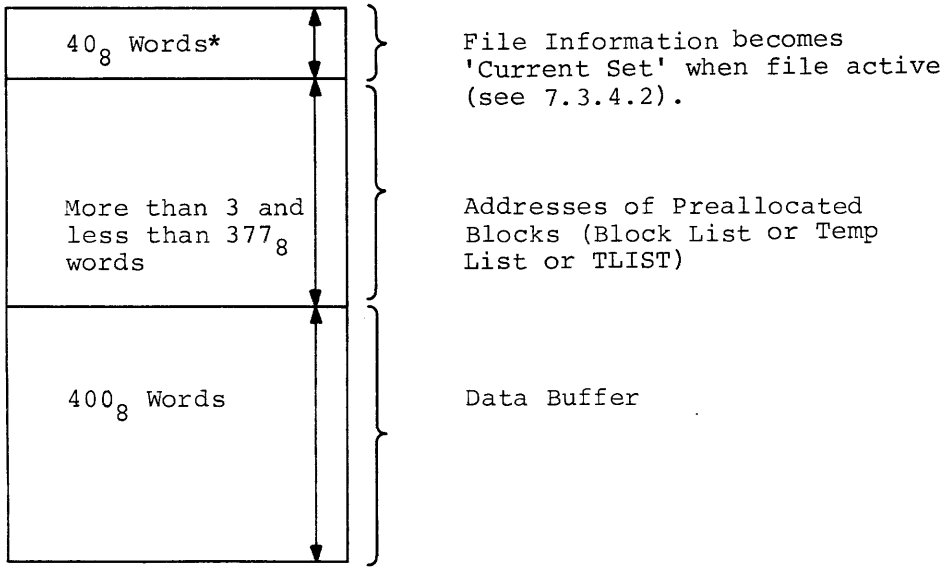
*This number is determined by assembly parameters.

Word #	Contents	Description
Ø*	nnnnnn	Total number of blocks described by this physical block.
1	nnnnnn	First data block pointer.
2	nnnnnn	Second data block pointer.
3	nnnnnn	Third data block pointer.
.	.	.
.	.	.
.	.	.
376	nnnnnn	Pointer to previous RIB block or -1 if no previous RIB block.
377	nnnnnn	Pointer to next RIB block or -1 if no next RIB block.

* Zeroth word of the RIB may not be zeroth word of physical block. This occurs whenever the entire RIB will fit in the last data block of the file.

Figure 7-8

Retrieval Information Block



*This is not a fixed number. It is different for RP, RK and RF.

Figure 7-9

Disk Buffer

The following commands return a buffer to the pool, if any was allocated.

.INIT
.CLOSE
.MTAPE (rewind)

7.3.4.2 The Current Set

The handlers retain information about the last file and .DAT slot processed in an internal storage area. This area is called the "Current Set", and is swapped back to the file's buffer whenever a command to a different file is used. Thus,

.WRITE to .DAT slot A
.WRITE to .DAT slot B

will swap the Current Set, but...

.WRITE to .DAT slot A
.TRAN to .DAT slot A
.WRITE to .DAT slot A

will not swap the Current Set.

7.3.5 Pre-allocation

The handlers pre-allocate blocks on the disk upon all .ENTER commands, and whenever sufficient .WRITE commands have been issued to use up the pre-allocated blocks. The number of pre-allocated blocks will be the minimum of the number of free blocks on the device and the number of address slots available in the Temp List (block list).

When the handlers pre-allocate blocks, they fill out the bit maps, and immediately fill out the RIB and write it out in one of the pre-allocated blocks.

Upon a .CLOSE command, the handlers give back unused blocks, and re-write the RIB.

The number of blocks in the Block List depends on the size of the buffer, which is determined at system generation by setting the buffer size. The larger the Block List, the faster will be output. Smaller Block Lists may give more efficient allocation of core and disk space. Smaller buffers save core. Further, the number of pre-allocated blocks may affect concurrently opened files on a disk that is tight for space. Thus, if the Block List is sixty entries long, and there are forty blocks left on the disk, a .ENTER to .DAT slot will pre-allocate all forty, leaving none for any subsequent .ENTER's to different .DAT slots.

IOPS 70 will occur when there are less than four free blocks on the disk when a handler tries to pre-allocate blocks.

7.3.6 Storage Allocation Tables (SAT's)[1]

The disk handlers use a Storage Allocation Table, in order to distinguish between allocated and free blocks. If more than one physical block is required, the individual blocks are called Submaps.

Unlike DECtape, the Storage Allocation Table is never held in core. When the handlers wish to preallocate some blocks, they read in the required Submap, and write out the updated one.

Storage Allocation blocks use the following format:

WORD Ø	Total blocks on the disk
WORD 1	Number of blocks described by this Submap
WORD 2	Number of blocks occupied in this Submap
WORD 3	First word of the bit map (eighteen blocks per word)
.	
.	
.	
WORD 376	Pointer to previous Submap (or 777777)
WORD 377	Pointer to next Submap (or 777777)

The bit maps refer to blocks in numerical order. Thus, bit Ø of word three of a Submap will refer to block N, bit 1 will refer to block N+1, and so on. The block is free if the corresponding bit equals Ø. Starting and ending block numbers for all Submaps are retained in the handlers. Bit Ø of word three in the first submap, refers to block zero.

---

[1]The first SAT block is located at 1776_8 for the RF and RK system and 764_8 for the RP system.

7.3.7 Bad Allocation Tables (BAT's)

Occasionally, a particular block on the disk will not record data correctly. In such instances, the handlers should be prevented from using the bad blocks. Accordingly, PIP maintains a Bad Allocation Table. Whenever a user updates that table, PIP will set the appropriate bit in the Storage Allocation Table. The block is thus made unavailable. Refer to the PIP XVM Utility Manual for more information.

CHAPTER 8
WRITING NEW I/O DEVICE HANDLERS

This chapter contains information essential for writing new I/O device handlers to work in DOS.

8.1 I/O DEVICE HANDLERS, AN INTRODUCTION

All communications between user programs and I/O device handlers are made via CAL instructions followed by an argument list. The CAL Handler in the Monitor performs preliminary setups, checks the CAL calling sequence, and transfers control via a JMP* instruction to the entry point of the device handler. When the control transfer occurs (see Figures 8-1 and 8-2), the AC contains the address of the CAL in bits 3 through 17 and bits 0, 1, and 2 indicate the status of the Link, Bank/Page mode, and Memory Protect, respectively, at the time of the CAL. Note that the contents of the AC at the time of the CAL is not preserved when control is returned to the user.

On machines that have API, the execution of a CAL instruction automatically raises the priority to the highest software level (level 4). Control passes to the handler while it is still at level 4. Device handlers must remain at level 4 until control is returned to the user.

A debreak and restore (DBR) instruction should be executed just prior to the JMP* which returns control to the user, allowing debreak from level 4 and restoring the conditions of the Link, Bank/Page mode, and Memory Protect. Any IOT's issued at the CAL level (level 4 if API present, mainstream if no API) should be executed immediately before the

```
DBR
JMP*
```

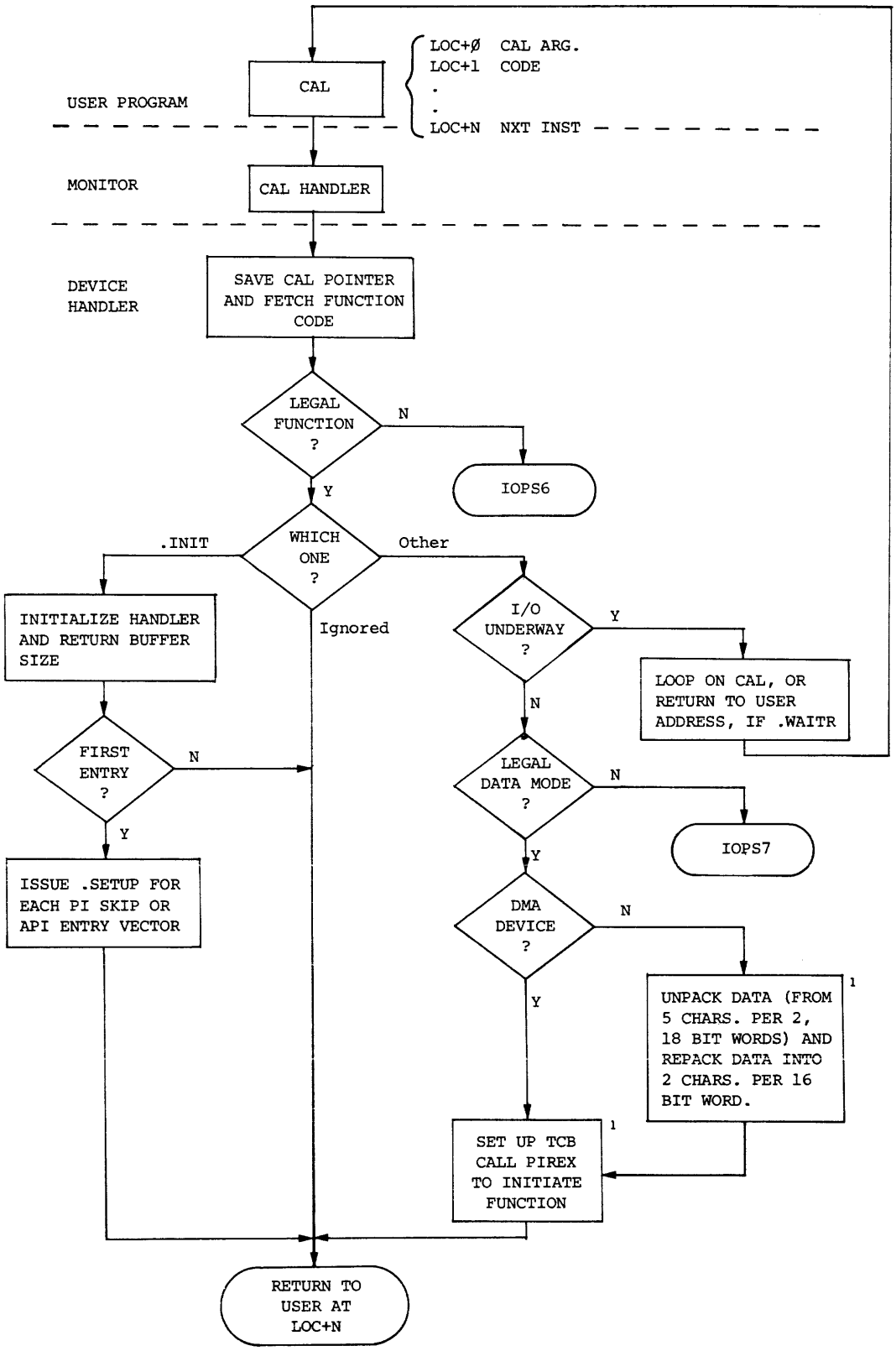


Figure 8-1

CAL Entry to Device Handler

[1]For non-unibus devices both these branches would be replaced by a single initiate function routine.

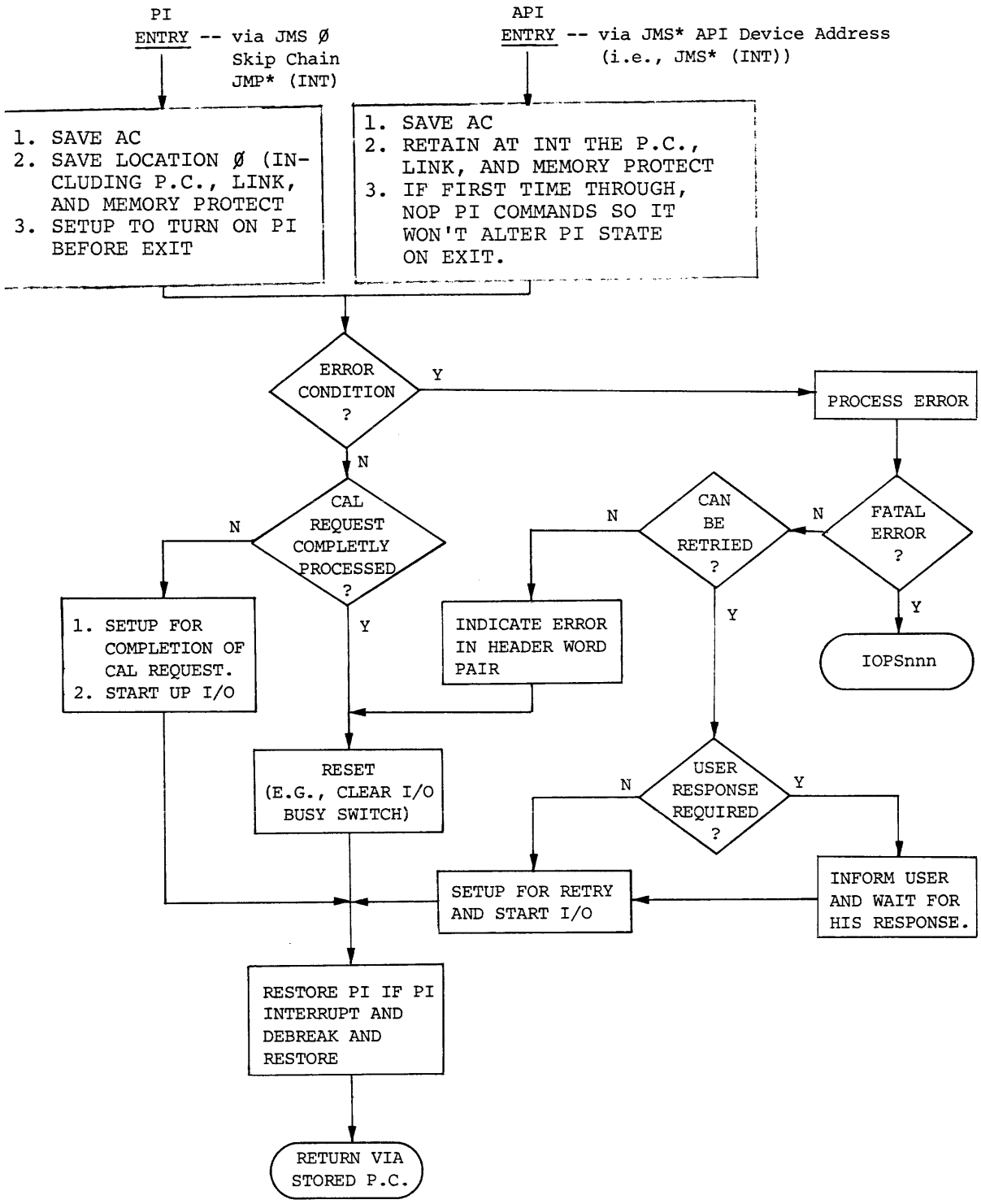


Figure 8-2

PI and API Entries to Device Handlers

exit sequence in order to ensure that the exit takes place before the interrupt from the issued IOT occurs.

CAL's may be executed within an I/O handler, but not while processing an interrupt.

8.1.1 Setting Up the Skip Chain and API (Hardware) Channel Registers

When the Monitor is loaded, the Program Interrupt (PI) Skip Chain and the Automatic Priority Interrupt (API) channels are set up to handle the TTY keyboard and printer and clock interrupts only. The Skip Chain contains the other skip IOT instructions, but indirect jumps to an error routine result if a skip occurs.

All unused API channels, memory protect, memory parity, and powerfail, also transfer control to the error address.

When a device handler is called for the first time in a core load, it must call a Monitor routine (.SETUP) to set up its skip(s) in the Skip Chain, or its API channel, prior to performing any I/O functions.

The calling sequence is as follows:

```
     CAL N           /N = API channel register 40 through 77.
                     /0 if device not connected to API.
     16              /.SETUP function code.
     SKP IOT         /Skip IOT for this device.
     DEVINT          /Address of interrupt handler.
       (normal return)
```

8.1.2 Handling the Interrupt

DEVINT exists in the device handler in the following format to allow for either API or PI interrupts. The following is for UNIBUS devices only:

```
ONLY1   LAC     (NOP        /LEAVE PI ALONE.  WHEN API IS RUNNING
        DAC     DEVION      /THESE REGISTERS
        DAC     DEVIOF      /ARE AVAILABLE
        DAC     IGNRPI      /THIS IS ONCE ONLY CODE
        JMP     COMMON
DEVPIC  DAC     DEVAC       /SAVE AC
        LAC*    (Ø
        DAC     DEVOUT      /SAVE PC, LINK, ADDRESSING MODE AND
                            /MEMORY PROTECT
        JMP     COMMON
DEVINT  JMP     DEVPIC      /PI ENTRY
        DAC     DEVAC       /API ENTRY; SAVE AC
        LAC     DEVINT
        DAC     DEVOUT      /SAVE PC, LINK, ADDRESSING MODE AND
IGNRPI  JMP     ONLY1       /API IS OPERATING, SO LEAVE PI ALONE.
                            /PI INTERRUPTS ARE NOT POSSIBLE, BE-
                            /CAUSE .SETUP EFFECTIVELY NOP'S PI
                            /SKIPS.
COMMON  CAPI-               /CLEAR API LEVEL '-' DONE FLAG.
DEVION  ION                 /PI ALLOWS INTERRUPTS; API DOES A NOP.
        .
        .
        .
DEVIOF  IOF                 /API DOES NOP; PI TURNS IO OFF TO ENSURE
                            /NON-REENTRANCE AFTER ISSUING IOT'S.
        LAC     (TCB        /GET ADDRESS  OF TCB IN AC
        SIOA                /PREVIOUS TCB ACCEPTED?
        JMP     .-1         /NO
        LIOR                /YES.  LOAD REGISTER IN INTERRUPT LINK
        .                   /THIS CAUSES A BR7 (HIGHER LEVEL)
/DISMISS ROUTINE            /INTERRUPT ON THE PDP-11.
        .
        .
        LAC     DEVAC       /RESTORE AC.
DVSWCH  ION                 /ION OR NOP.
        DBR                 /DEBREAK AND RESTORE CONDITIONS
        JMP*    DEVOUT      /OF LINK, ADDRESSING MODE AND MEMORY
                            /PROTECT.
```

If the Index, Autoincrement, or EAE registers are used by the I/O device handler, it is necessary to save and restore them.

The following is for non-UNIBUS devices:

```
ONLY1   LAC     (NOP        /LEAVE PI ALONE. WHEN API IS RUNNING
        DAC     DEVION      /THESE REGISTERS
        DAC     DEVIOF      /ARE AVAILABLE
        DAC     IGNRPI      /THIS IS ONCE ONLY CODE
        JMP     COMMON
DEVPIC  DAC     DEVAC       /SAVE AC
        LAC*    (Ø
        DAC     DEVOUT      /SAVE PC, LINK, ADDRESSING MODE AND
                            /MEMORY PROTECT
        JMP     COMMON
DEVINT  JMP     DEVPIC      /PI ENTRY
        DAC     DEVAC       /API ENTRY; SAVE AC
        LAC     DEVINT
        DAC     DEVOUT      /SAVE PC, LINK, ADDRESSING MODE AND
IGNRPI  JMP     ONLY1       /API IS OPERATING, SO LEAVE PI ALONE.
                            /PI INTERRUPTS ARE NOT POSSIBLE, BE-
                            /CAUSE .SETUP EFFECTIVELY NOP'S PI
                            /SKIPS.
COMMON  DEVCF               /CLEAR DEVICE DONE FLAG.
DEVION  ION                 /PI ALLOWS INTERRUPTS; API DOES A NOP.
        .
        .
        .
DEVIOF  IOF                 /API DOES NOP; PI TURNS IO OFF TO ENSURE
                            /NON-REENTRANCE AFTER ISSUING IOT'S.
        DEVIOT
        .
        .
        .
/DISMISS ROUTINE
        .
        .
        .
        LAC     DEVAC       /RESTORE AC.
DVSWCH  ION                 /ION OR NOP.
        DBR                 /DEBREAK AND RESTORE CONDITIONS
        JMP*    DEVOUT      /OF LINK, ADDRESSING MODE AND MEMORY
                            /PROTECT.
```

If the Index, Autoincrement, or EAE registers are used by the I/O device handler, it is necessary to save and restore them.

.SETUP must contain information for both API and PI. The XVM/DOS Monitor will set up the skip chain and/or API channels appropriately. The SGEN XVM Utility Manual gives the method for incorporating new handlers and associated Skip Chain entries into the Monitor.

8.2 WRITING SPECIAL I/O DEVICE HANDLERS

This chapter contains information prepared specifically to aid those users who plan to write their own special I/O device handlers for DOS.

DOS is designed to enable users to incorporate their own device handlers. Precautions should be taken when writing the handler however, to ensure compatibility with the Monitor.

Here is a summary of handler operation. The handler is entered via a JMP* from the Monitor as a result of a CAL instruction. The contents of the AC contain the address of the CAL in bits 3 through 17. Bit 0 contains the Link, bit 1 contains the Bank/Page Mode status, and bit 2 contains the Memory Protect status. The previous contents of the AC are lost.

In order to show the steps required in writing an I/O device handler, a complete handler (Example B) was developed with the aid of a skeleton handler (Example A). In addition, Appendices A and B are flowcharts of DTA and the A version of the disk handlers. Example A is referenced by part numbers to illustrate the development of Example B, a finished Analog-to-Digital Converter (ADC) I/O Handler. The ADC handler shown in Example B was written for the Type AFØ1B Analog to Digital Converter. This handler is used to read data from the ADC and store it in the user's I/O buffer.

The reader, while looking at the skeleton of a specialized handler as shown in Example A, should make the following decisions about his own handler. (The decisions made in this case are in reference to developing the ADC handler):

a. Services that are required of the handler (flags, receiving or sending of data, etc.) - By looking at the ADC IOT's shown in the Reference Manual, it can be seen that there are three IOT instructions to be implemented. These instructions are: Skip if Converter Flag Set, Select and Convert, and Read Converter Buffer.

The only service the ADC handler performs is that of receiving data and storing it in user specified areas. This handler will have a standard 256-word buffer.

b. Data Modes used (for example, IOPS ASCII, etc.) - Since there is only one format of input from the Type AF01B ADC, mode specification is unnecessary in Example C.

c. Which I/O macros are needed for the handler's specific use; that is, .INIT, .CLOSE, .READ, etc. For an ADC, the user would be concerned with four of the macros.

(1) .INIT would be used to set up the associated API channel register or the interrupt skip IOT sequence in the Program Interrupt Skip Chain. This is done by a CAL (N) as shown in Part III of Example A, where (N) is the channel address.

(2) .READ is used to transfer data from the ADC. When the .READ macro is issued, the ADC handler will initiate reading of the specified number of data words and then return control to the user. The analog input data received is in its raw form. It is up to the programmer to convert the data to a usable format.

(3) .WAIT detects the availability of the user's buffer area and ensures that the I/O transfer is completed. It would be used to ensure a complete transfer before processing the requested data.

(4) .WAITR detects the availability of the user's buffer area as in (3) above. If the buffer is not available, control is returned to a user specified address, which allows other processing to continue.

d. Implementation of the API or PIC interrupt service routine - Example A shows an API or PIC interrupt service routine that handles interrupts, processes the data and initiates new data requests to fully satisfy the .READ macro request. Note that the routines in Example A will operate with or without API. Example B uses the routines exactly as they are shown in Example A.

During the actual writing of Example B, consideration was given to the implementation of the I/O macros in the new handler in one of the following ways:

(1) Execute the function in a manner appropriate to the given device as discussed in(c). .INIT, .READ, .WAIT, and .WAITR were implemented into the ADC handler (Example B) under the subroutine names ADINIT, ADREAD, ADWAIT (.WAIT and .WAITR).

Wait for completion of previous I/O. (Example B shows the setting of the ADUND switch in the ADREAD subroutine to indicate I/O underway.)

(2) Ignore the function if meaningless to the device. See Example B (.FSTAT results in JMP ADIGN2) in the dispatch table DSPCH. For ignored macros, the return address must be incremented in some cases, depending upon the number of arguments following the CAL.

(3) Issue an error message in the case where it is not possible to perform the I/O function - (An example would be trying to execute a .ENTER on the paper tape reader.) In Example B, the handler jumps to DVERR6 which returns to the Monitor with a standard error code in the AC.

e. Special considerations for UNIBUS device handlers
When new handlers are written for devices on the UNIBUS in a UC15 system (RK based or RF/RP based UC15 option) the following has to be considered.

Since communication between the device handler on the XVM and the driver task running under PIREX on the PDP-11 is through Task Control Blocks (TCB), space in the Common Memory (memory that can be addressed by the XVM and the PDP-11) must be provided. The system as supplied by DEC has space reserved in the Resident Monitor for 3 user defined devices/programs/tasks, (refer to Section 2.9 for more information). This TCB must be properly setup (refer to the XVM UNICHANNEL Software Manual for more information) before the handler calls PIREX to initiate the operation.

Driver tasks (TTT)[1] running under PIREX report errors by setting the appropriate code (XX) in the device error status table in PIREX (refer to XVM UNICHANNEL Software Manual, for more information). The XVM/DOS poller prints out this error message, which appears as follows:

```
IOPSUC  TTT  XX
```

Users have to decipher this message. An example of this is,

```
IOPSUC  LPU  4
```

which reports that the LP11/LS11 line printer is not ready. There is no error message type out from the handler. This method of error handling is incorporated to permit error report during operation of these devices/tasks etc., under PIREX when their corresponding handlers are not present in core on the XVM (e.g., during Spooling).

[1]Each task running under PIREX has a 3 character code assigned to it which is present in the PIREX error table at assembly time.

After the handler has been written and assembled, the Monitor must then be modified to recognize the new handler. This is accomplished by the use of SGEN XVM described in the SGEN XVM Utility Manual.

When the system generation is complete, PIP XVM (refer to PIP XVM Utility Manual) must be used to add the new handler to the IOS UFD. At this time, the user is ready to use his specialized device handler in the XVM/DOS system.

8.2.1 Discussion of Example A by Parts

Part 1	Stores CAL pointer and argument pointer, and picks up function code from argument string.
Part 2	By getting proper function code in Part 1 and adding a JMP DSPCH, the CAL function is dispatched to the proper routine.
Part 3	This is the .SETUP CAL used to set up the PI skip chain or the API channel register.
Part 4	Shows the API and PI handlers. It is suggested these be used as shown.
Part 5	This area reserved for processing interrupt and performing any additional I/O.
Part 6	Interrupt dismiss routine.
Part 7	Increments argument pointer in bypassing arguments of ignored macro CAL's.

8.2.2 Example A, Skeleton I/O Device Handler

```
/CAL ENTRY ROUTINE
        .GLOBL DEV,             /MUST BE OF FORM AAA,
,MED=3                          /,MED (MONITOR ERROR DIAGNOSTIC)
DEV,    DAC     DVCALP          /SAVE CAL POINTER
        DAC     DVARGP          /AND ARGUMENT POINTER
        ISZ     DVARGP          /POINTS TO FUNCTION CODE
        LAC*    DVARGP          /GET CODE
        AND     (77777          /REMOVE UNIT NO IF APPLICABLE
        ISZ     DVARGP          /POINTS TO CAL*2
        TAD     (JMP DSPCH
        DAC     DSPCH           /DISPATCH WITH
DSPCH   XX                      /MODIFIED JUMP
        JMP     DVINIT          /1 = ,INIT
        JMP     DVFSAT          /2 = ,FSTAT, ,DLETE, ,RENAM
        JMP     DVSEEK          /3 = ,SEEK
        JMP     DVENTR          /4 = ,ENTER
        JMP     DVCLER          /5 = ,CLEAR
        JMP     DVCLOS          /6 = ,CLOSE
        JMP     DVMTAP          /7 = ,MTAPE
        JMP     DVREAD          /10 = ,READ
        JMP     DVWRTE          /11 = ,WRITE
        JMP     DVWAIT          /12 = ,WAIT
        JMP     DVTRAN          /13 = ,TRAN

/ILLEGAL FUNCTIONS IN ABOVE TABLE CODED AS:
/       JMP     DVERR6

/FUNCTION CODE ERROR
DVERR6  LAW     6               /ERROR CODE 6
        JMP*    (,MED+1         /TO MONITOR

/DATA MODE ERROR
DVERR7  LAW     7               /ERROR CODE 7
        JMP*    (,MED+1         /TO MONITOR

/DEVICE NOT READY
DVERR4  LAC     (RETURN         /RETURN (ADDRESS IN HANDLER)
                                /TO RETURN TO WHEN NOT READY
                                /CONDITION HAS BEEN REMOVED

        DAC*    (,MED
        LAC     (4              /ERROR CODE 4
        JMP*    (,MED+1         /TO MONITOR

/I/O UNDERWAY LOOP
DVBUSY  DBR                     /BREAK FROM LEVEL 4
        JMP*    DVCALP          /LOOP ON CAL

/NORMAL RETURN FROM CAL
DVCK    DBR                     /BREAK FROM LEVLE 4
        JMP*    DVARGP          /RETURN AFTER CAL AND
                                /ARGUMENT STRING

/THE DVINIT ROUTINE MUST INCLUDE
/A ,SETUP CALLING SEQUENCE FOR
```

```
/EACH FLAG CONNECTED TO API
/AND/OR PI A(AT SGEN TIME),
/THE SETUP CALLING SEQUENCE IS:

DVINIT  CAL     N               /N = API CHANNEL REGISTER
                                /(40 -77), N = 0 IF NOT CONNECTED
                                /TO API

        16                      /IOPS FUNCTION CODE
        SKPIOT                  /SKIP IOT TO TEST THE FLAG
        DEVINT                  /ADDRESS OF INTERRUPT
                                /HANDLER (PI OR API)

/THIS SPACE MAY BE USED FOR I/O SUBROUTINES

/INTERRUPT HANDLER FOR API OR PI
ONLY1   LAC     (NOP
        DAC     DEVION
        DAC     DEVIOF
        DAC     DVSWCH
        DAC     IGNRPI
        JMP     COMMON
DEVPIC  DAC     DEVAC           /SAVE AC
        LAC*    (0              /SAVE:PC, LINK, BANK/PAGE MODE
        DAC     DEVOUT          /AND MEMORY PROTECT
        JMP     COMMON
DEVINT  JMP     DEVPIC          /PI ENTRY
        DAC     DEVAC           /API ENTRY: SAVE AC
        LAC     DEVINT          /SAVE: PC, LINK, BANK/PAGE MODE
        DAC     DEVOUT          /MEMORY PROTECT
IGNRPI  JMP     ONLY1           /LEAVE PI ALONE
COMMON  DEVCF                   /ENABLE PI OR NOP
DEVION  ION                     /ENABLE PI OR NOP

/THIS AI  THE AREA DEVOTED TO PROCESSING INTERRUPT AND
/PERFORMING ANY ADDITIONAL I/O DESIRED,

DEVIOF  IOF                     /DISABLE PI OR NOP
        DEVIOT                  /DIMISSAL BEFORE INTERRUPT
                                /FROM THIS IOT OCCURS

/INTERRUPT HANDLER DISMISS ROUTE
DVDISM  LAC     DEVAC           /RESTORE AC
DVSWCH  ION                     /ION OR NOP
        DBR                     /DEBREAK AND RESTORE
        JMP*    DEVOUT          /LINK, BANK/PAGE MODE, MEMORY
                                /PROTECT

/IF THE HANDLER USES THE AUTOINCREMENT , INDEX
/OR EAE REGISTERS, THEIR CONTENTS
/SHOULD BE SAVED AND RESTORED, FUNCTIONS
/POSSIBLY IGNORED SHOULD CONTAIN
/PROPER INDEXING TO BYPASS
/CAL ARGUMENT STRING
/
/CODE TO BYPASS IGNORED FUNCTIONS
/



DVIGN2  ISZ     DVARGP          /BYPASS FILE POINTER
        JMP     DVCK
```

8.2.3 Example B, Special Device Handler for AF01B A/D Converter

```
/ADC HANDLER
/
ADSF=701301       /SKIP IF CONVERSION FLAG IS SET
ADSC=701304       /SELECT AND CONVERT (ADC FLAG IS CLEARED
                  /AND A CONVERSION IS INITIALISED)
ADRB=701312       /READ CONVERTER BUFFER INTO AC AND CLEAR FLAG
/
          .GLOBL  ADC.
IDX=ISZ
.MED=3            /MED (MONITOR ERROR DIAGNOSTIC)
/
ADC.      DAC     ADCALP  /SAVE CAL POINTER
          DAC     ADARGP  /AND ARGUMENT POINTER
          IDX     ADARGP  /POINTS TO FUNCTION CODE
          LAC*    ADARGP  /GET CODE
          IDX     ADARGP  /POINTS TO CAL + 2
          TAD     (JMP DSPCH
          DAC     DSPCH   /DISPATCH WITH
DSPCH     XX                /MODIFIED JUMP
          JMP     ADINIT  /1=,INIT
          JMP     ADIGN2  /2=,FSTAT,,DLETE,,RENAM
          JMP     ADIGN2  /3=,SEEK
          JMP     ADERR6  /4=,ENTER
          JMP     ADERR6  /5=,CLEAR
          JMP     ADIGN1  /6=,CLOSE
          JMP     ADIGN1  /7=,MTAPE
          JMP     ADREAD  /10=,READ
          JMP     ADERR6  /11=,WRITE
          JMP     ADWAIT  /12=,WAIT
          JMP     ADERR6  /13=,TRAN
/
/ILLEGAL FUNCTIONS IN ABOVE TABLE CODED AS
/         JMP     ADERR6
          .EJECT
```

```
/
/FUNCTION CODE ERROR
/
ADERR6  LAW     6       /ERROR CODE 6
        JMP*    (,MED+1 /TO MONITOR
/DATA MODE ERROR
ADERR7  LAW     7       /ERROR CODE 7
        JMP*    (,MED+1 /TO MONITOR
/THE ADINT ROUTINE MUST INCLUDE A ,SETUP
/FOR EACH FLAG ASSOCIATED WITH THE DEVICE
/
ADINIT  IDX     ADARGP  /IDX TO RETURN BUFF SIZE
        ,DEC
        LAC     (256    /STANDARD BUFFER SIZE (DECIMAL)
        ,OCT
        DAC*    ADARGP  /RETURN IT TO USER
        IDX     ADARGP
ADCMOD  CAL     57      /57=API CHANNEL
ADCKSM  16              /,SETUP IOPS FUNCTION CODE
ADCBP   ADSF            /ADC SKIP IOT
ADLBHP  ADCINT          /ADDR. OF INTERRUPT
ADUND   LAC     .+2     /SET-UP ONCE ONLY
ADWC    DAC     ADCMOD  /SKIP SET-UP CODE IF MORE
ADWPCT  JMP     ADSTOP  /,INITS ARE DONE
/
/STOP ADC ROUTINE CLEARS I/O UNDERWAY SWITCH
/
ADSTOP  DZM     ADUND
        JMP     ADIG%1  /RETURN
/
/THE PREVIOUS TAGS IN THE CAL AREA ARE USED FOR
/STORAGE DURING THE ACTUAL ,READ FUNCTION
/
/ADCKSM IS FOR STORING THE CHECKSUM
/ADCBP IS THE CURRENT BUFFER POINTER
/ADLBHP IS THE LINE BUFFER HEADER POINTER
/ADUND IS FOR DEVICE UNDERWAY SWITCH
/ADWC IS USED AS THE COUNTER
/ADWPCT IS USED TO STORE CURRENT WORD COUNT
/
        .EJECT
```

```
ADWAIT  LAC     ADUND
        SNA
        JMP     ADIGN1
/I/O UNDERWAY LOOP
ADBUSY  DBR
        JMP*    ADCALP
/
/
ADREAD  LAC     ADUND   /CHECK TO SEE IF I/O IS UNDERWAY
        SZA!CMA         /IF NOT SET IT WITH -1
        JMP     ADBUSY  /IT WAS SET,GO BACK TO CAL
        DAC     ADUND   /SET IT
        LAC*    ADCALP  /LOOK AT MODE
        AND     (7000   /BITS 6-8 ONLY
        SZA             /IOPS BINARY?
        JMP     ADERR7  /NO, ERROR
        LAC*    ADARGP  /GET LINE BUFFER HEADER POINTER
        DAC     ADCBP   /STORE IT
        DAC     ADLBHP  /ALSO STORE IT FOR LATER HEADER
        IDX     ADARGP  /INCREMENT ARG. POINTER
        LAC*    ADARGP  /GET -L,B,W,C(2'S COMP)
        DAC     ADWC    /STORE IT IN WORD COUNTER
        DZM     ADWPCT  /ZERO WORD COUNT REG.
        DZM     ADCKSM  /ZERO CHECKSUM REG.
        IDX     ADCBP   /GET PAST HEADER PAIR
        IDX     ADCBP   /NOW POINTING AT BEGINNING OF
                        /BUFFER
        ADSC            /START UP DEVICE
ADIGN2  IDX     ADARGP  /INCR. FOR EXIT
ADIGN1  DBR             /BREAK FROM LEVEL 4
        JMP*    ADARGP  /RETURN AFTER CAL
/INTERRUPT HANDLER FOR API OR PIC
/
ONLY1   LAC     (NOP
        DAC     ADCION
        DAC     ADCONT
        DAC     ADSWCH
        DAC     IGNRPI
        JMP     COMMON
ADCPIC  DAC     ADCAC   /SAVE AC
        LAC*    (0)     /SAVE PC,LINK,EX. MODE
        DAC     ADCOUT  /MEM,PROT,
        .EJECT
```

```
        JMP     COMMON
ADCINT  JMP     ADCPIC  /PIC ENTRY
        DAC     ADCAC   /API ENTRY,SAVE AC
        LAC     ADCINT  /SAVE PC,LINK,EX,MODE
        DAC     ADCOUT  /MEM,PROT
IGNRPI  JMP     ONLY1
COMMON  ADRB            /READ CONVERTER BUFFER
ADCION  ION             /ENABLE PIC FOR OTHER DEVICES
        DAC*    ADCBP   /STORE DATA IN USER BUFFER
        IDX     ADCBP   /INC, BUFFER POINTER
        IDX     ADWPCT  /INC, WORD PAIR COUNTER
        TAD     ADCKSM  /ADD CHECKSUM
        DAC     ADCKSM  /STORE IT
        ISZ     ADWC    /IS I/O COMPLETE
        JMP     ADCONT  /NO KEEP GOING
        LAC     ADWPCT  /YES,COMPUTE WORD COUNT PAIR
        IAC             /MAY BE ODD
        SWHA            /TO TOP HALF
        RAR             /MAKE WD. PRS,
        AND     (377000 /8 BITS ONLY
        DAC*    ADLBHP  /STORE IN HEADER #1
        IDX     ADLBHP  /INC, TO STORE CKSUM
        TAD     ADCKSM  /ADD WORD PAIR COUNT
        DAC*    ADLBHP  /STORE IN HEADER #2
        DZM     ADUND   /CLEAR DEVICE UNDERWAY
        JMP     ADDISM  /EXIT
ADCONT  IOF             /DISABLE PIC OR NOP
        ADSC            /BEFORE INTERRUPT FROM THIS IOT OCCURS
/INTERRUPT HANDLER DISMISS RTE
/
ADDISM  LAC     ADCAC   /RESTORE AC
        .EJECT
```

```
ADSWCH  ION             /ION OR NOP
        DBR             /DEBREAK AND RESTORE
        JMP*    ADCOUT  /LINK,EX,MODE,MEM,PROT
ADCALP  0               /ADD CAL POINTER
ADARGP  0               /ADD ARGUMENT POINTER
ADCOUT  0               /PC,L,FM,MP
ADCAC   0               /AC SAVED HERE
/
        .END
```

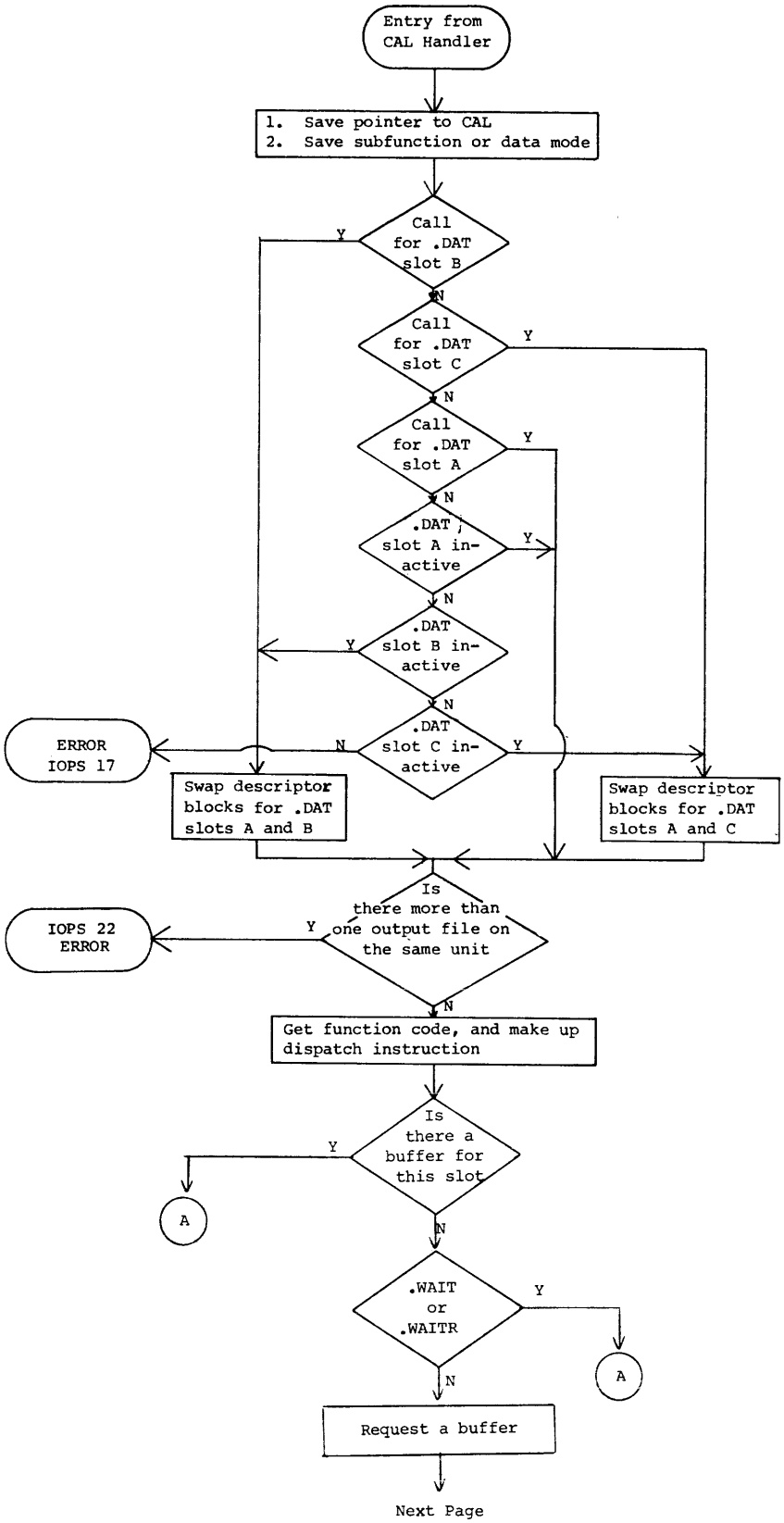
CHAPTER 9
XVM/DOS BATCHING FACILITIES

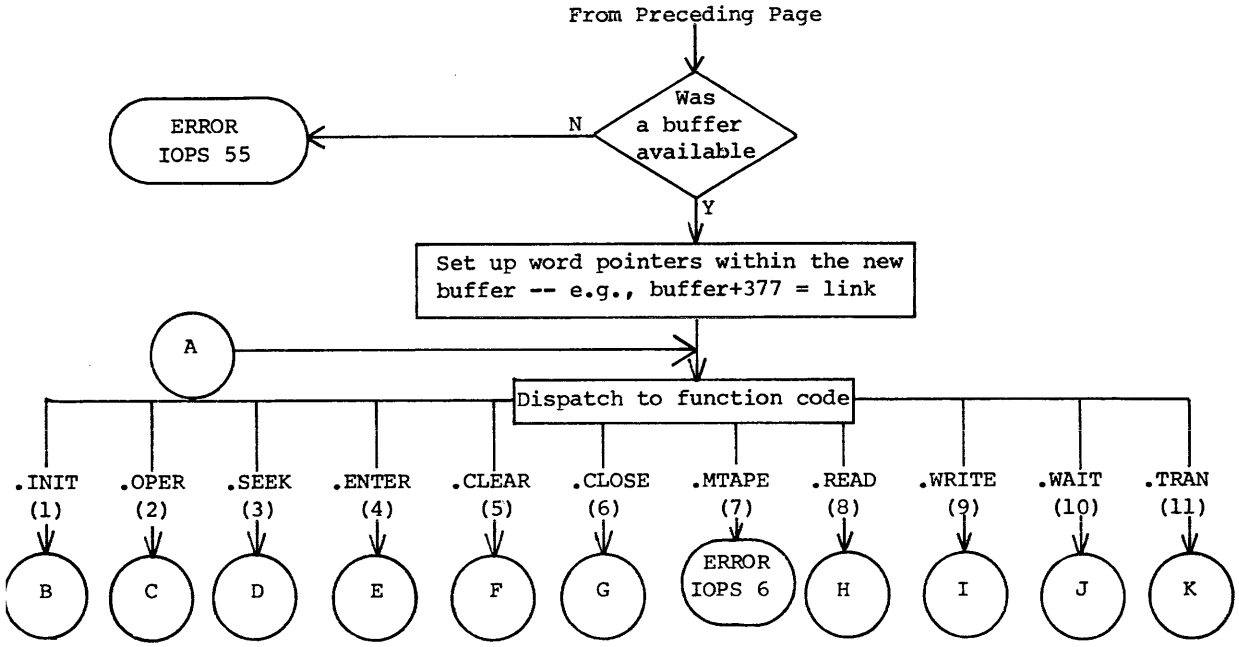
(To Be Supplied At A Later Date)

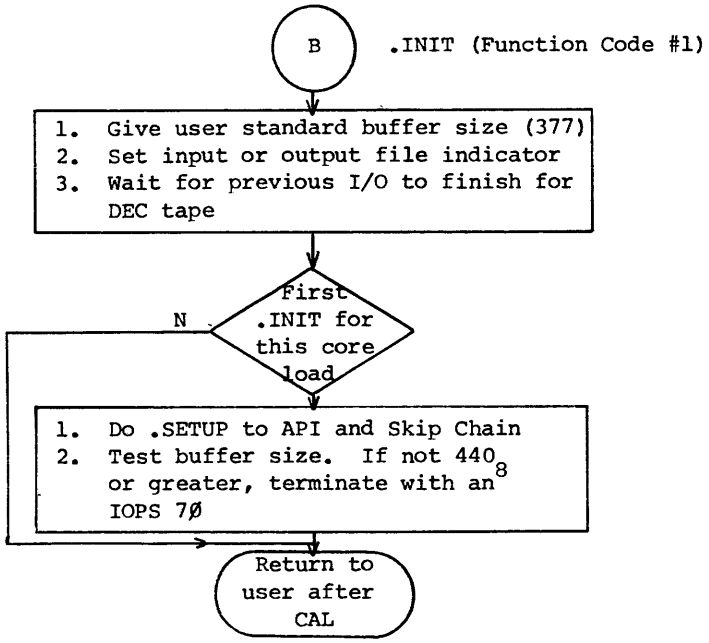
APPENDIX A

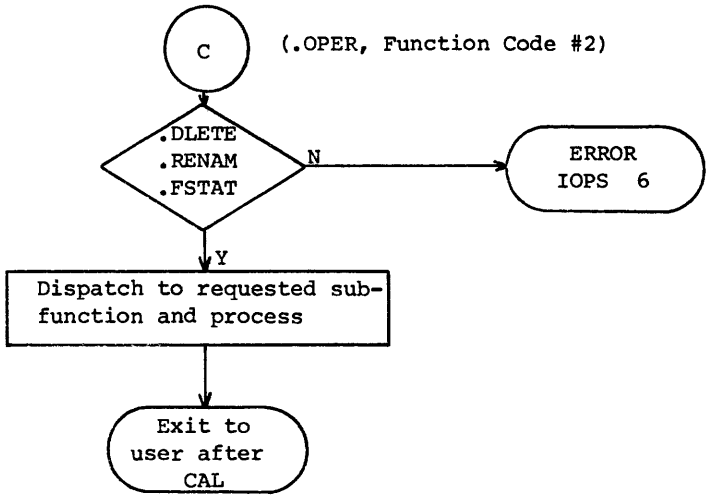
DECTAPE "A" HANDLER (DTA.)

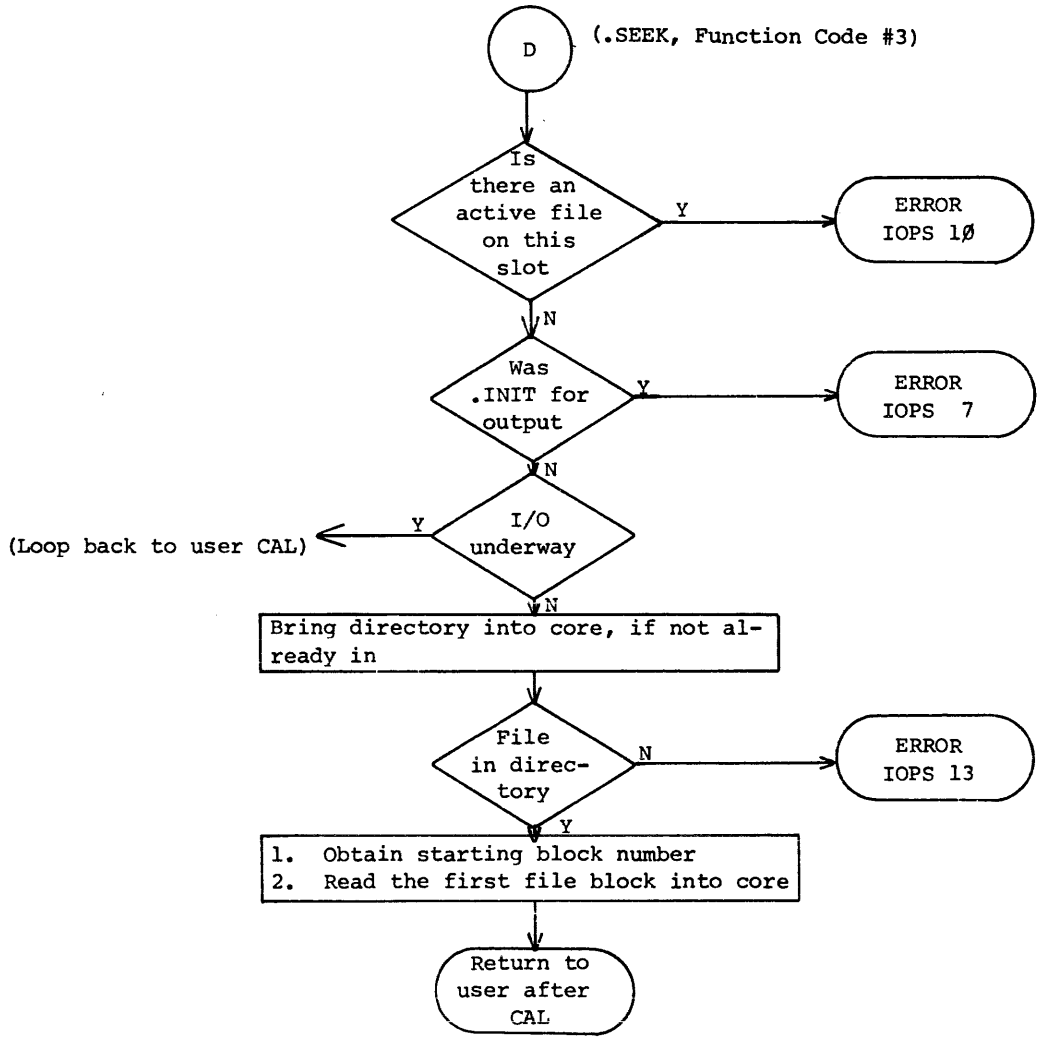
The following simplified charts describe the approximate flow of control through the DECtape "A" Handler.

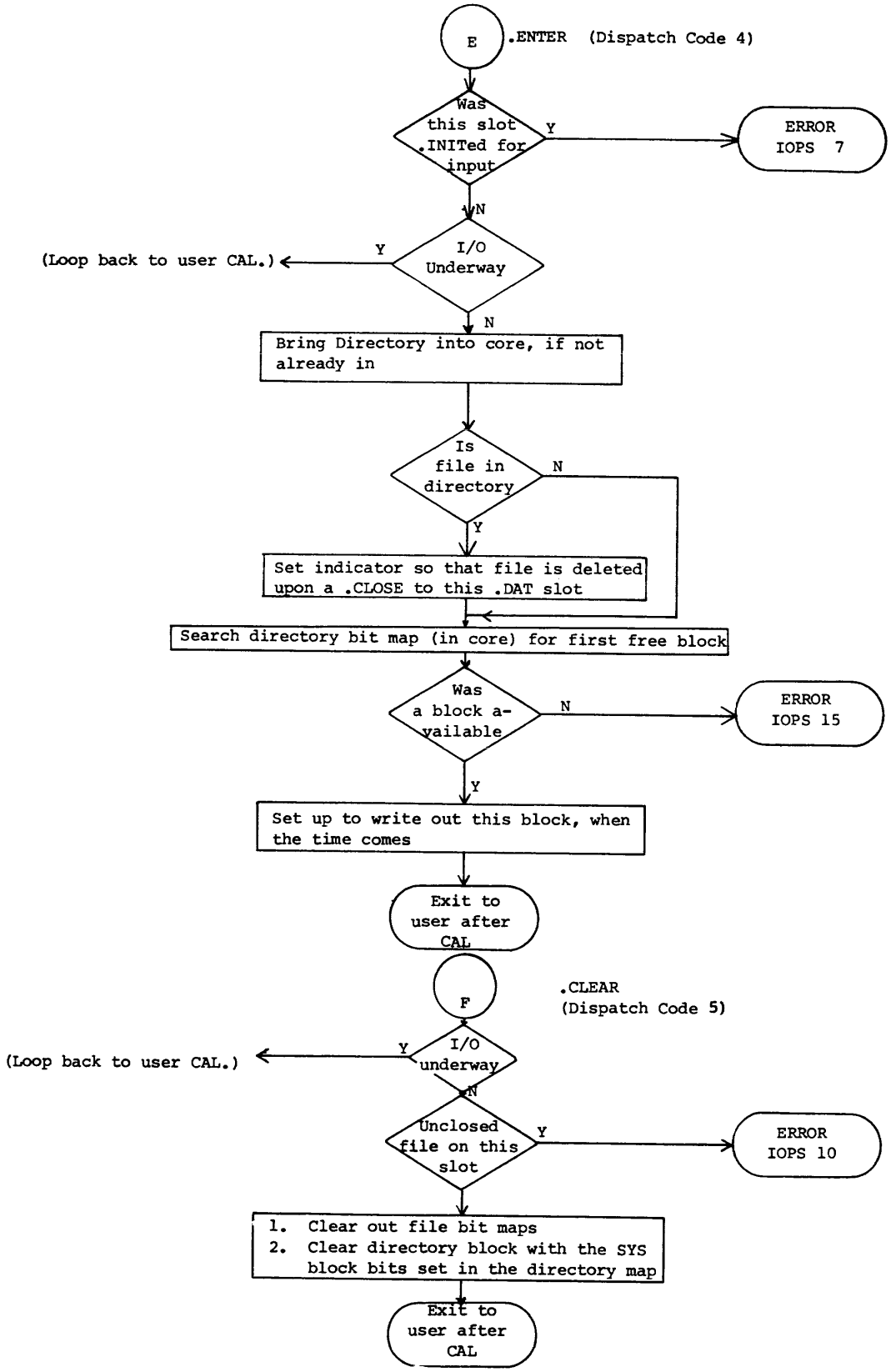

Entry from CAL Handler
1. Save pointer to CAL
2. Save subfunction or data mode
Call for .DAT slot B
Call for .DAT slot C
Call for .DAT slot A
.DAT slot A inactive
.DAT slot B inactive
.DAT slot C inactive
ERROR IOPS 17
Swap descriptor blocks for .DAT slots A and B
Swap descriptor blocks for .DAT slots A and C
Is there more than one output file on the same unit
IOPS 22 ERROR
Get function code, and make up dispatch instruction
Is there a buffer for this slot
A
.WAIT or .WAITR
A
Request a buffer
Next Page
Y
N

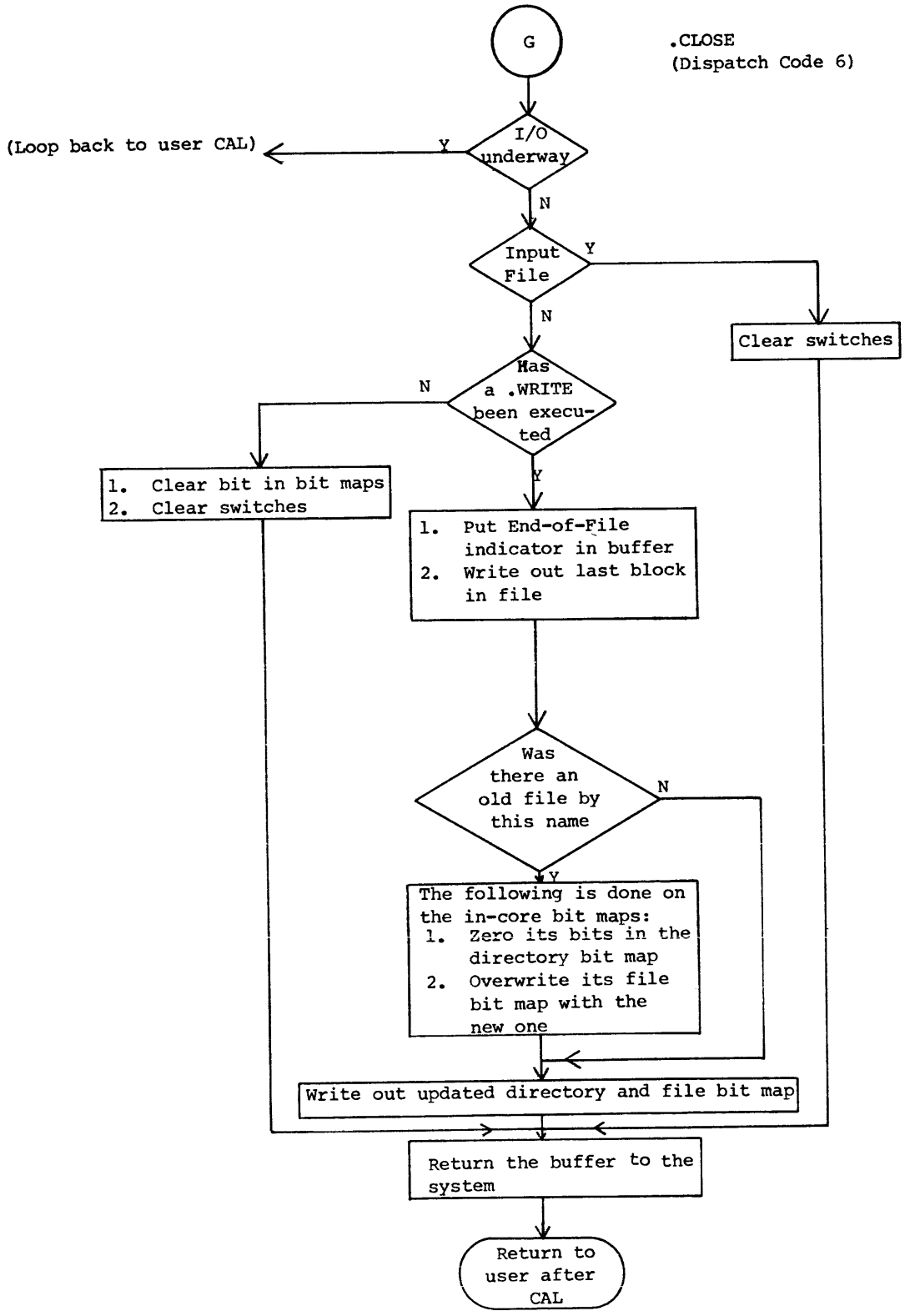

From Preceding Page
Was a buffer available
N
ERROR IOPS 55
Y
Set up word pointers within the new buffer -- e.g., buffer+377 = link
A
Dispatch to function code
.INIT (1)
.OPER (2)
.SEEK (3)
.ENTER (4)
.CLEAR (5)
.CLOSE (6)
.MTAPE (7)
.READ (8)
.WRITE (9)
.WAIT (10)
.TRAN (11)
B
C
D
E
F
G
ERROR IOPS 6
H
I
J
K

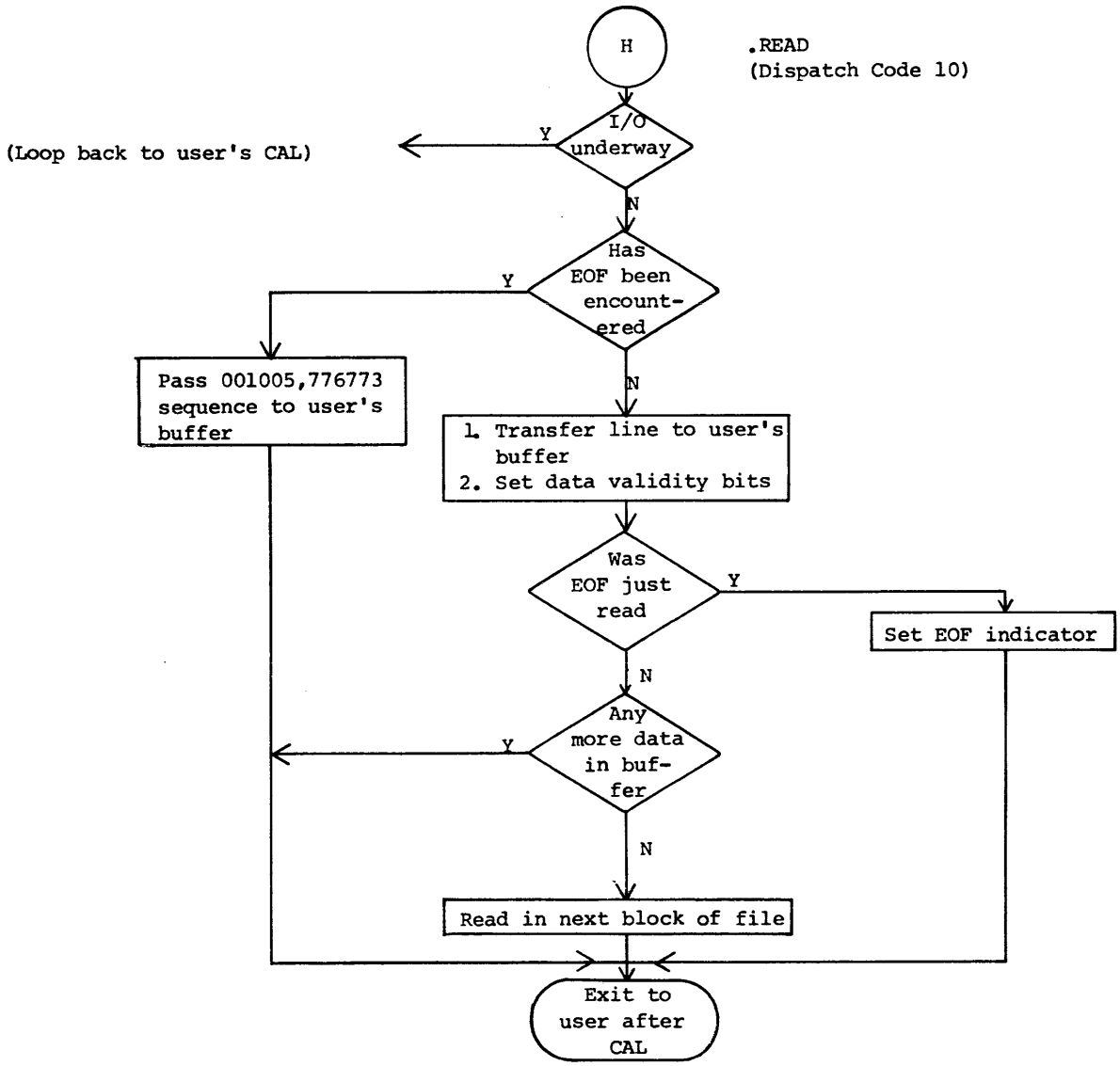

B
.INIT (Function Code #1)
1. Give user standard buffer size (377)
2. Set input or output file indicator
3. Wait for previous I/O to finish for DEC tape
First .INIT for this core load
N
1. Do .SETUP to API and Skip Chain
2. Test buffer size. If not 440 8 or greater, terminate with an IOPS 7Ø
Return to user after CAL

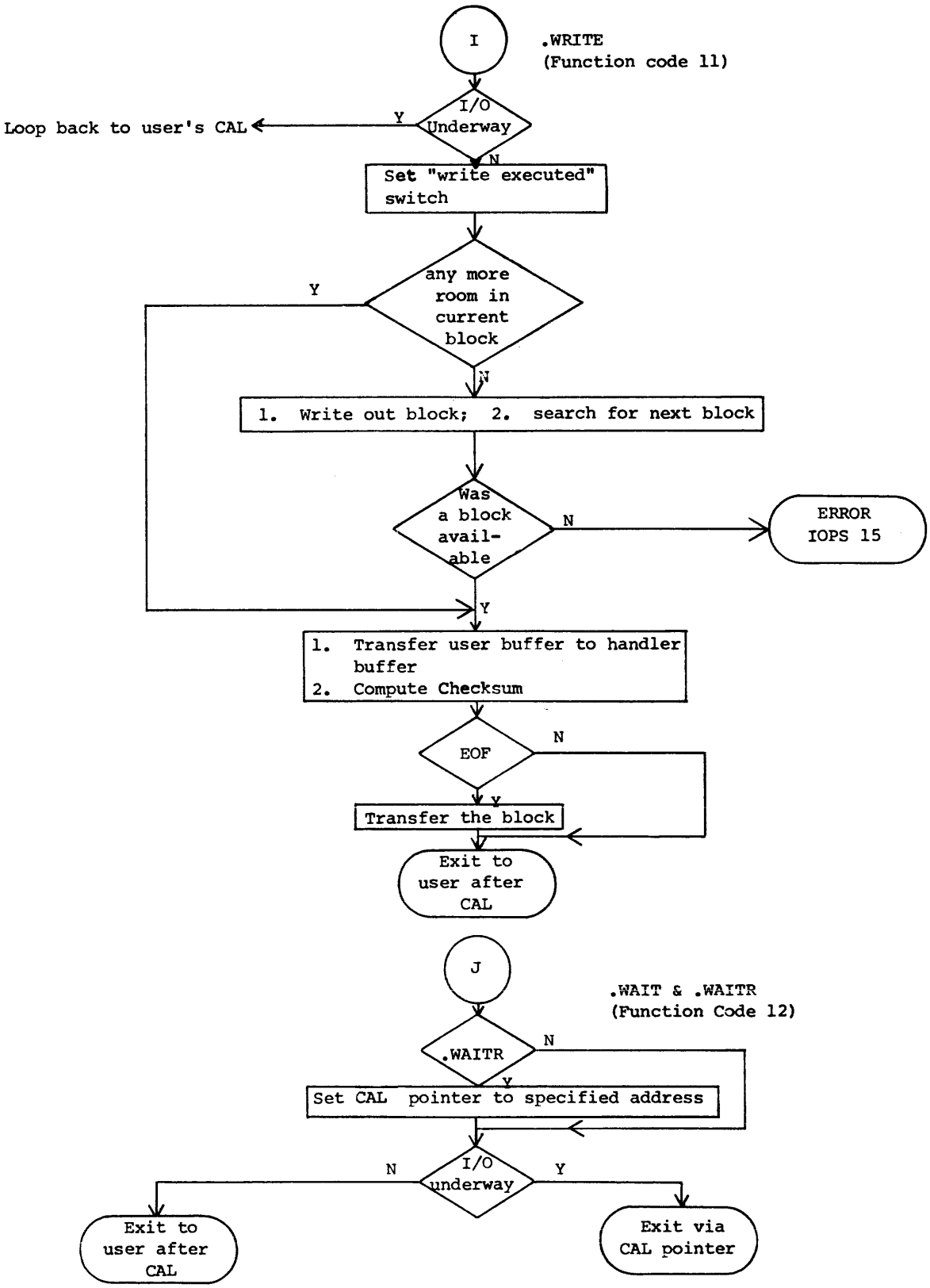

C
(.OPER, Function Code #2)
.DLETE
.RENAM
.FSTAT
N
ERROR
IOPS 6
Y
Dispatch to requested sub-function and process
Exit to user after CAL

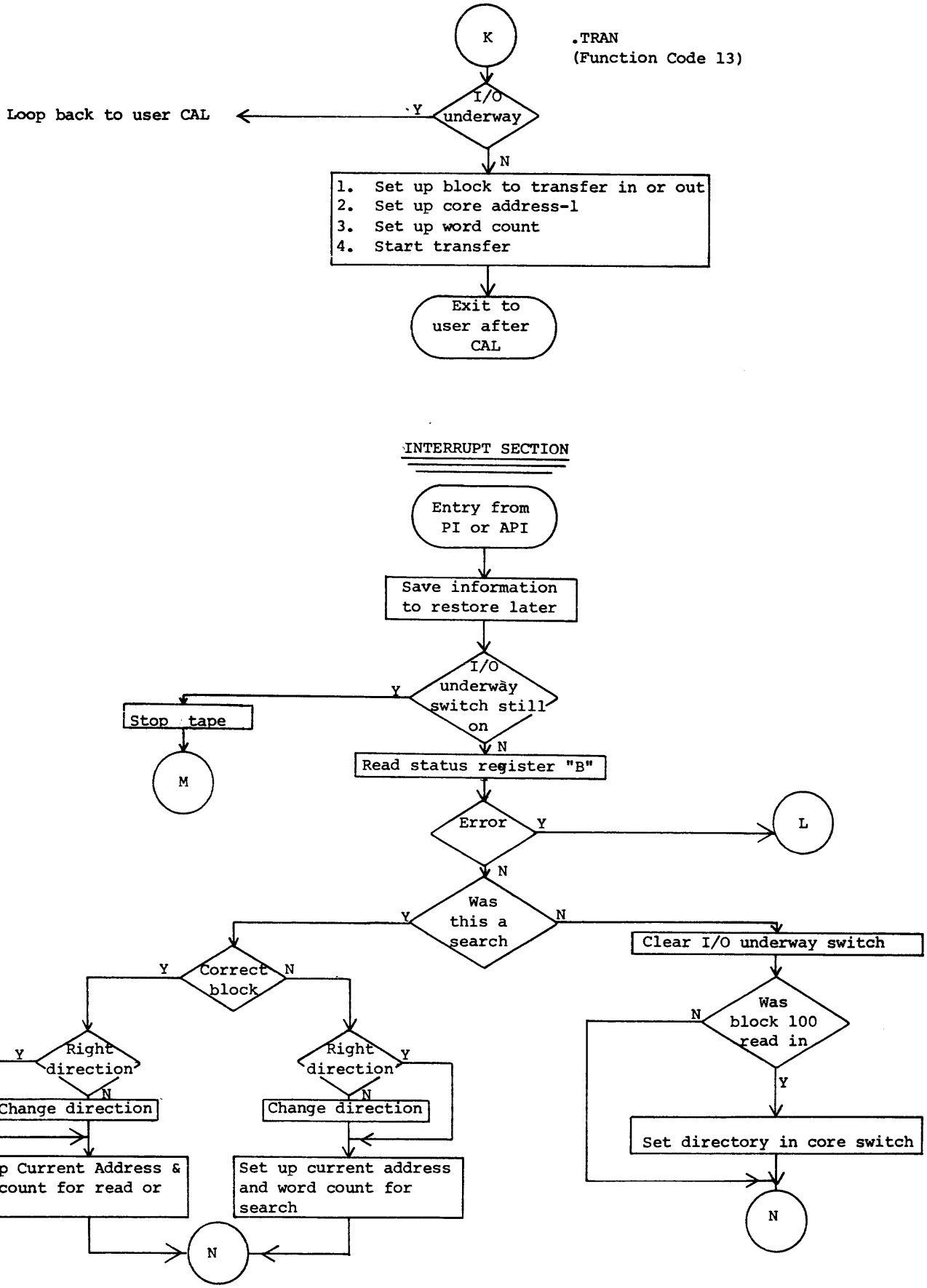

D
(.SEEK, Function Code #3)
Is there an active file on this slot
Y
ERROR
IOPS 1Ø
N
Was .INIT for output
Y
ERROR
IOPS 7
N
I/O underway
Y
(Loop back to user CAL)
N
Bring directory into core, if not already in
File in directory
N
ERROR
IOPS 13
Y
1. Obtain starting block number
2. Read the first file block into core
Return to user after CAL

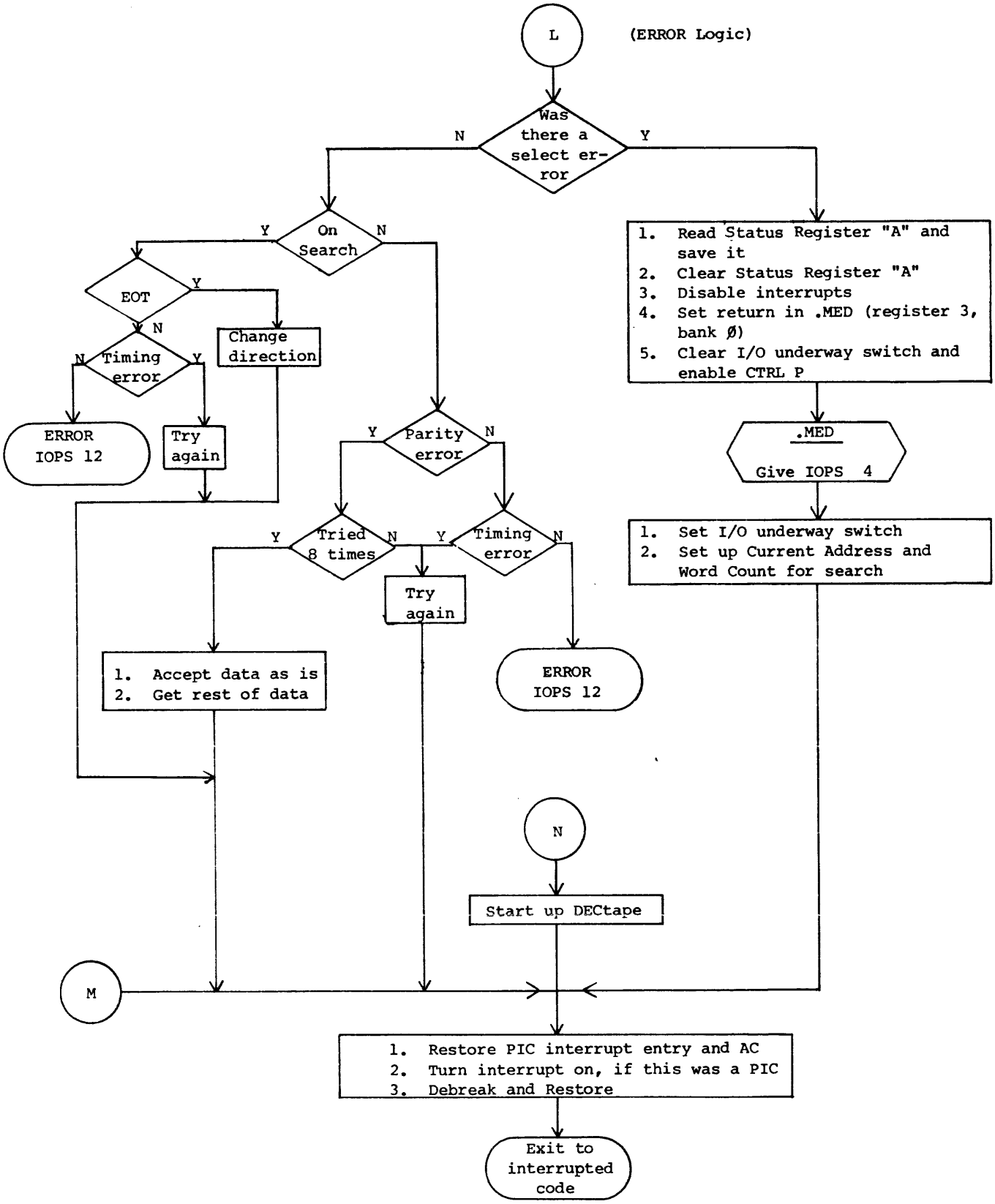

E
.ENTER (Dispatch Code 4)
Was this slot .INITed for input
Y
ERROR IOPS 7
N
I/O Underway
Y
(Loop back to user CAL.)
N
Bring Directory into core, if not already in
Is file in directory
N
Y
Set indicator so that file is deleted upon a .CLOSE to this .DAT slot
Search directory bit map (in core) for first free block
Was a block available
N
ERROR IOPS 15
Y
Set up to write out this block, when the time comes
Exit to user after CAL
F
.CLEAR
(Dispatch Code 5)
I/O underway
Y
(Loop back to user CAL.)
N
Unclosed file on this slot
Y
ERROR IOPS 10
1. Clear out file bit maps
2. Clear directory block with the SYS block bits set in the directory map
Exit to user after CAL


G
.CLOSE
(Dispatch Code 6)
I/O underway
Y
(Loop back to user CAL)
N
Input File
Y
N
Clear switches
Has a .WRITE been executed
N
Y
1. Clear bit in bit maps
2. Clear switches
1. Put End-of-File indicator in buffer
2. Write out last block in file
Was there an old file by this name
N
Y
The following is done on the in-core bit maps:
1. Zero its bits in the directory bit map
2. Overwrite its file bit map with the new one
Write out updated directory and file bit map
Return the buffer to the system
Return to user after CAL


H
.READ
(Dispatch Code 10)
I/O underway
Y
(Loop back to user's CAL)
N
Has EOF been encount-ered
Y
N
Pass 001005,776773 sequence to user's buffer
1. Transfer line to user's buffer
2. Set data validity bits
Was EOF just read
Y
Set EOF indicator
N
Any more data in buf-fer
Y
N
Read in next block of file
Exit to user after CAL


I
.WRITE
(Function code 11)
I/O Underway
Y
Loop back to user's CAL
N
Set "write executed" switch
any more room in current block
Y
N
1. Write out block; 2. search for next block
Was a block avail-able
N
ERROR IOPS 15
Y
1. Transfer user buffer to handler buffer
2. Compute Checksum
EOF
N
Y
Transfer the block
Exit to user after CAL
J
.WAIT & .WAITR
(Function Code 12)
.WAITR
N
Y
Set CAL pointer to specified address
I/O underway
N
Y
Exit to user after CAL
Exit via CAL pointer

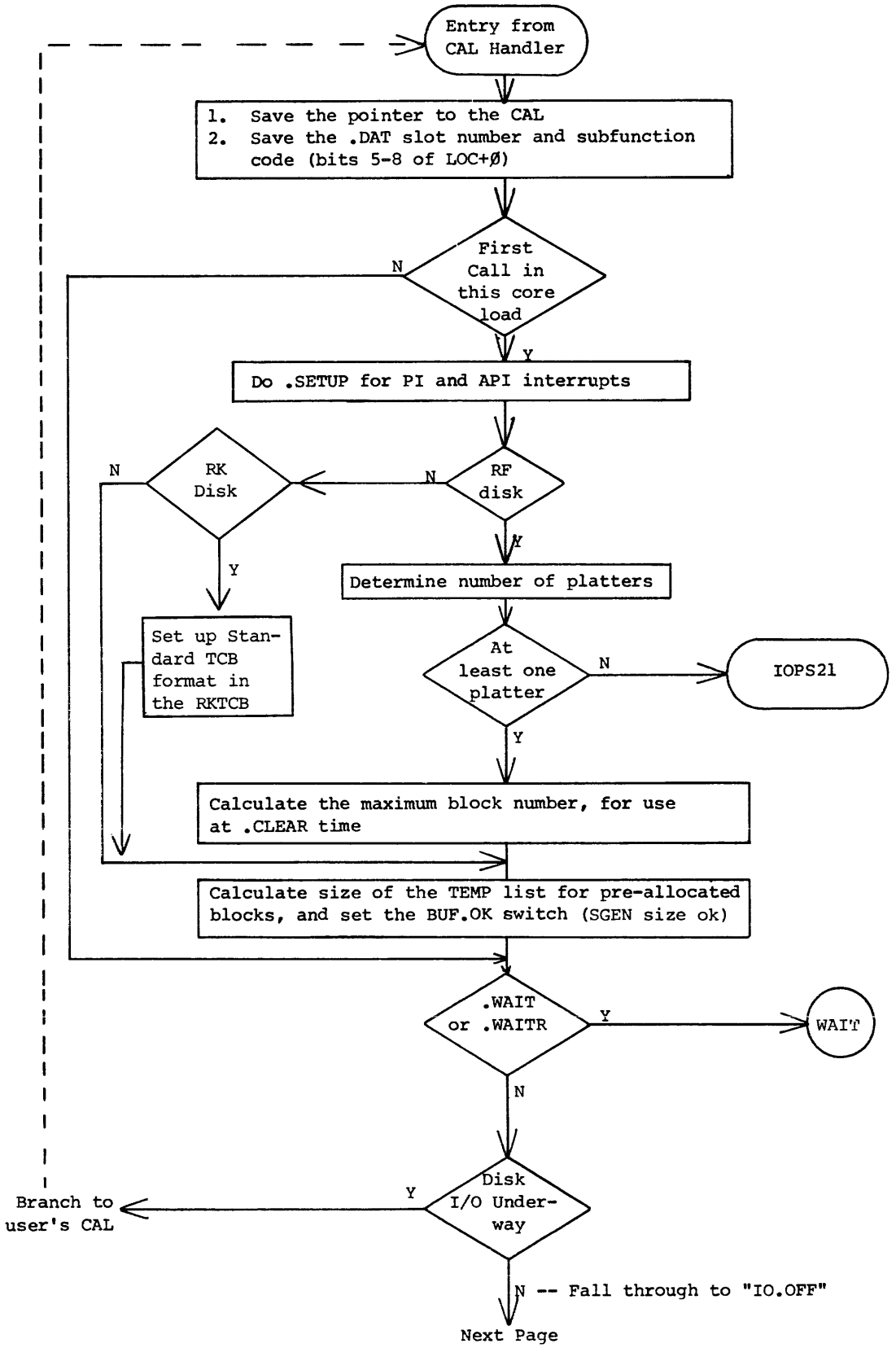

K
.TRAN
(Function Code 13)
I/O underway
Y
Loop back to user CAL
N
1. Set up block to transfer in or out
2. Set up core address-1
3. Set up word count
4. Start transfer
Exit to user after CAL
INTERRUPT SECTION
Entry from PI or API
Save information to restore later
I/O underway switch still on
Y
Stop tape
M
N
Read status register "B"
Error
Y
L
N
Was this a search
Y
N
Clear I/O underway switch
Correct block
Y
N
Was block 100 read in
N
Y
Right direction
Y
N
Change direction
Right direction
Y
N
Change direction
et up Current Address & ord count for read or rite
Set up current address and word count for search
Set directory in core switch
N
N

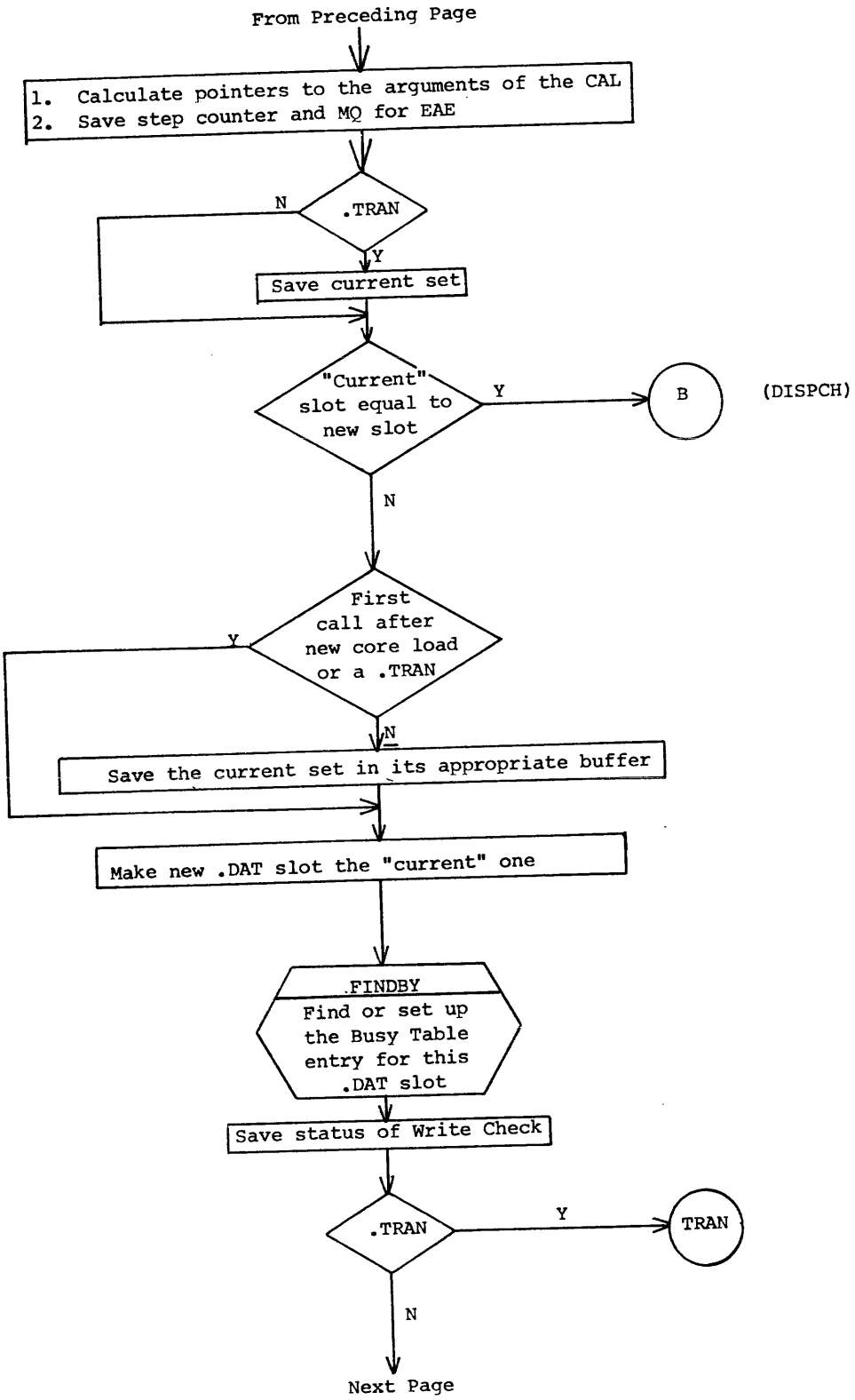

L
(ERROR Logic)
Was there a select error
N
Y
On Search
Y
N
EOT
Y
N
Change direction
Timing error
N
Y
ERROR IOPS 12
Try again
Parity error
Y
N
Tried 8 times
Y
N
Timing error
Y
N
Try again
1. Accept data as is
2. Get rest of data
ERROR IOPS 12
1. Read Status Register "A" and save it
2. Clear Status Register "A"
3. Disable interrupts
4. Set return in .MED (register 3, bank Ø)
5. Clear I/O underway switch and enable CTRL P
.MED
Give IOPS 4
1. Set I/O underway switch
2. Set up Current Address and Word Count for search
N
Start up DECtape
M
1. Restore PIC interrupt entry and AC
2. Turn interrupt on, if this was a PIC
3. Debreak and Restore
Exit to interrupted code

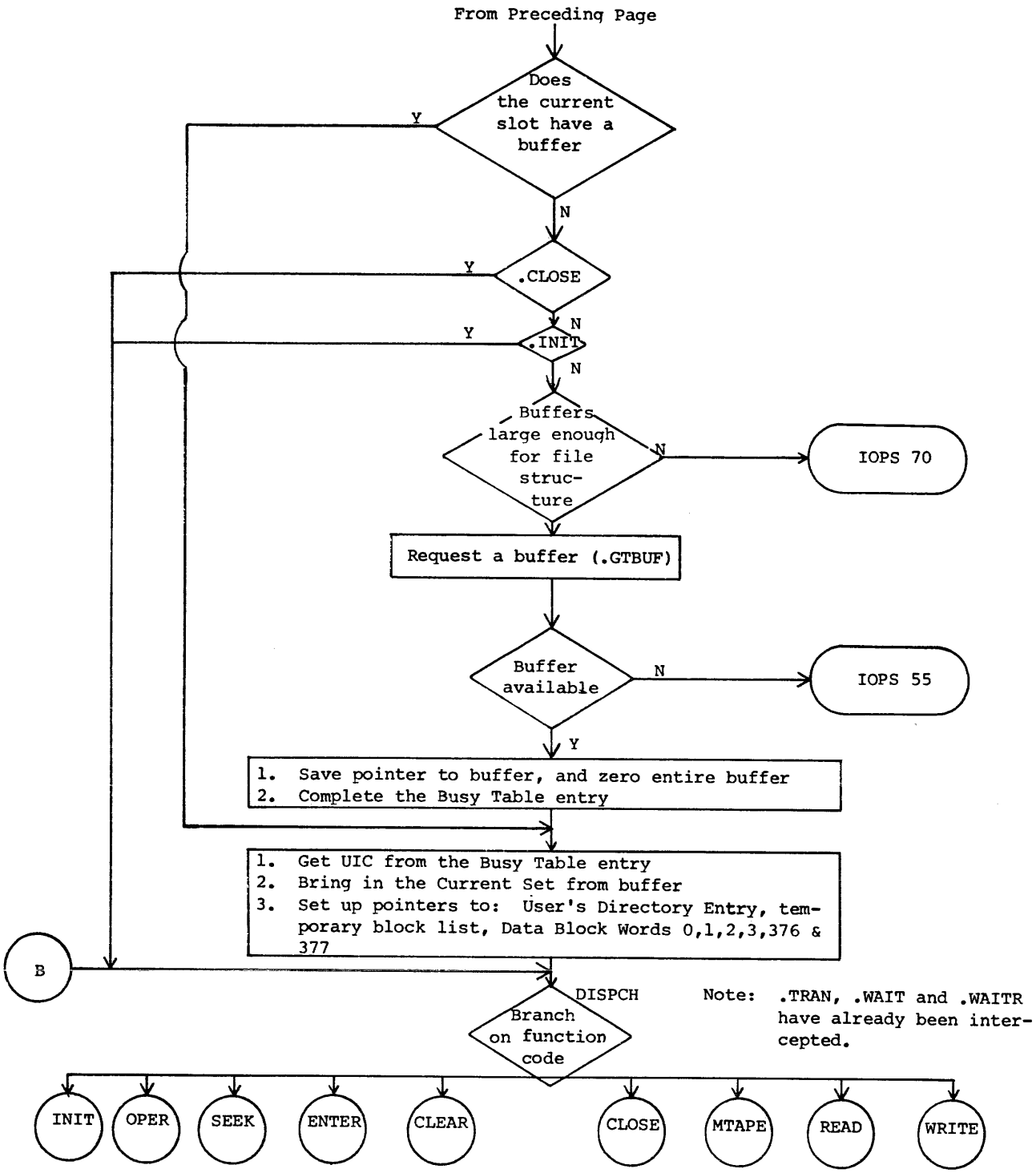
APPENDIX B

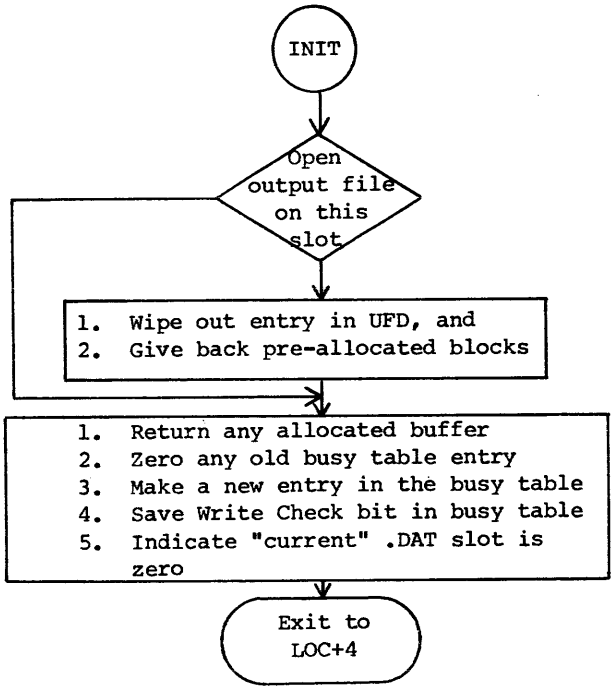
DISK "A" HANDLERS

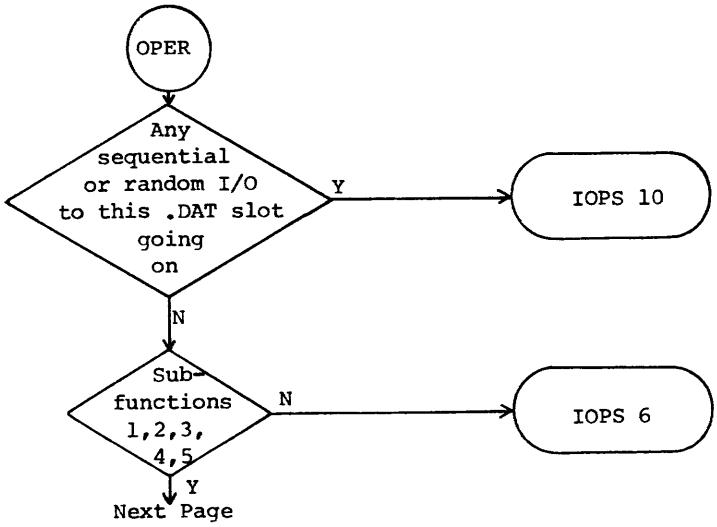
The following simplified charts describe the approximate flow of control through the Disk "A" Handlers.

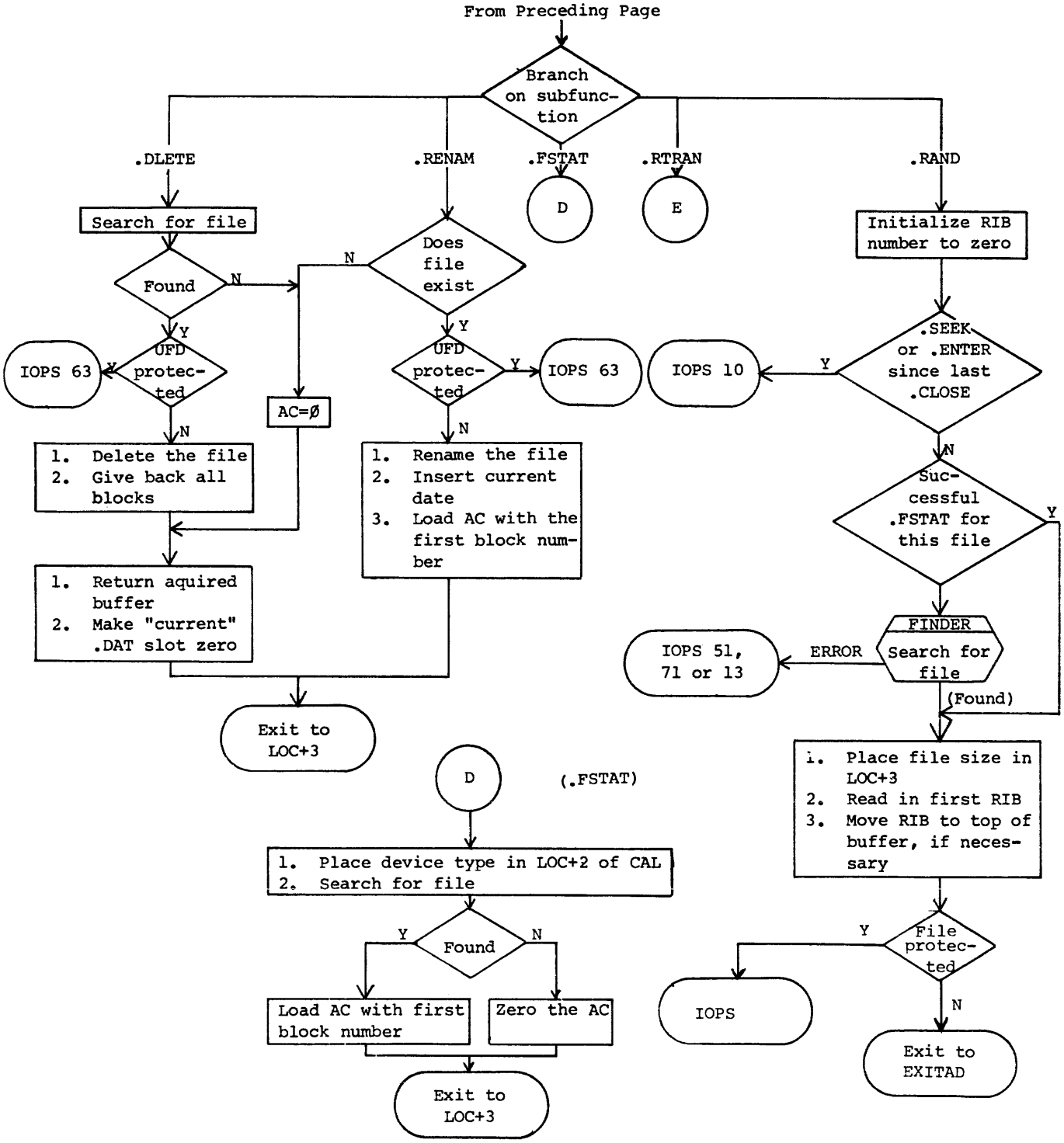

Entry from CAL Handler
1. Save the pointer to the CAL
2. Save the .DAT slot number and subfunction code (bits 5-8 of LOC+Ø)
First Call in this core load
N
Y
Do .SETUP for PI and API interrupts
RK Disk
N
RF disk
N
Y
Y
Determine number of platters
Set up Standard TCB format in the RKTCB
At least one platter
N
IOPS21
Y
Calculate the maximum block number, for use at .CLEAR time
Calculate size of the TEMP list for pre-allocated blocks, and set the BUF.OK switch (SGEN size ok)
.WAIT or .WAITR
Y
WAIT
N
Disk I/O Underway
Y
Branch to user's CAL
N -- Fall through to "IO.OFF"
Next Page

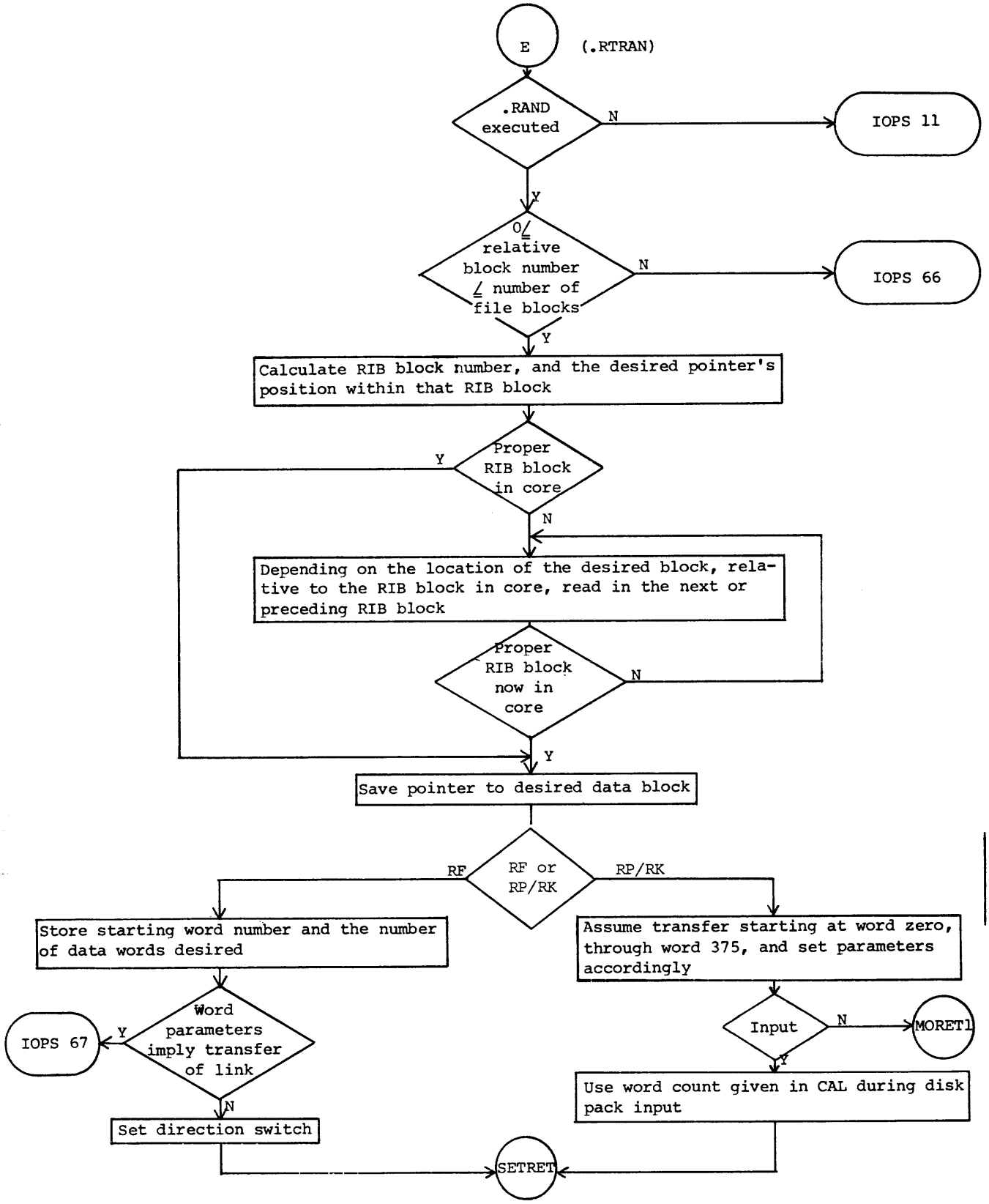

From Preceding Page
1. Calculate pointers to the arguments of the CAL
2. Save step counter and MQ for EAE
.TRAN
N
Y
Save current set
"Current" slot equal to new slot
Y
B
(DISPCH)
N
First call after new core load or a .TRAN
Y
N
Save the current set in its appropriate buffer
Make new .DAT slot the "current" one
.FINDBY
Find or set up the Busy Table entry for this .DAT slot
Save status of Write Check
.TRAN
Y
TRAN
N
Next Page

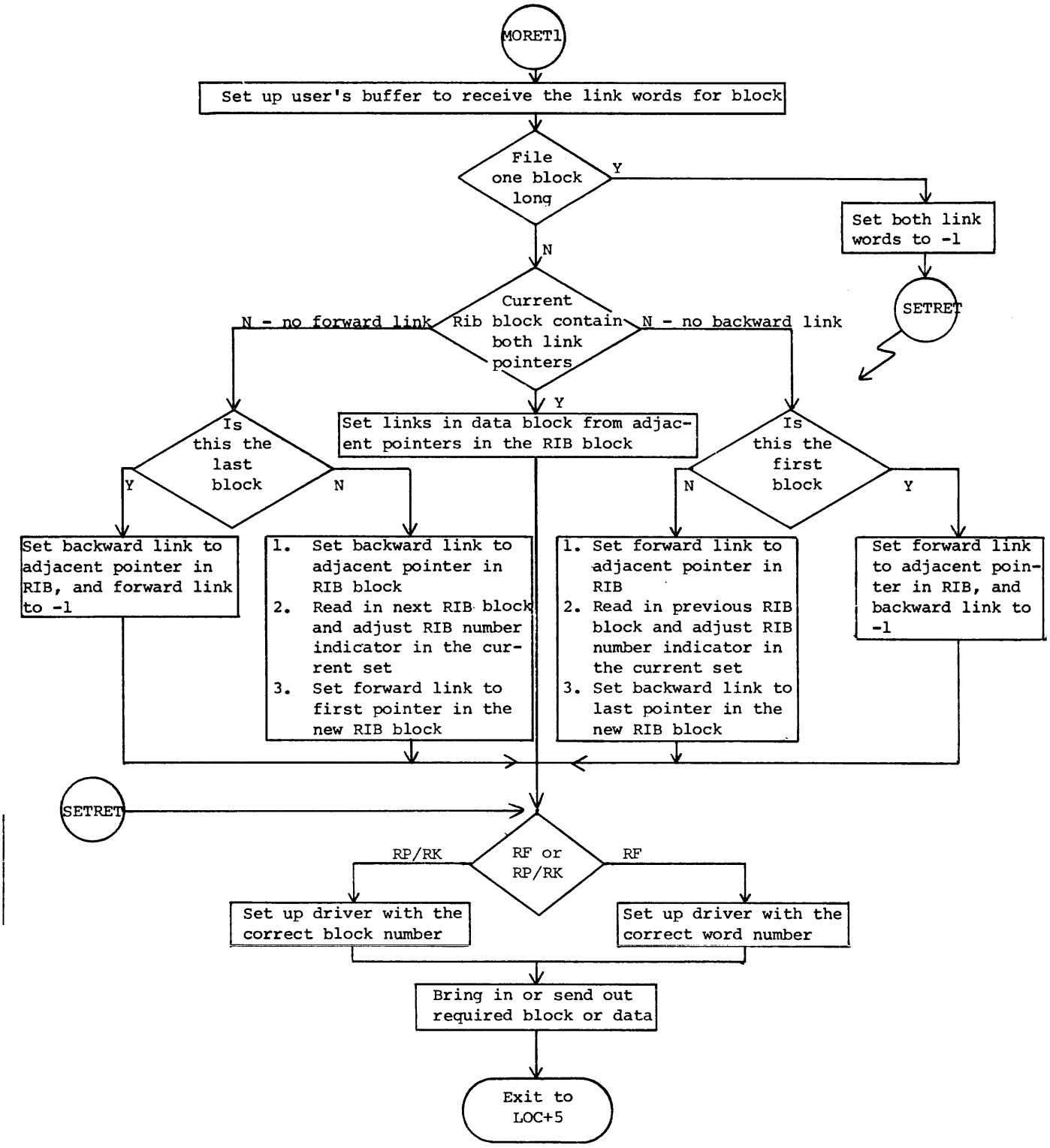

From Preceding Page
Does the current slot have a buffer
Y
N
.CLOSE
Y
N
.INIT
Y
N
Buffers large enough for file struc-ture
N
IOPS 70
Request a buffer (.GTBUF)
Buffer available
N
IOPS 55
Y
1. Save pointer to buffer, and zero entire buffer
2. Complete the Busy Table entry
1. Get UIC from the Busy Table entry
2. Bring in the Current Set from buffer
3. Set up pointers to: User's Directory Entry, temporary block list, Data Block Words 0,1,2,3,376 & 377
B
DISPCH
Branch on function code
Note: .TRAN, .WAIT and .WAITR have already been intercepted.
INIT
OPER
SEEK
ENTER
CLEAR
CLOSE
MTAPE
READ
WRITE

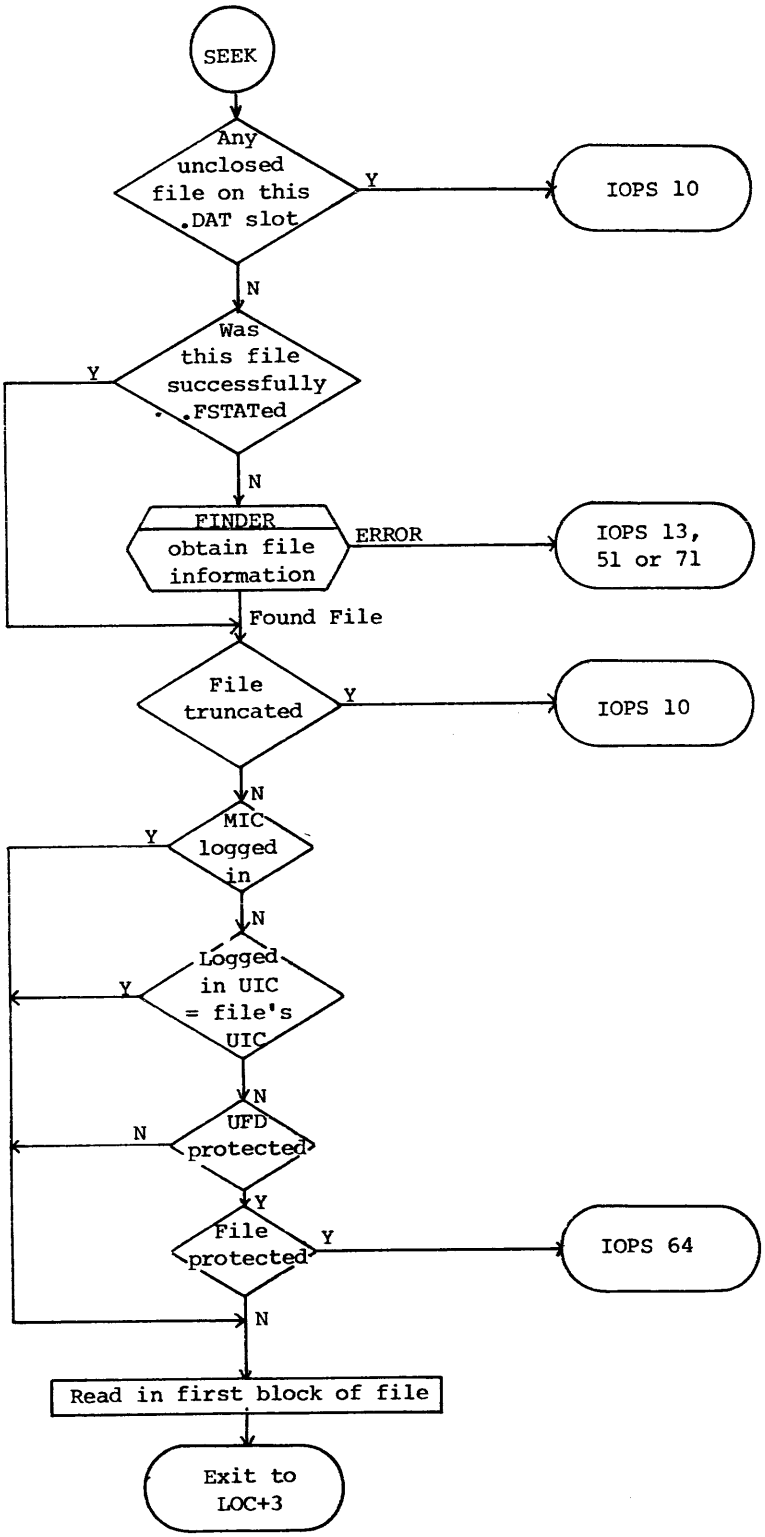

INIT
Open output file on this slot
1. Wipe out entry in UFD, and
2. Give back pre-allocated blocks
1. Return any allocated buffer
2. Zero any old busy table entry
3. Make a new entry in the busy table
4. Save Write Check bit in busy table
5. Indicate "current" .DAT slot is zero
Exit to LOC+4

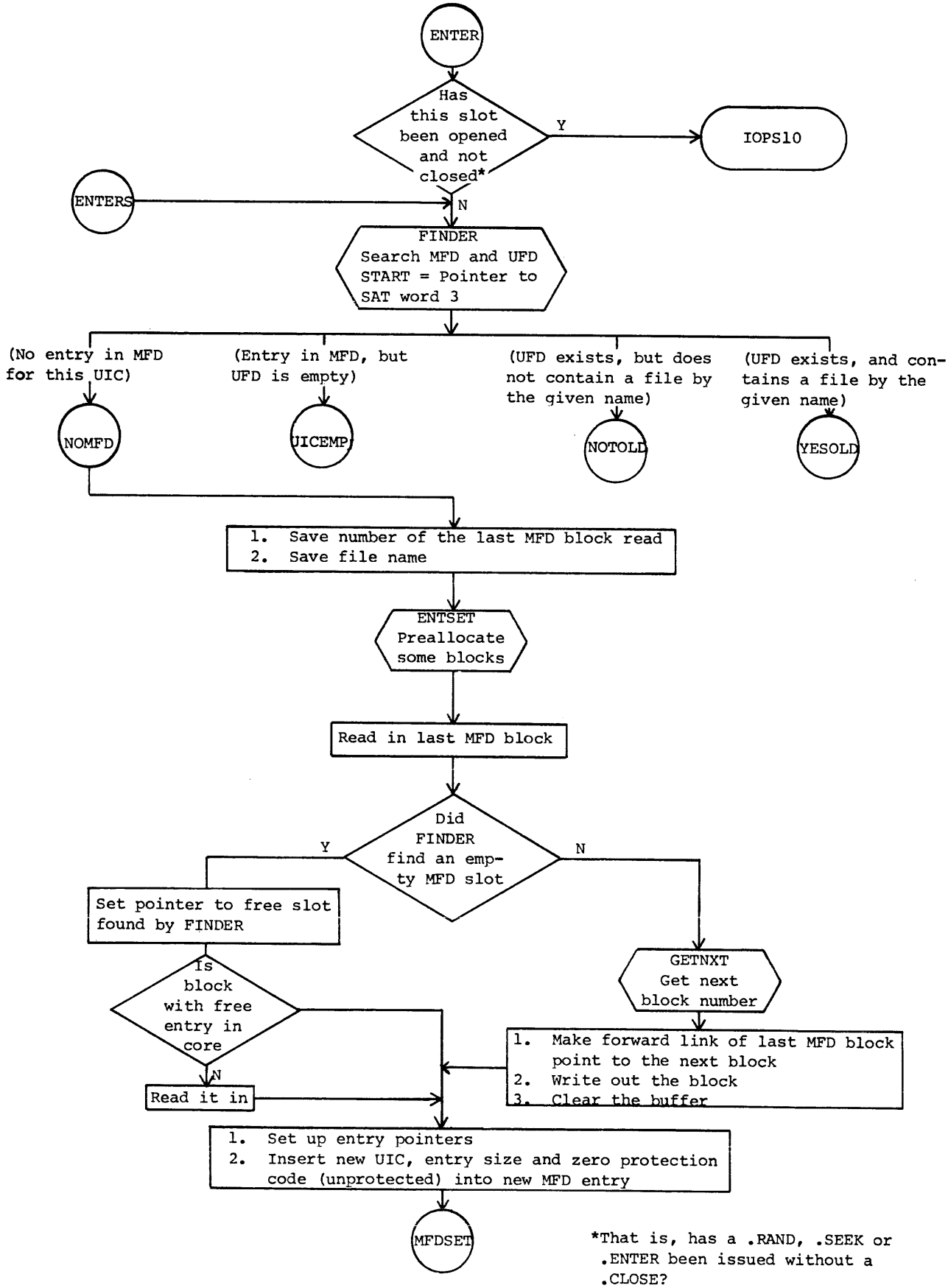

OPER
Any sequential or random I/O to this .DAT slot going on
Y
IOPS 10
N
Sub-functions 1,2,3, 4,5
N
IOPS 6
Y
Next Page

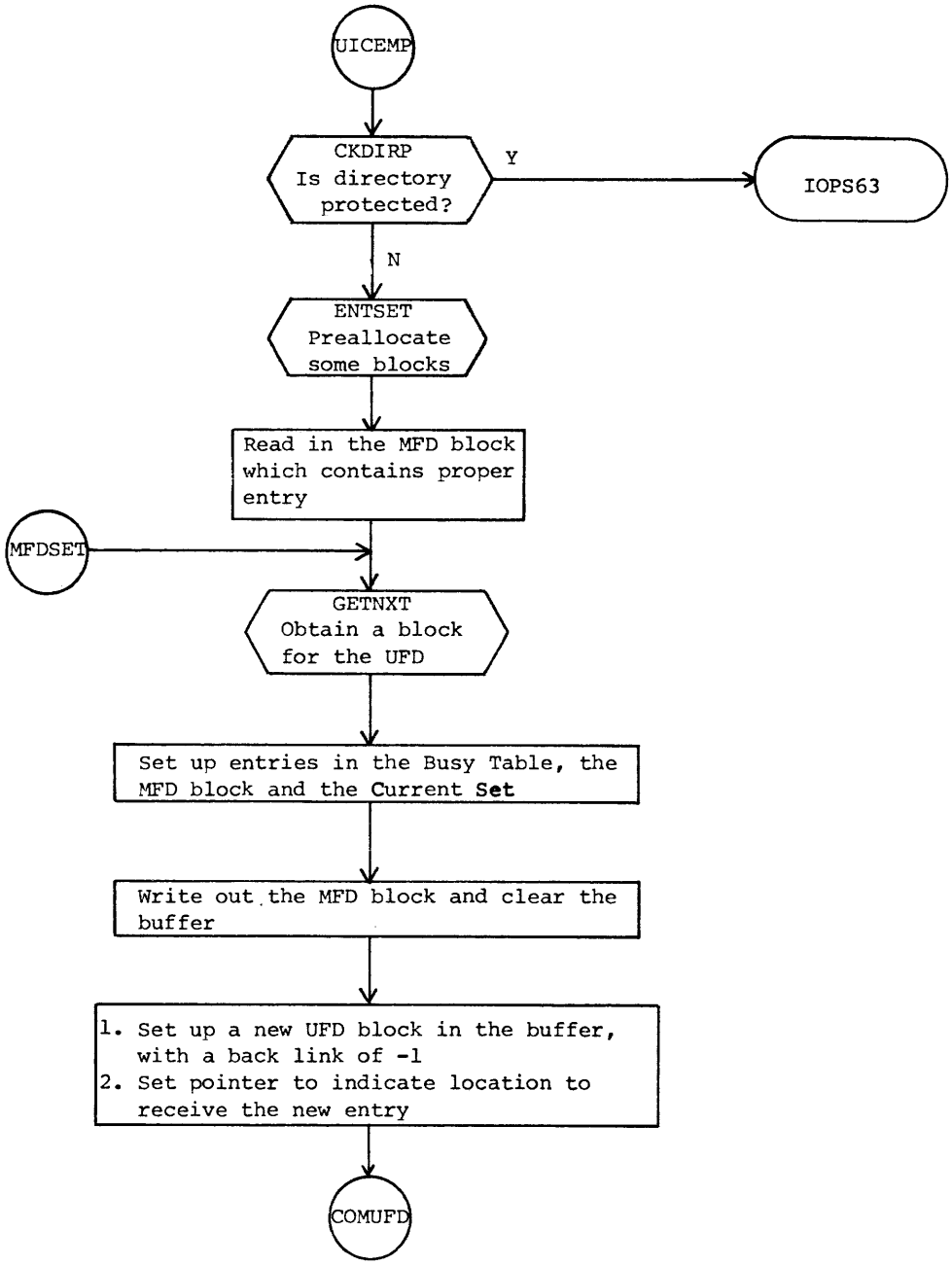

From Preceding Page
Branch on subfunction
.DLETE
.RENAM
.FSTAT
.RTRAN
.RAND
D
E
Search for file
Found
N
Y
IOPS 63
UFD protected
Y
N
1. Delete the file
2. Give back all blocks
AC=Ø
1. Return aquired buffer
2. Make "current" .DAT slot zero
Exit to LOC+3
Does file exist
N
Y
UFD protected
Y
IOPS 63
N
1. Rename the file
2. Insert current date
3. Load AC with the first block number
Initialize RIB number to zero
.SEEK or .ENTER since last .CLOSE
Y
IOPS 10
N
Successful .FSTAT for this file
Y
FINDER
Search for file
ERROR
IOPS 51, 71 or 13
(Found)
1. Place file size in LOC+3
2. Read in first RIB
3. Move RIB to top of buffer, if necessary
File protected
Y
IOPS
N
Exit to EXITAD
D
(.FSTAT)
1. Place device type in LOC+2 of CAL
2. Search for file
Found
Y
N
Load AC with first block number
Zero the AC
Exit to LOC+3

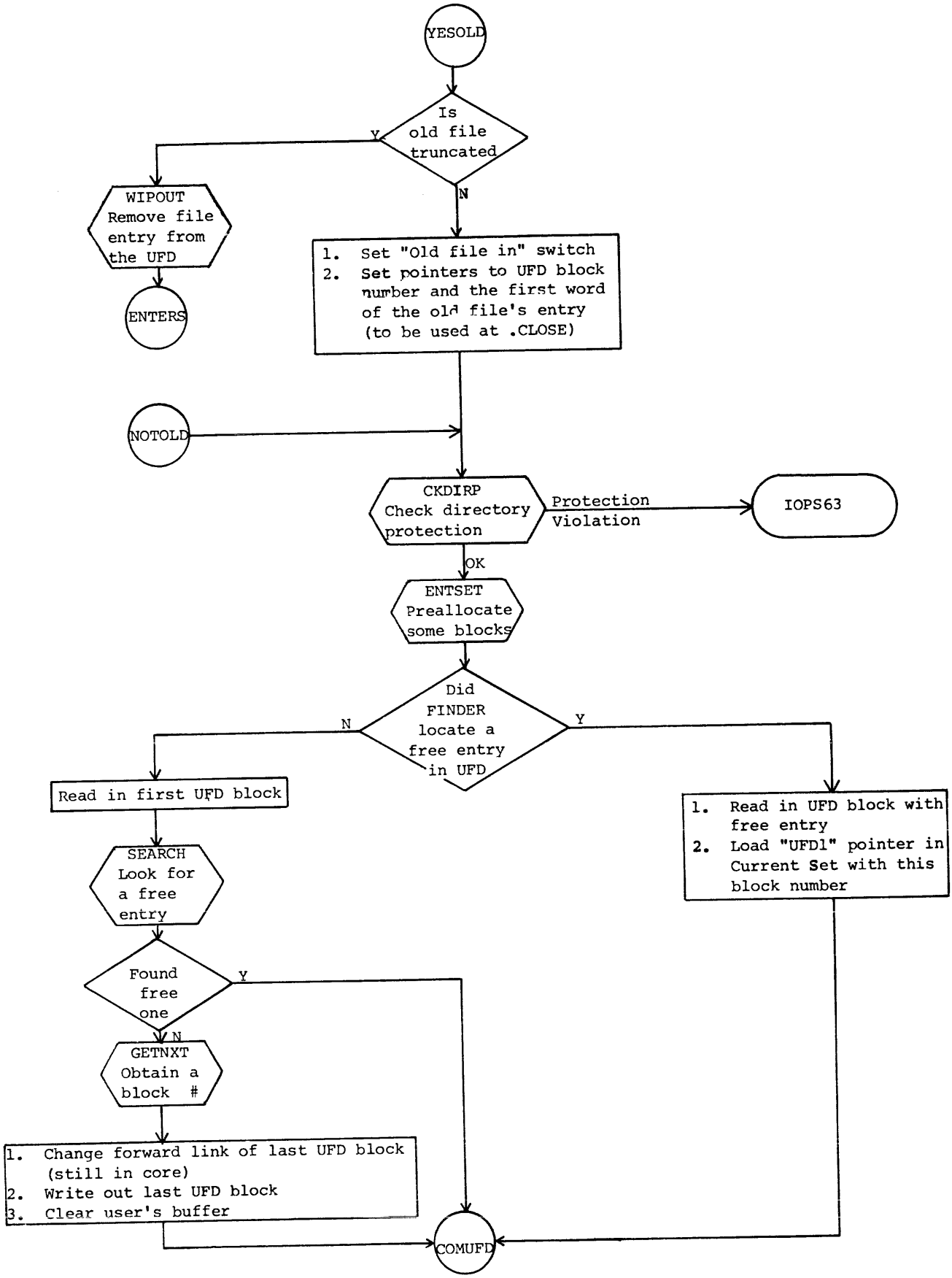

E
(.RTRAN)
.RAND executed
N
IOPS 11
Y
0≤ relative block number ≤ number of file blocks
N
IOPS 66
Y
Calculate RIB block number, and the desired pointer's position within that RIB block
Proper RIB block in core
Y
N
Depending on the location of the desired block, relative to the RIB block in core, read in the next or preceding RIB block
Proper RIB block now in core
N
Y
Save pointer to desired data block
RF or RP/RK
RF
RP/RK
Store starting word number and the number of data words desired
Assume transfer starting at word zero, through word 375, and set parameters accordingly
Word parameters imply transfer of link
Y
IOPS 67
N
Input
N
MORET1
Y
Set direction switch
Use word count given in CAL during disk pack input
SETRET

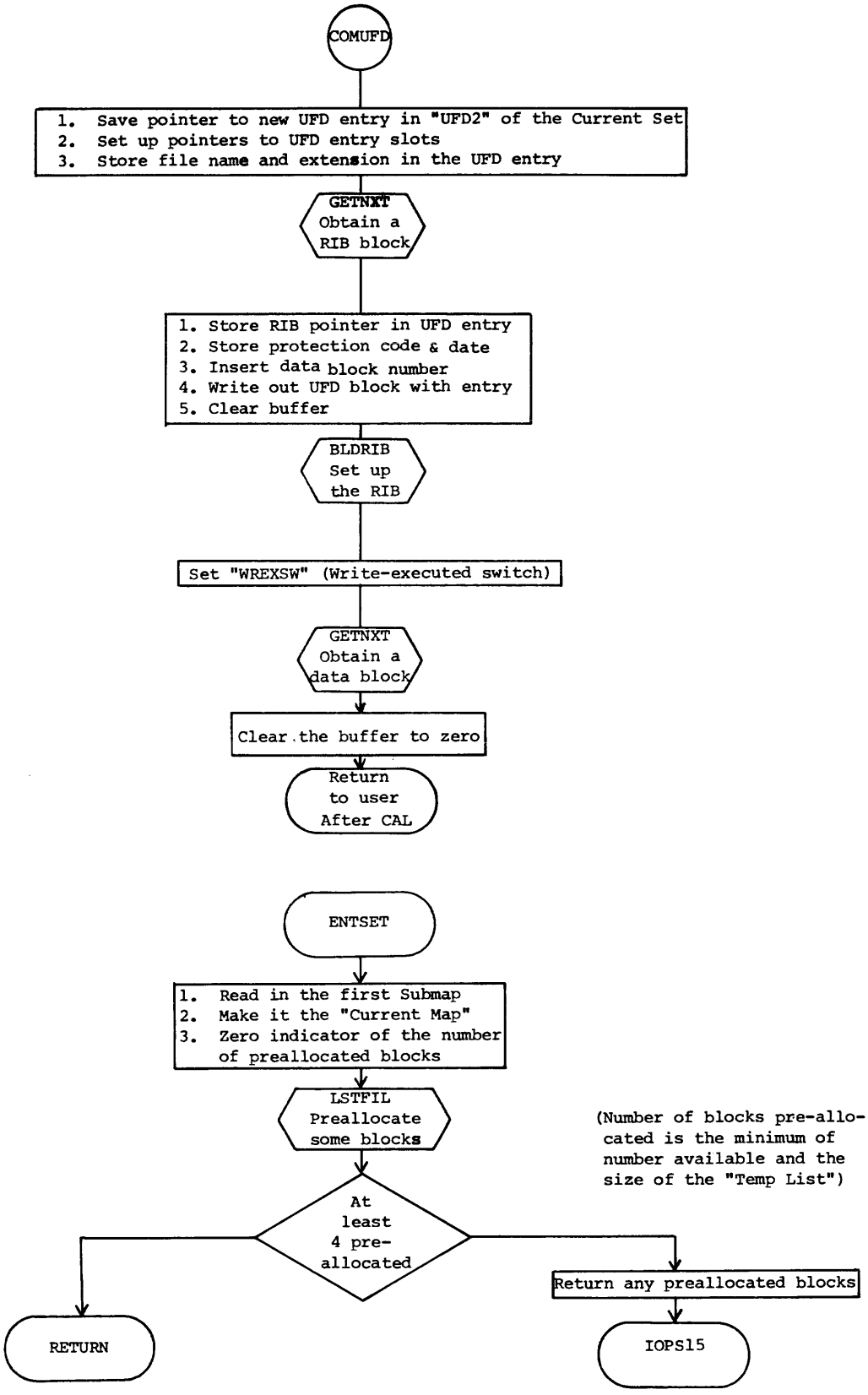

MORET1
Set up user's buffer to receive the link words for block
File one block long
Y
Set both link words to -1
SETRET
N
Current Rib block contain both link pointers
N - no forward link
N - no backward link
Y
Is this the last block
Y
N
Set links in data block from adjacent pointers in the RIB block
Is this the first block
N
Y
Set backward link to adjacent pointer in RIB, and forward link to -1
1. Set backward link to adjacent pointer in RIB block
2. Read in next RIB block and adjust RIB number indicator in the current set
3. Set forward link to first pointer in the new RIB block
1. Set forward link to adjacent pointer in RIB
2. Read in previous RIB block and adjust RIB number indicator in the current set
3. Set backward link to last pointer in the new RIB block
Set forward link to adjacent pointer in RIB, and backward link to -1
SETRET
RF or RP/RK
RP/RK
RF
Set up driver with the correct block number
Set up driver with the correct word number
Bring in or send out required block or data
Exit to LOC+5

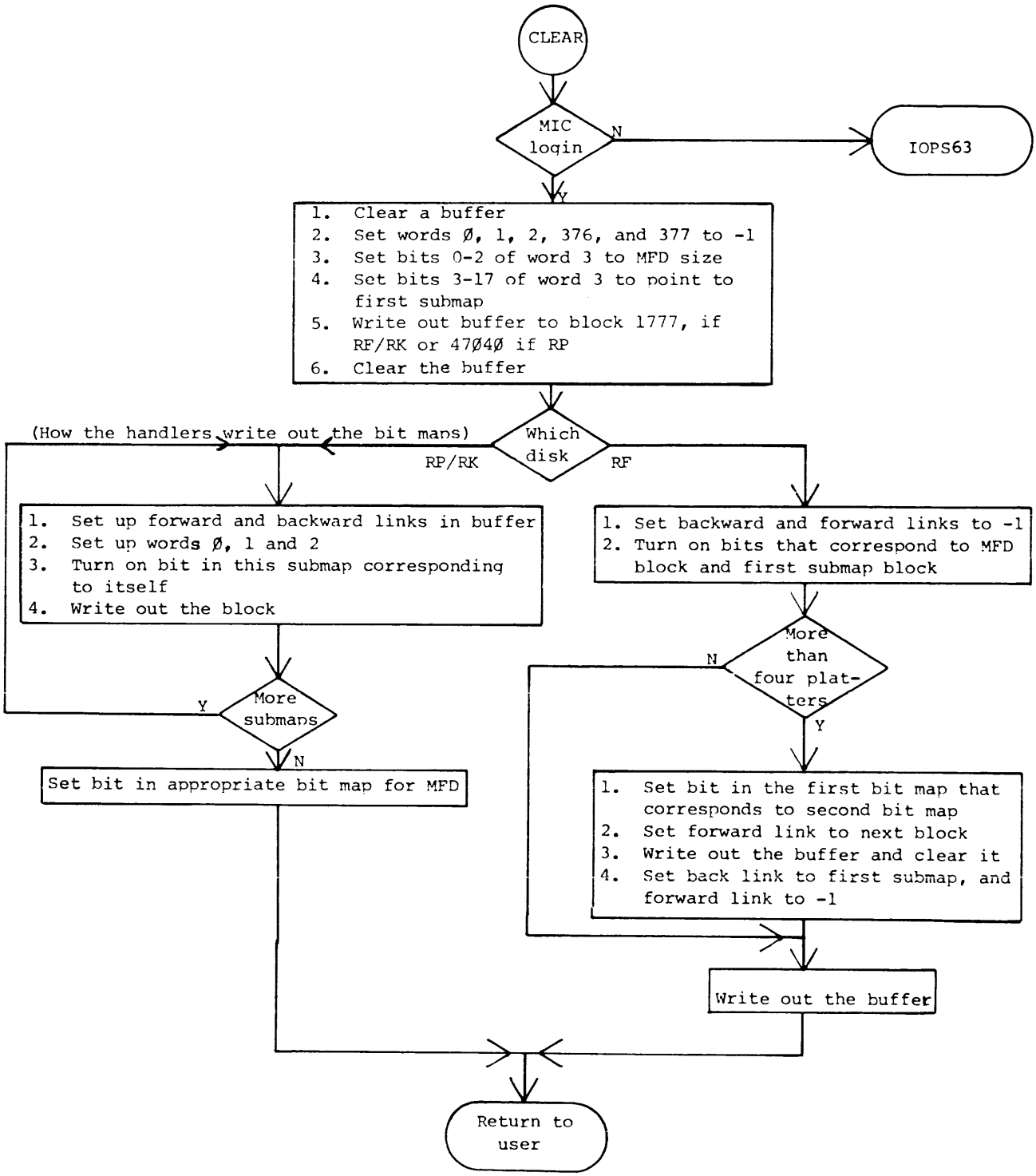

SEEK
Any unclosed file on this .DAT slot
Y
IOPS 10
N
Was this file successfully .FSTATed
Y
N
FINDER obtain file information
ERROR
IOPS 13, 51 or 71
Found File
File truncated
Y
IOPS 10
N
MIC logged in
Y
N
Logged in UIC = file's UIC
Y
N
UFD protected
N
Y
File protected
Y
IOPS 64
N
Read in first block of file
Exit to LOC+3

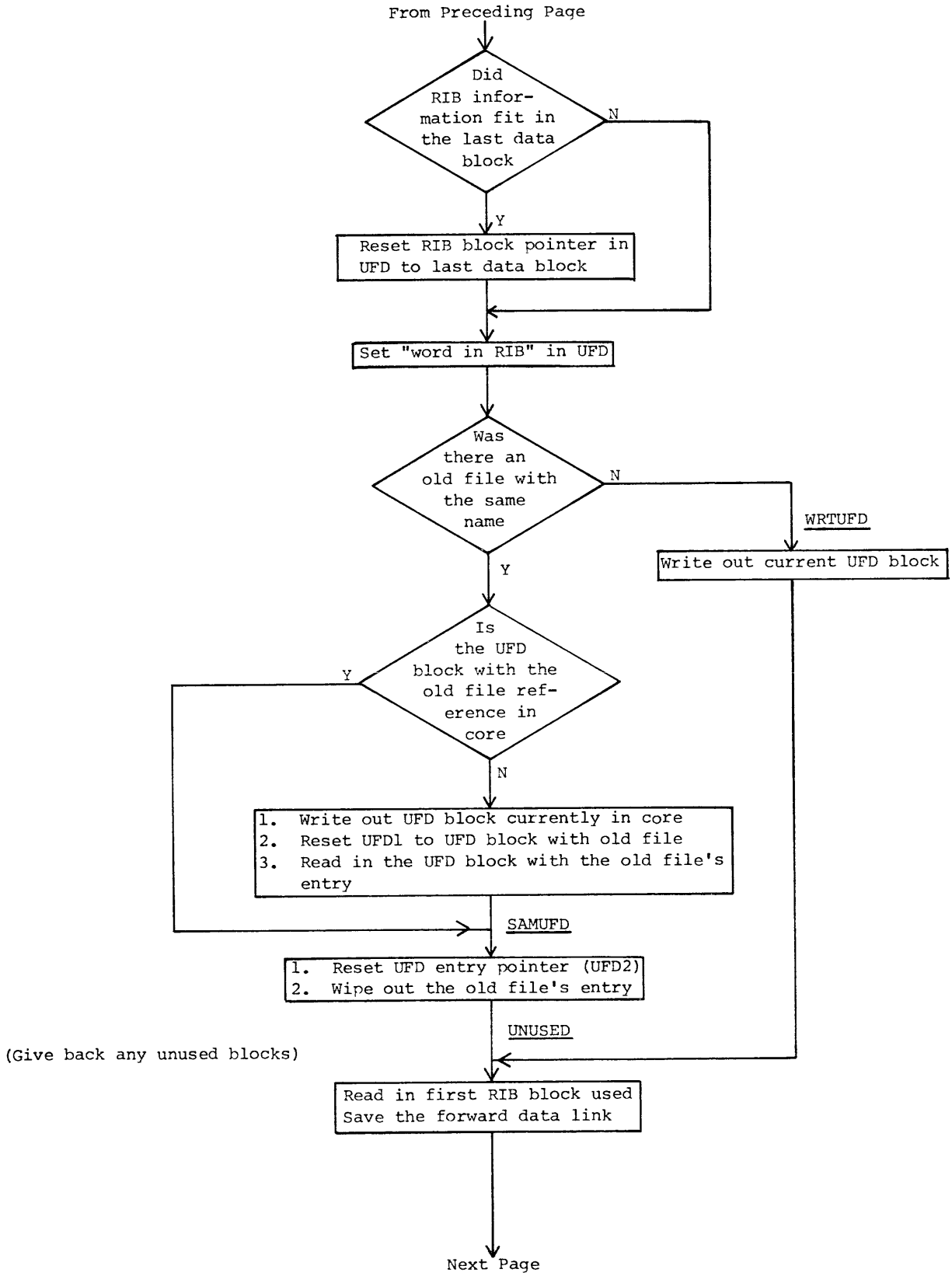

ENTER
Has this slot been opened and not closed*
Y
IOPS10
ENTERS
N
FINDER
Search MFD and UFD
START = Pointer to SAT word 3
(No entry in MFD for this UIC)
(Entry in MFD, but UFD is empty)
(UFD exists, but does not contain a file by the given name)
(UFD exists, and contains a file by the given name)
NOMFD
UICEMP
NOTOLD
YESOLD
1. Save number of the last MFD block read
2. Save file name
ENTSET
Preallocate some blocks
Read in last MFD block
Did FINDER find an empty MFD slot
Y
N
Set pointer to free slot found by FINDER
Is block with free entry in core
N
Read it in
GETNXT
Get next block number
1. Make forward link of last MFD block point to the next block
2. Write out the block
3. Clear the buffer
1. Set up entry pointers
2. Insert new UIC, entry size and zero protection code (unprotected) into new MFD entry
MFDSET
*That is, has a .RAND, .SEEK or .ENTER been issued without a .CLOSE?

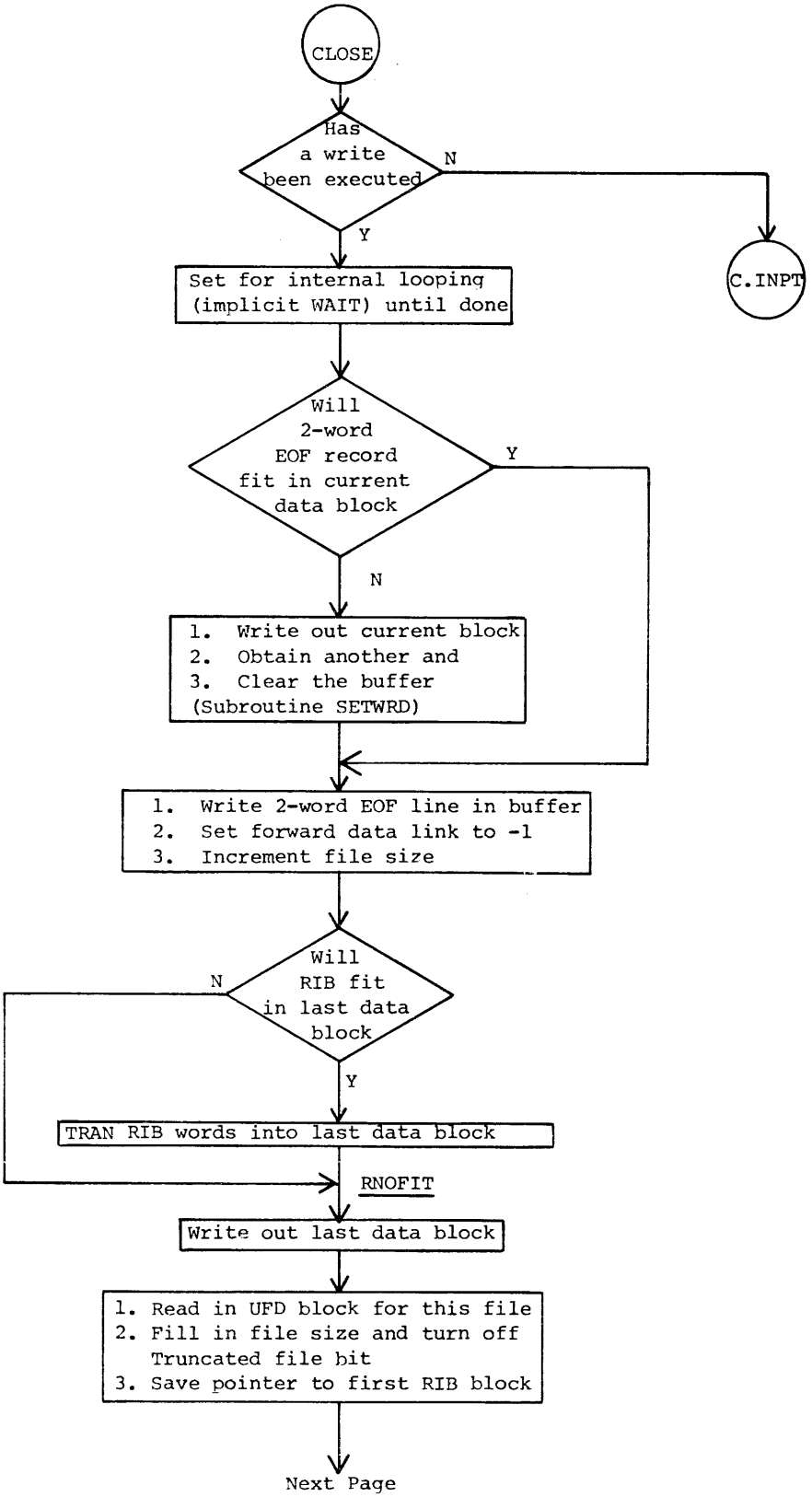

UICEMP
CKDIRP
Is directory protected?
Y
IOPS63
N
ENTSET
Preallocate some blocks
Read in the MFD block which contains proper entry
MFDSET
GETNXT
Obtain a block for the UFD
Set up entries in the Busy Table, the MFD block and the Current Set
Write out the MFD block and clear the buffer
1. Set up a new UFD block in the buffer, with a back link of -1
2. Set pointer to indicate location to receive the new entry
COMUFD

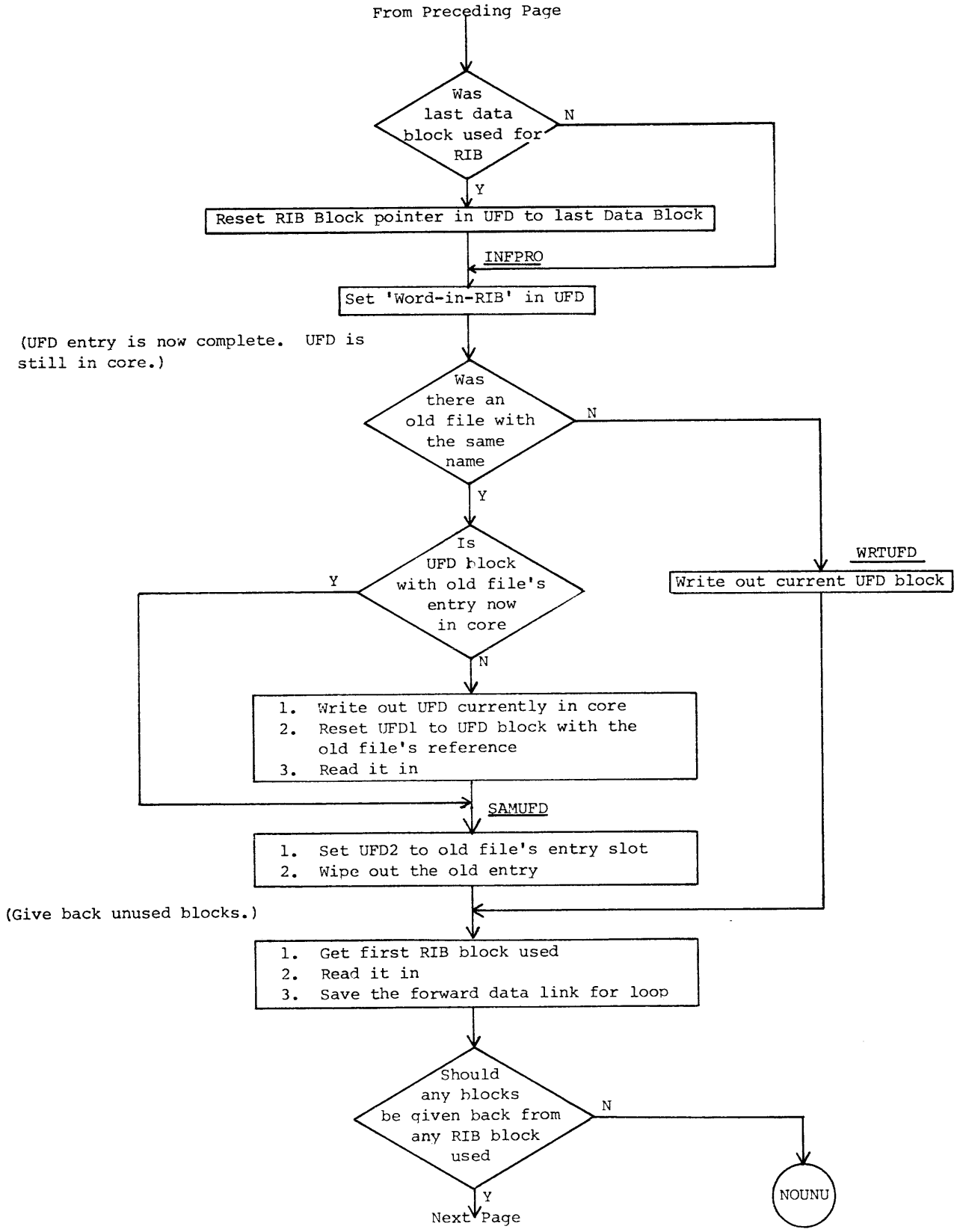

YESOLD
Is old file truncated
Y
N
WIPOUT
Remove file entry from the UFD
ENTERS
1. Set "Old file in" switch
2. Set pointers to UFD block number and the first word of the old file's entry (to be used at .CLOSE)
NOTOLD
CKDIRP
Check directory protection
Protection Violation
IOPS63
OK
ENTSET
Preallocate some blocks
Did FINDER locate a free entry in UFD
N
Y
Read in first UFD block
1. Read in UFD block with free entry
2. Load "UFD1" pointer in Current Set with this block number
SEARCH
Look for a free entry
Found free one
Y
N
GETNXT
Obtain a block #
1. Change forward link of last UFD block (still in core)
2. Write out last UFD block
3. Clear user's buffer
COMUFD

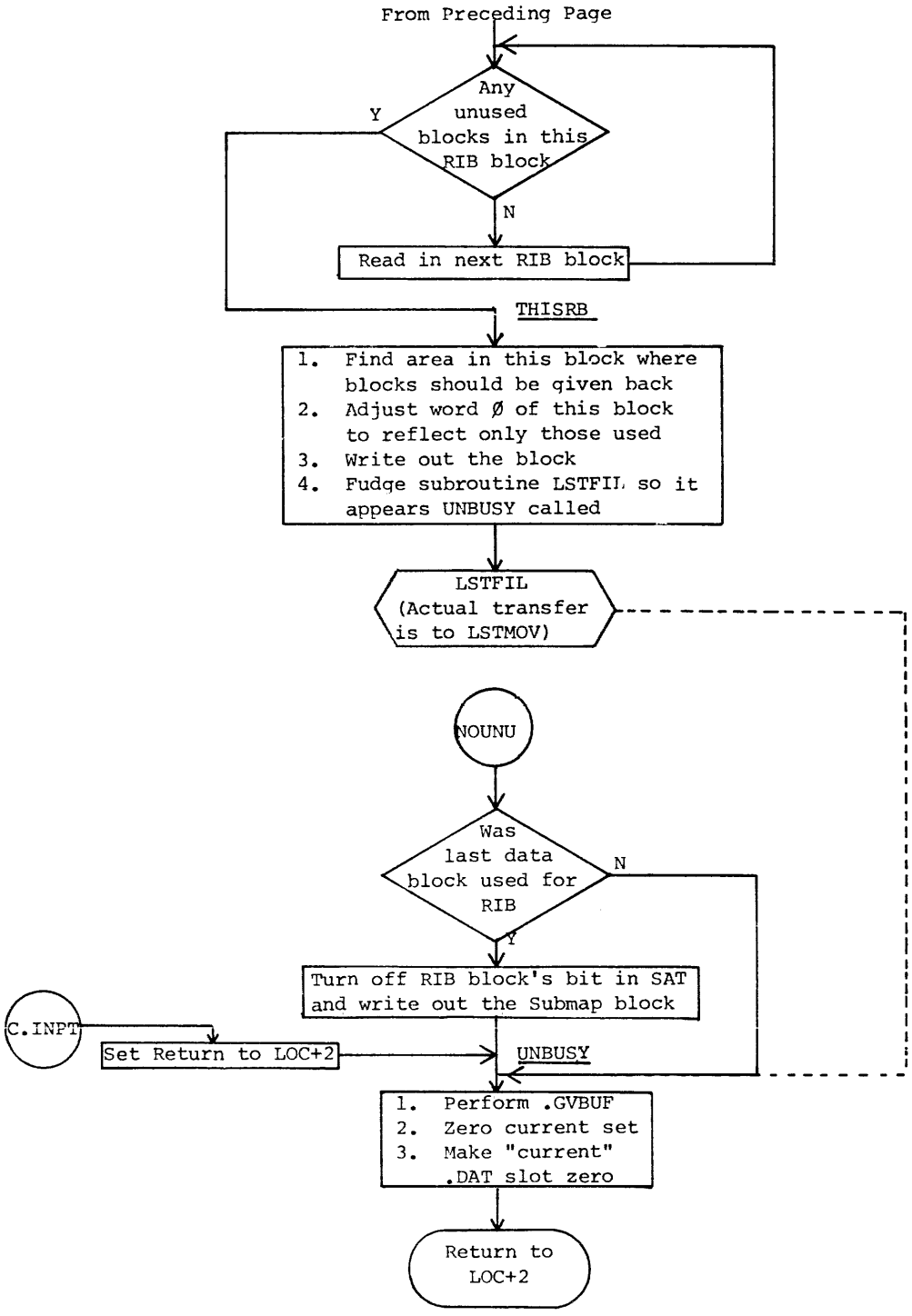

COMUFD
1. Save pointer to new UFD entry in "UFD2" of the Current Set
2. Set up pointers to UFD entry slots
3. Store file name and extension in the UFD entry
GETNXT
Obtain a
RIB block
1. Store RIB pointer in UFD entry
2. Store protection code & date
3. Insert data block number
4. Write out UFD block with entry
5. Clear buffer
BLDRIB
Set up
the RIB
Set "WREXSW" (Write-executed switch)
GETNXT
Obtain a
data block
Clear the buffer to zero
Return
to user
After CAL
ENTSET
1. Read in the first Submap
2. Make it the "Current Map"
3. Zero indicator of the number
of preallocated blocks
LSTFIL
Preallocate
some blocks
(Number of blocks pre-allo-
cated is the minimum of
number available and the
size of the "Temp List")
At
least
4 pre-
allocated
Return any preallocated blocks
RETURN
IOPS15

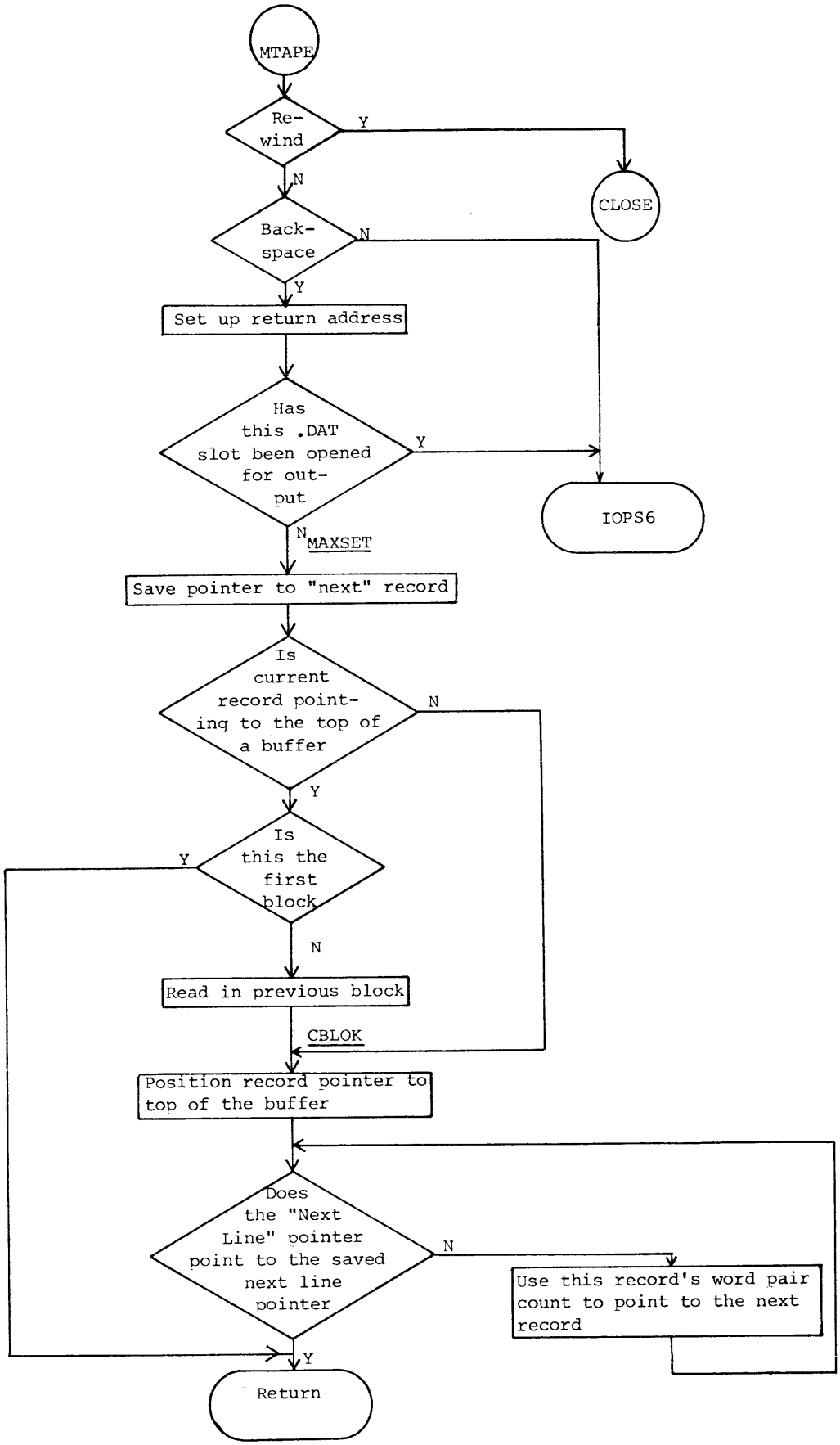

CLEAR
MIC login
N
IOPS63
Y
1. Clear a buffer
2. Set words Ø, 1, 2, 376, and 377 to -1
3. Set bits 0-2 of word 3 to MFD size
4. Set bits 3-17 of word 3 to point to first submap
5. Write out buffer to block 1777, if RF/RK or 47Ø4Ø if RP
6. Clear the buffer
(How the handlers write out the bit maps)
Which disk
RP/RK
RF
1. Set up forward and backward links in buffer
2. Set up words Ø, 1 and 2
3. Turn on bit in this submap corresponding to itself
4. Write out the block
1. Set backward and forward links to -1
2. Turn on bits that correspond to MFD block and first submap block
More than four platters
N
Y
More submaps
Y
N
Set bit in appropriate bit map for MFD
1. Set bit in the first bit map that corresponds to second bit map
2. Set forward link to next block
3. Write out the buffer and clear it
4. Set back link to first submap, and forward link to -1
Write out the buffer
Return to user

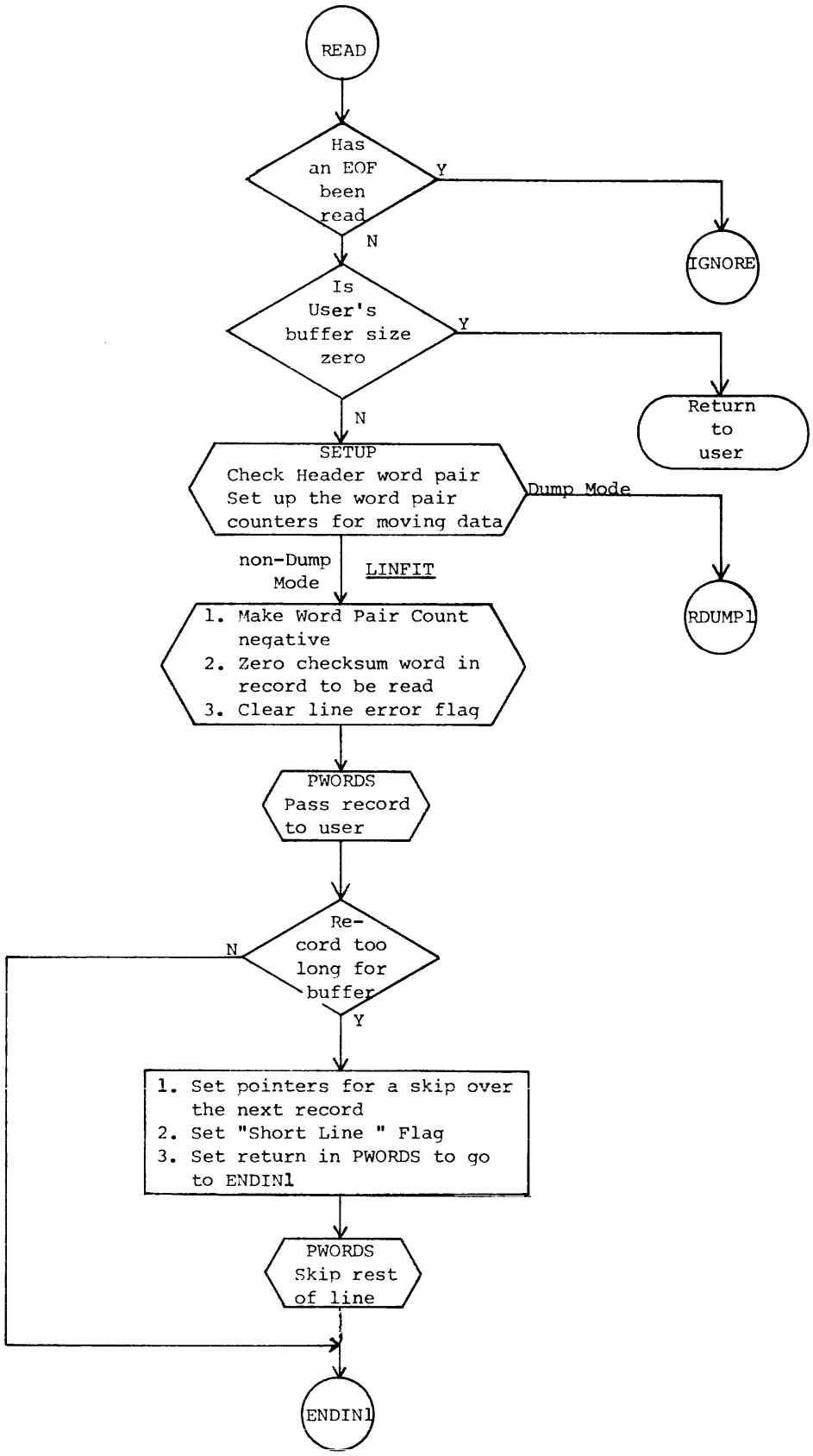

From Preceding Page
Did RIB information fit in the last data block
N
Y
Reset RIB block pointer in UFD to last data block
Set "word in RIB" in UFD
Was there an old file with the same name
N
WRTUFD
Write out current UFD block
Y
Is the UFD block with the old file reference in core
Y
N
1. Write out UFD block currently in core
2. Reset UFDl to UFD block with old file
3. Read in the UFD block with the old file's entry
SAMUFD
1. Reset UFD entry pointer (UFD2)
2. Wipe out the old file's entry
UNUSED
(Give back any unused blocks)
Read in first RIB block used
Save the forward data link
Next Page

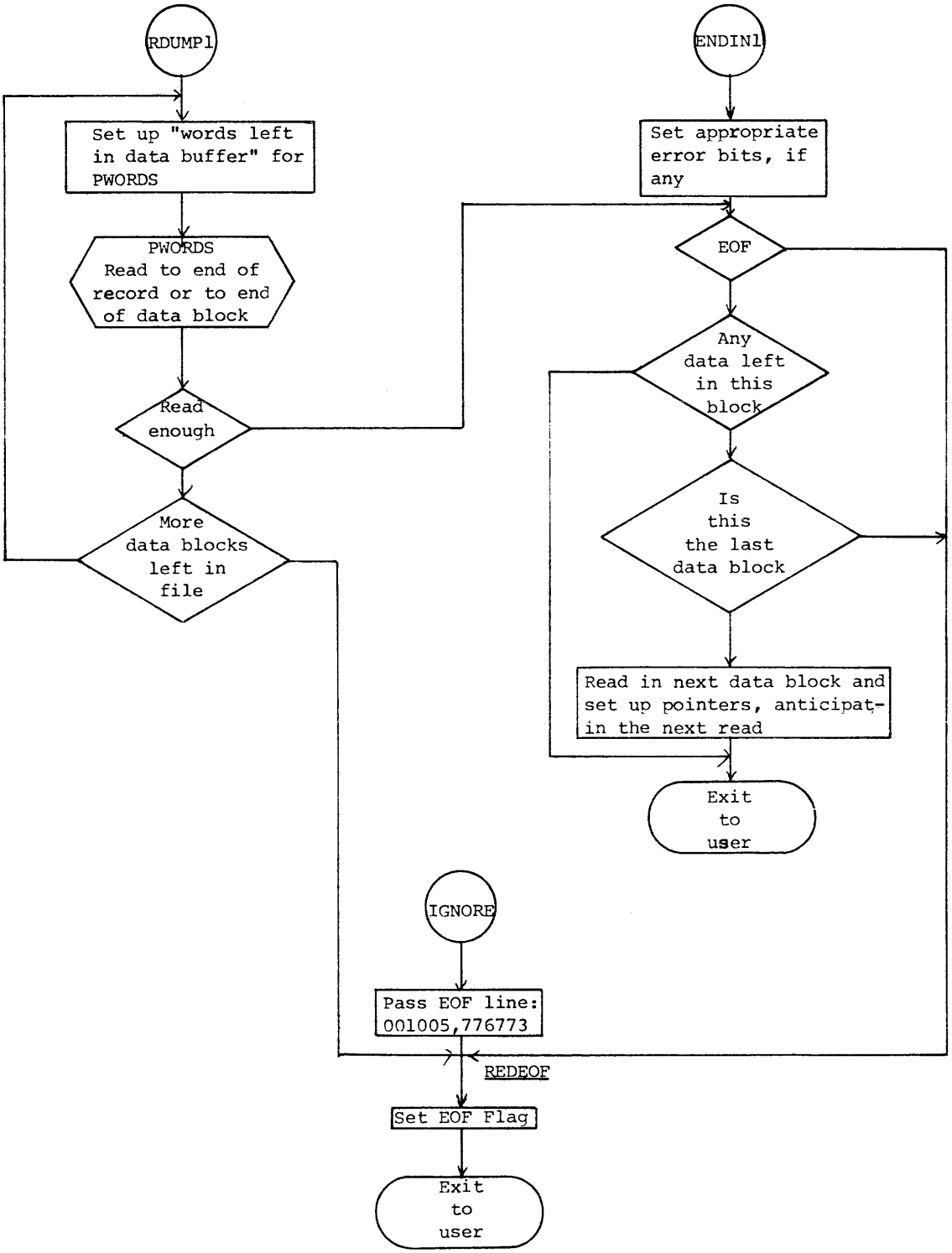

CLOSE
Has a write been executed
N
Y
C.INPT
Set for internal looping (implicit WAIT) until done
Will 2-word EOF record fit in current data block
Y
N
1. Write out current block
2. Obtain another and
3. Clear the buffer
(Subroutine SETWRD)
1. Write 2-word EOF line in buffer
2. Set forward data link to -1
3. Increment file size
Will RIB fit in last data block
N
Y
TRAN RIB words into last data block
RNOFIT
Write out last data block
1. Read in UFD block for this file
2. Fill in file size and turn off Truncated file bit
3. Save pointer to first RIB block
Next Page

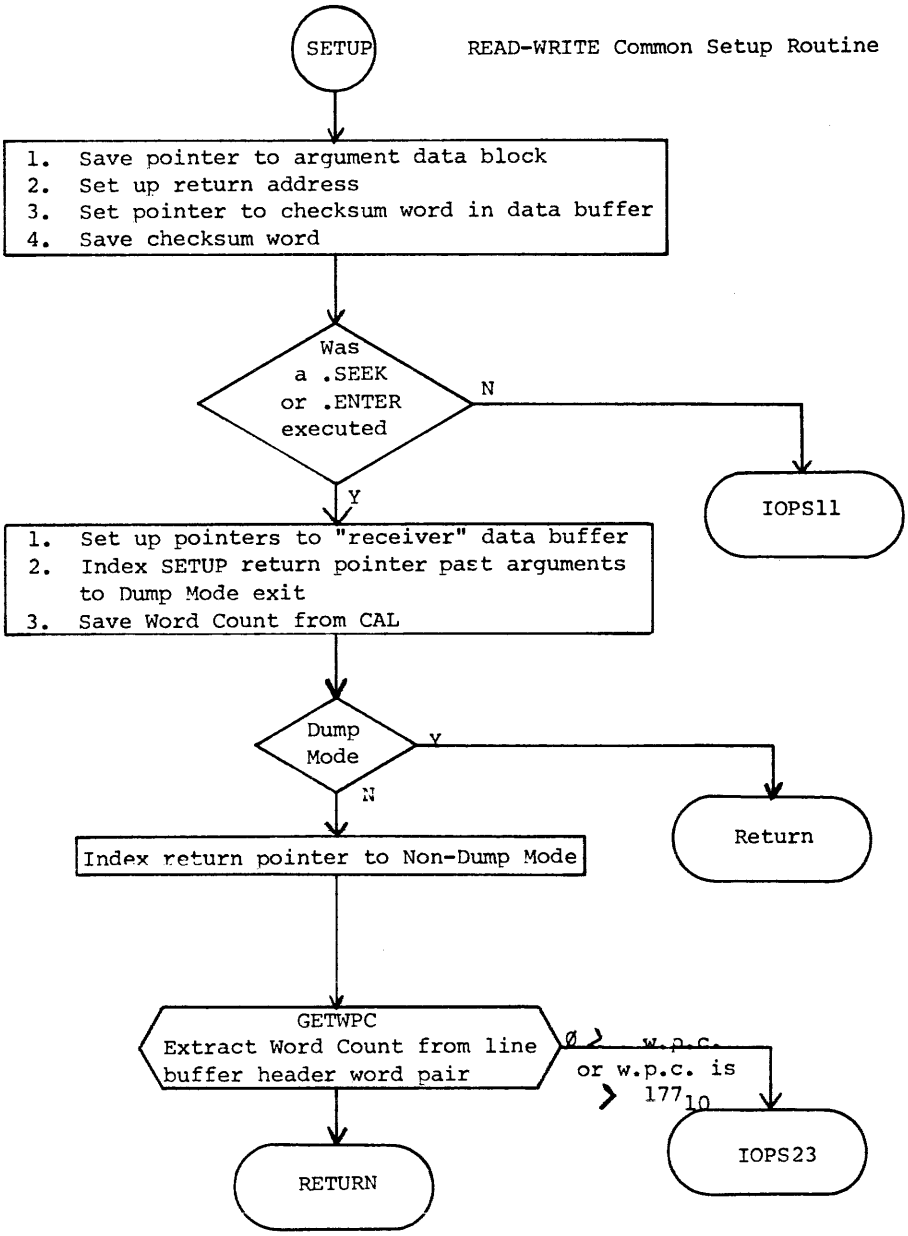

From Preceding Page
Was last data block used for RIB
N
Y
Reset RIB Block pointer in UFD to last Data Block
INFPRO
Set 'Word-in-RIB' in UFD
(UFD entry is now complete. UFD is still in core.)
Was there an old file with the same name
N
Y
Is UFD block with old file's entry now in core
Y
N
WRTUFD
Write out current UFD block
1. Write out UFD currently in core
2. Reset UFD1 to UFD block with the old file's reference
3. Read it in
SAMUFD
1. Set UFD2 to old file's entry slot
2. Wipe out the old entry
(Give back unused blocks.)
1. Get first RIB block used
2. Read it in
3. Save the forward data link for loop
Should any blocks be given back from any RIB block used
N
Y
NOUNU
Next Page

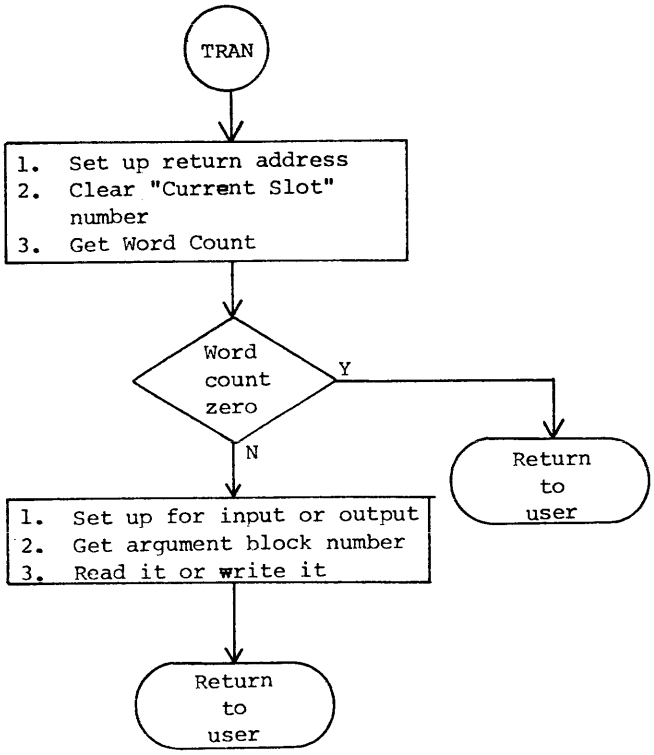

From Preceding Page
Any unused blocks in this RIB block
Y
N
Read in next RIB block
THISRB
1. Find area in this block where blocks should be given back
2. Adjust word Ø of this block to reflect only those used
3. Write out the block
4. Fudge subroutine LSTFIL so it appears UNBUSY called
LSTFIL (Actual transfer is to LSTMOV)
NOUNU
Was last data block used for RIB
N
Y
Turn off RIB block's bit in SAT and write out the Submap block
C.INPT
Set Return to LOC+2
UNBUSY
1. Perform .GVBUF
2. Zero current set
3. Make "current" .DAT slot zero
Return to LOC+2

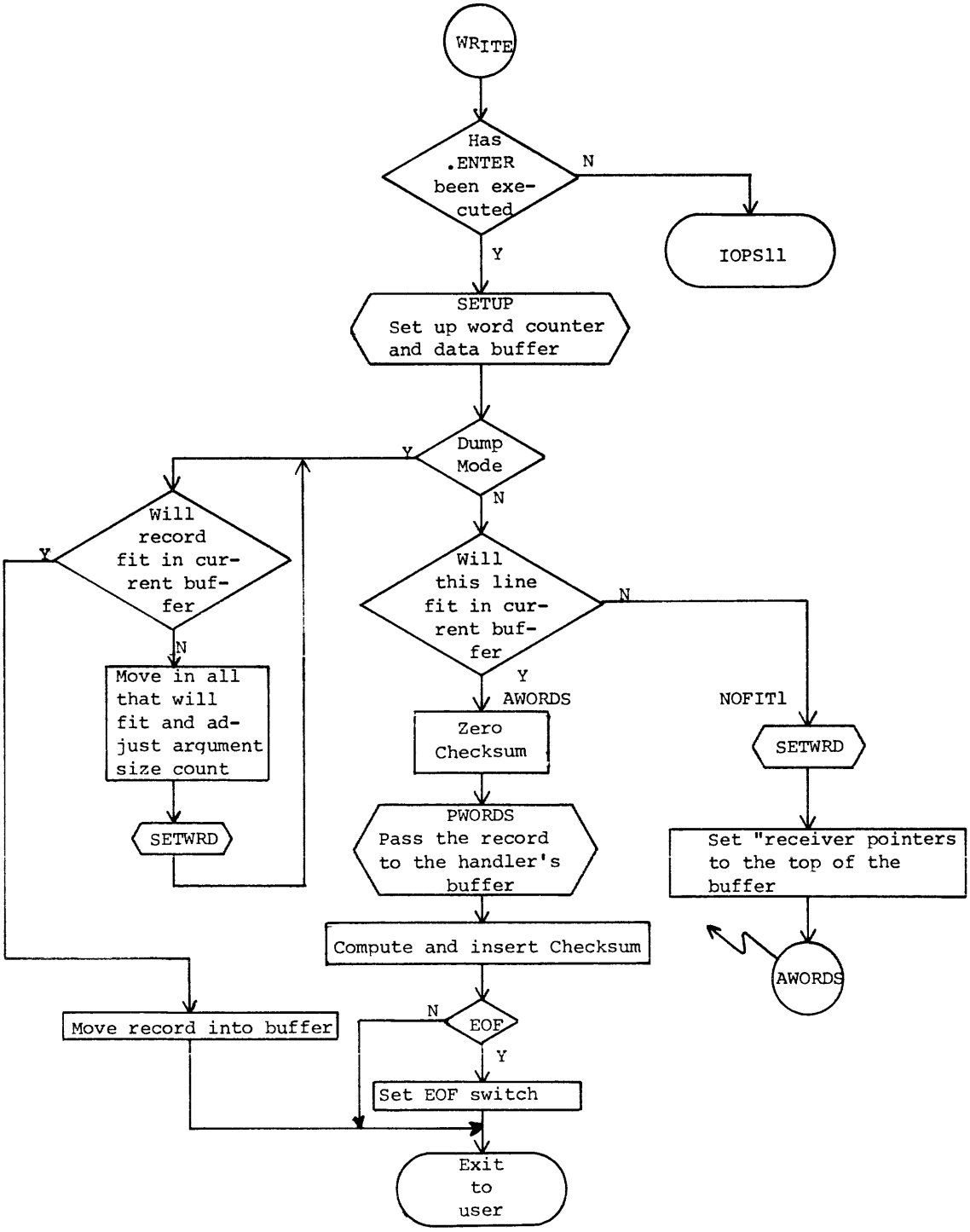

MTAPE
Re-wind
Y
N
CLOSE
Back-space
N
Y
Set up return address
Has this .DAT slot been opened for out-put
Y
IOPS6
N
MAXSET
Save pointer to "next" record
Is current record point-ing to the top of a buffer
N
Y
Is this the first block
Y
N
Read in previous block
CBLOK
Position record pointer to top of the buffer
Does the "Next Line" pointer point to the saved next line pointer
N
Use this record's word pair count to point to the next record
Y
Return

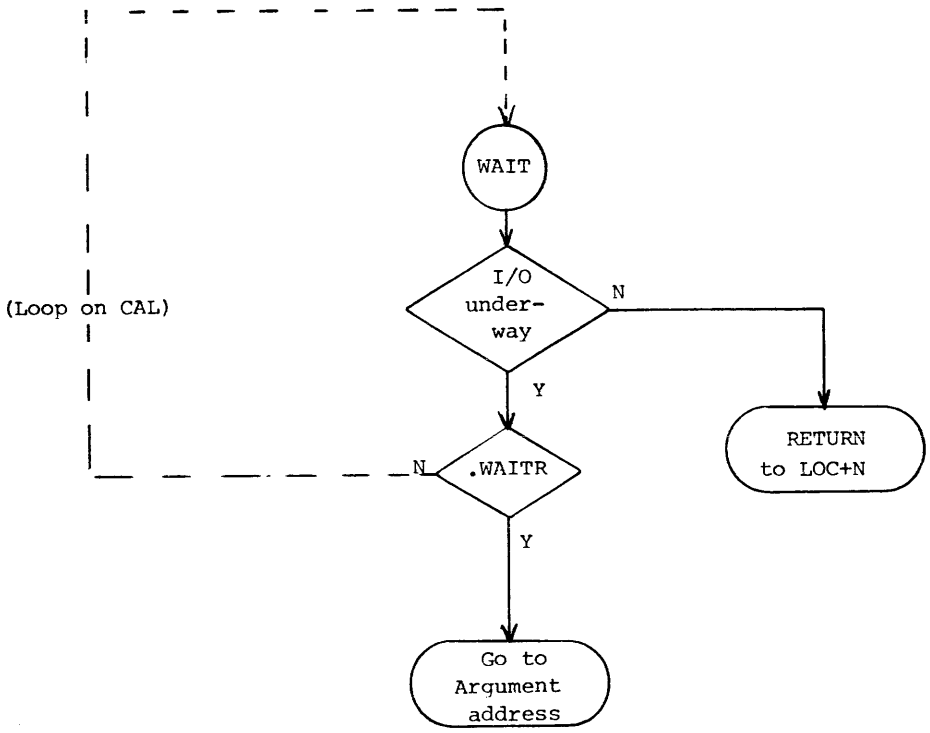

READ
Has an EOF been read
Y
N
IGNORE
Is User's buffer size zero
Y
N
Return to user
SETUP
Check Header word pair
Set up the word pair counters for moving data
Dump Mode
RDUMP1
non-Dump Mode
LINFIT
1. Make Word Pair Count negative
2. Zero checksum word in record to be read
3. Clear line error flag
PWORDS
Pass record to user
Re-cord too long for buffer
N
Y
1. Set pointers for a skip over the next record
2. Set "Short Line " Flag
3. Set return in PWORDS to go to ENDIN1
PWORDS
Skip rest of line
ENDIN1

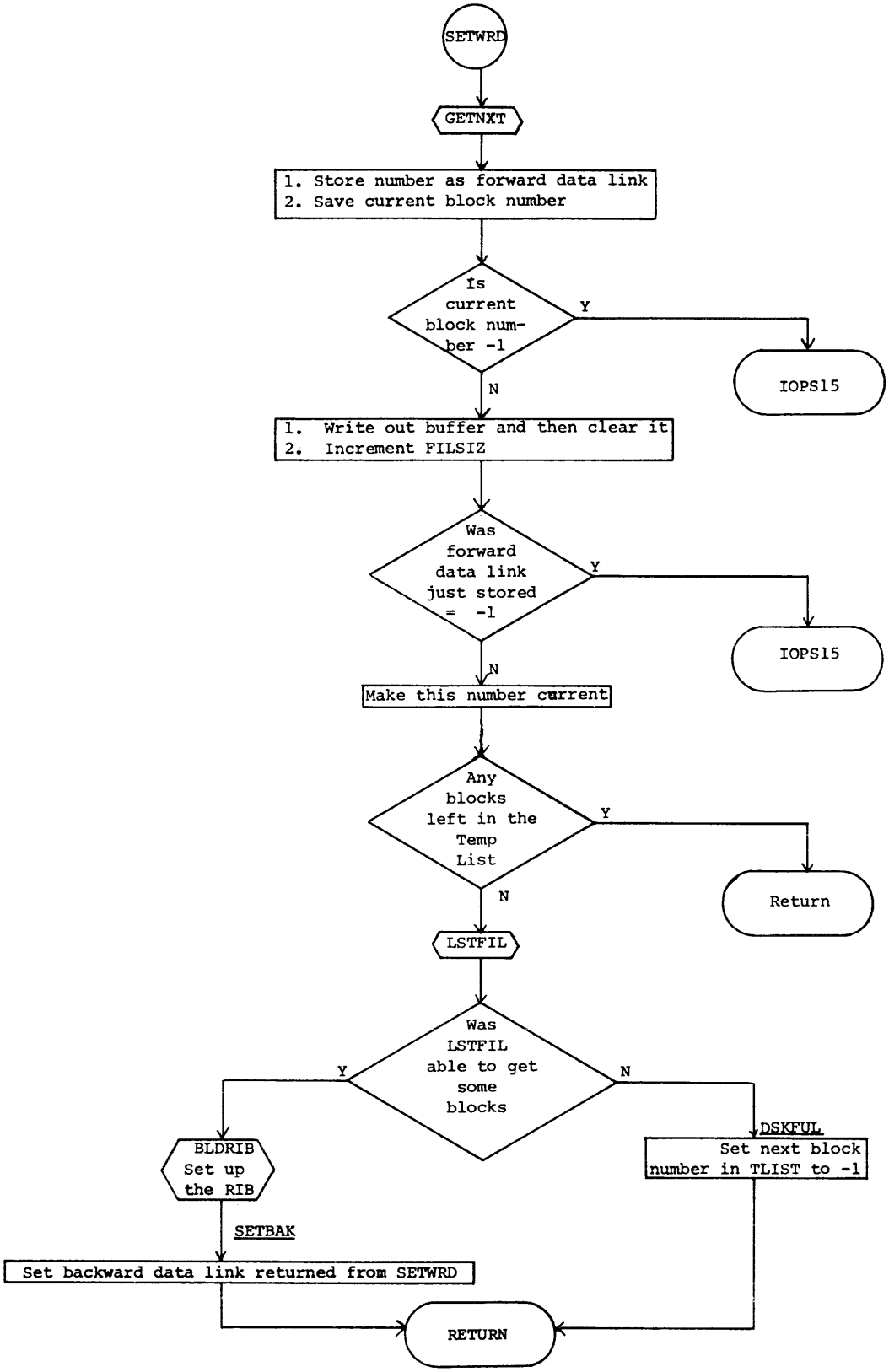

RDUMP1
Set up "words left in data buffer" for PWORDS
PWORDS
Read to end of record or to end of data block
Read enough
More data blocks left in file
ENDIN1
Set appropriate error bits, if any
EOF
Any data left in this block
Is this the last data block
Read in next data block and set up pointers, anticipat-in the next read
Exit to user
IGNORE
Pass EOF line: 001005,776773
REDEOF
Set EOF Flag
Exit to user

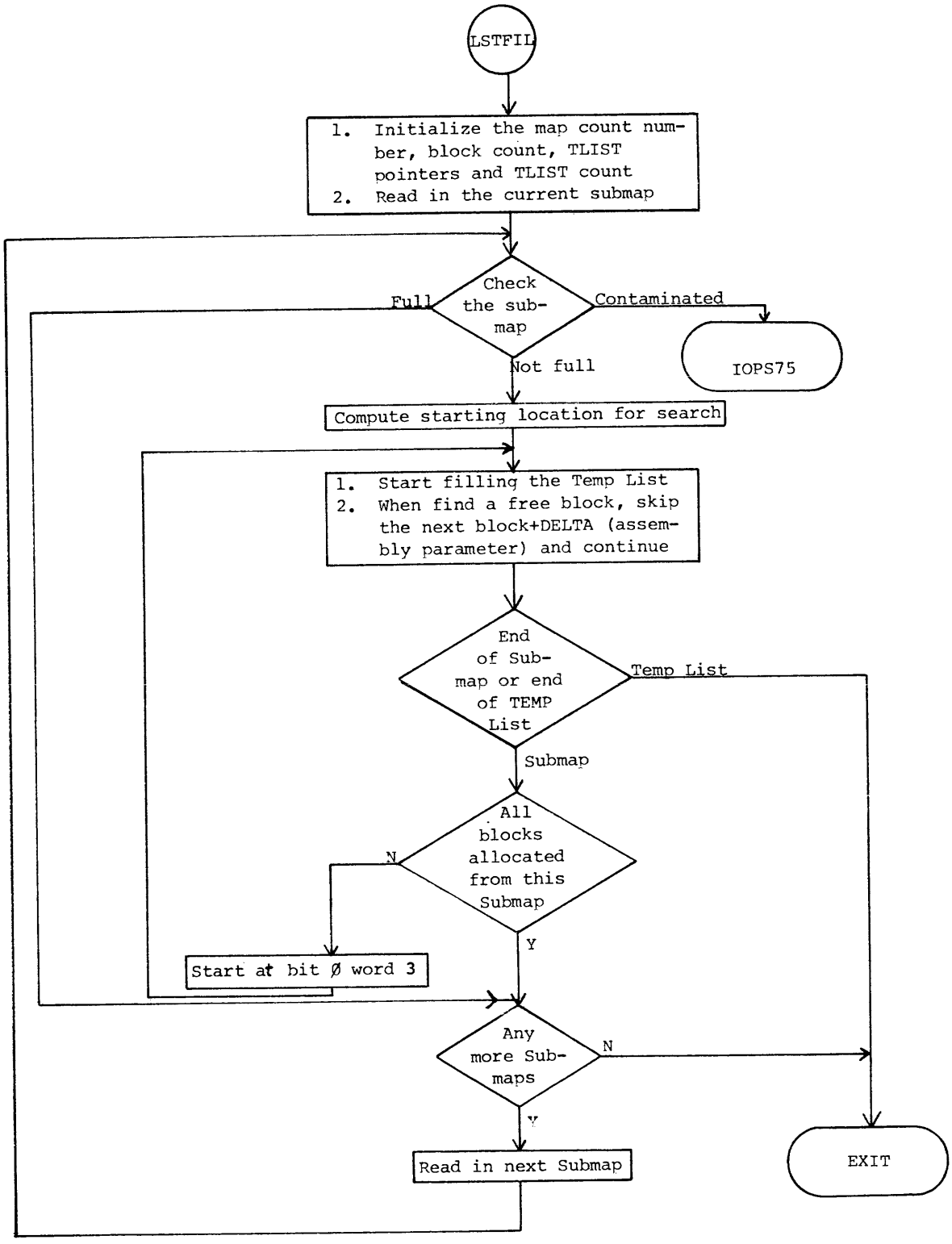

SETUP
READ-WRITE Common Setup Routine
1. Save pointer to argument data block
2. Set up return address
3. Set pointer to checksum word in data buffer
4. Save checksum word
Was a .SEEK or .ENTER executed
N
Y
IOPS11
1. Set up pointers to "receiver" data buffer
2. Index SETUP return pointer past arguments to Dump Mode exit
3. Save Word Count from CAL
Dump Mode
Y
N
Return
Index return pointer to Non-Dump Mode
GETWPC
Extract Word Count from line buffer header word pair
Ø
w.p.c. or w.p.c. is > 177₁₀
IOPS23
RETURN

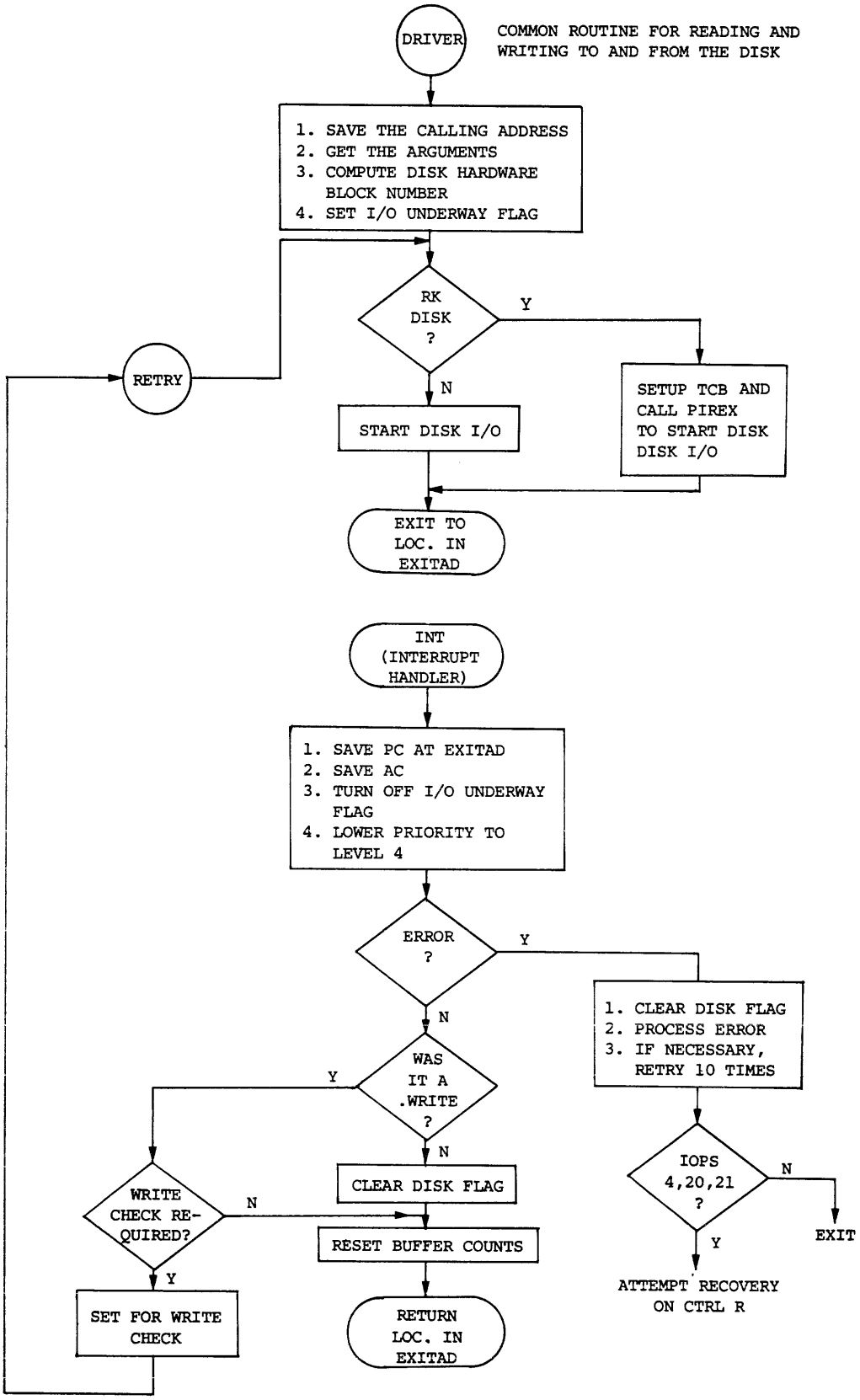

TRAN
1. Set up return address
2. Clear "Current Slot" number
3. Get Word Count
Word count zero
Y
N
Return to user
1. Set up for input or output
2. Get argument block number
3. Read it or write it
Return to user


WRITE
Has .ENTER been executed
N
IOPS11
Y
SETUP
Set up word counter and data buffer
Dump Mode
Y
N
Will record fit in current buffer
Y
N
Move in all that will fit and adjust arqument size count
SETWRD
Will this line fit in current buffer
N
Y
AWORDS
Zero Checksum
PWORDS
Pass the record to the handler's buffer
NOFIT1
SETWRD
Set "receiver pointers to the top of the buffer
AWORDS
Compute and insert Checksum
Move record into buffer
N
EOF
Y
Set EOF switch
Exit to user


(Loop on CAL)
WAIT
I/O under-way
N
Y
RETURN to LOC+N
N
.WAITR
Y
Go to Argument address


SETWRD
GETNXT
1. Store number as forward data link
2. Save current block number
Is current block number -1
Y
IOPS15
N
1. Write out buffer and then clear it
2. Increment FILSIZ
Was forward data link just stored = -1
Y
IOPS15
N
Make this number current
Any blocks left in the Temp List
Y
Return
N
LSTFIL
Was LSTFIL able to get some blocks
Y
N
BLDRIB
Set up the RIB
DSKFUL
Set next block number in TLIST to -1
SETBAK
Set backward data link returned from SETWRD
RETURN


LSTFIL
1. Initialize the map count number, block count, TLIST pointers and TLIST count
2. Read in the current submap
Check the submap
Full
Contaminated
IOPS75
Not full
Compute starting location for search
1. Start filling the Temp List
2. When find a free block, skip the next block+DELTA (assembly parameter) and continue
End of Submap or end of TEMP List
Temp List
Submap
All blocks allocated from this Submap
N
Y
Start at bit Ø word 3
Any more Submaps
N
Y
Read in next Submap
EXIT



DISK "A" HANDLERS

APPENDIX C
PROCEDURE FILE

```
ASG

1 ASSIGN DEVICE UIC TO .DAT
A @D00(@D11()@)@ <@U00(@D12()@)@> @A00()@


ASM

2 MACRO AND LINE EDITOR
A @D00(@D11()@)@ <@U00(@D12()@)@> -14/@D03(@D11()@)@ <@U03(@D12()@)@> -15
B.PRE
@A00(FILTMP)@
@A03(@A00(FILTMP)@)@
A @D00(@D11()@)@ <@U00(@D12()@)@> -11/@D01(@D11()@)@ <@U01(@D12()@)@> -10
A @D02(@D11()@)@ <@U02(@D12()@)@> -14/@D05(@D11()@)@ <@U05(@D12()@)@> -13
A @D04(LP)@ <@U04(@D12()@)@> -12
MACRO
@0(BL)@_@A00(FILTMP)@@D14()@

BNK

14 BANK MODE OPERATION-ON
BANK ON

BUF

1 NUMBER OF BUFFERS
BUFFS @A00(4)@

CHN

1 SPECIFY 7 OR 9 TRACK MAGTAPE
C @A00()@

CMP

1 SOURCE COMPARE
A @D00(@D11()@)@ <@U00(@D12()@)@> -15/@D01(@D11()@)@ <@U01(@D12()@)@> -14
SRCCOM
@0()@_@A00()@/@A01()@@D14()@

DIR

1 LIST DIRECTORY
PIP
L LP_@A00(@D11()@)@ <@U00(@D12()@)@>@D14()@
```

```
DLG
2 LOGOUT UIC
LOGOUT

DMP
01 DUMP UTILITY - EXPANDED SUB FILE
A @D00(@D11()@)@ <@U00(@D12()@)@> -14/@D01(LP)@ <@U01(@D12()@)@> -12
DUMP
@A00(ALL)@@D14()@

DOS
1,3, GENERAL PRC FILE FOR GIVING COMMAND STRINGS
@A00(@D14()@)@

FIL
2 CREATE A FILE FROM CARDS/EDITOR
A @D00(@D11()@)@ <@U00(@D12()@)@> -14
A @D01(@D11()@)@ <@U01(@D12()@)@> -15
B.PRE
@A00(FILTMP)@
@A01(@A00(FILTMP)@)@

FOR
2 FORTRAN IV AND LINE EDITOR
A @D00(@D11()@)@ <@U00(@D12()@)@> -14/@D01(@D11()@)@ <@U01(@D12()@)@> -15
B.PRE
@A00(FILTMP)@
@A01(@A00(FILTMP)@)@
A @D00(@D11()@)@ <@U00(@D12()@)@> -11/@D03(@D11()@)@ <@U03(@D12()@)@> -13
A @D02(LP)@ <@U02(@D12()@)@> -12
F4
@O(BL)@_@A00(FILTMP)@@D14()@

JOB
2 START NEW JOB
LOG JOB @A00()@ BEGIN @D14()@
T
LOGIN @A02(SCR)@
A NON 2,3,4,7,10,11,12,13,14,15,16,17,20/@D11()@ 1
PIP
N @D11()@ <SCR>@D14()@
@A03()@
KEEP @A04(OFF)@
TIMEST @A01(1)@:00
```

```
KEP

1 RETAIN DEVICE ASSIGNMENTS
KEEP @A00(ON)@


LCM

13 SUPPLEMENT TO LIB PRC-UPDATE .LIBR
@A00(CLOSE@D13()@)@ @A01(@D13()@)@ @A02()@


LIB

1
A @D00(@D11()@)@ <@U00(@D12()@)@> -14
A @D01(@D00(@D11()@)@)@ <@U01(@U00(@D12()@)@)@> -15
A @D02(@D11()@)@ <@U02(@D12()@)@> -10
A @D03(LP)@ <@U03(@D12()@)@> -12
UPDATE
@O(LUS)@_@A00(.LIBR)@@D14()@


LNK

13 DIRECT SUB FILE - BUILDS LINKS FOR EXECUTE FILE-USE WITH OVL PRC
@A00(@D14()@)@@D14()@


LOG

2 LOGIN UIC
LOGIN @A00(SCR)@


LST

2 LIST CONTENTS OF FILE ON LINE PRINTER
PIP
T LP_@D00(@D11()@)@ <@U00(@D12()@)@> @A00(FILTMP)@ (A)@D14()@


MAC

2 MAC-ELEVEN AND LINE EDITOR
A @D00(@D11()@)@ <@U00(@D12()@)@> -14/@D01(@D11()@)@ <@U01(@D12()@)@> -15
B.PRE
@A00(FILTMP)@
@A01(@A00(FILTMP)@)@
A @D00(@D11()@)@ <@U00(@D12()@)@> -11
A @D02(LP)@ <@U02(@D12()@)@> -12
MAC11
@O(BL)@_@A00(FILTMP)@@D14()@
```

```
MAP

1;3 DIRECT SUB FILE FOR CHAIN OPTION AND RES CODE ONLY
CHAIN
@A00(TMPXCT)@@D14()@
@A01(SZ)@@D14()@
@A02(FILTMP)@@D14()@
@D14()@

MIC

2 LOGIN MIC UIC
MICLOG @A00()@

MNT

1 MOUNT TAPE# ON DRIVE #
LOGW MOUNT @O(D)@-TAPE# @A00()@ ON DRIVE# @A01()@ - WRITE @A02(LOCK)@

MSG

13 MESSAGE TO OPERATOR-DIRECT SUB FILE
LOG @A00()@

MSW

13 MESSAGE TO OPERATOR W/WAIT-DIRECT SUB
LOGW @A00()@

NDR

1 CREATE NEW DIRECTORY
PIP
N @A00(@D11()@)@ <@U00(@D12()@)@>@D14()@

OVL

13 DIRECT SUB FILE - USE FOR BUILDING OVERLAYS(CHAIN)
CHAIN
@A00(TMPXCT)@@D14()@
@A01(SZ)@@D14()@
@A02(FILTMP)@@D14()@

PAG

1 PAGE MODE OPERATION-ON
PAGE ON
```

PRT

```
1 SPECIFY PROTECTION CODE
P @A00(2)@
```

RUN

```
02 RUN
F4
BL_@A00()@ @D14()@
CHAIN
@A00()@@D14()@
NM@D14()@
@A00()@@D14()@
@D14()@
E @A00()@
```

QDP

```
1 DUMP CORE ON TERMINAL ERRORS-NO ARGUMENTS
QDUMP
```

XCT

```
2 EXECUTE
A @D00(@D11()@)@ <@U00(@D12()@)@> -4
E @A00(TMPXCT)@
```

XVM

```
1 XVM MODE OPERATION ON/OFF
XVM @A00(ON)@
```

INDEX

Please cut along this line.

READER'S COMMENTS

NOTE: This form is for document comments only. Problems with software should be reported on a Software Problem Report (SPR) form.

Did you find errors in this manual? If so, specify by page.

_____

_____

_____

_____

_____

_____

Did you find this manual understandable, usable, and well-organized? Please make suggestions for improvement.

_____

_____

_____

_____

_____

_____

Is there sufficient documentation on associated system programs required for use of the software described in this manual? If not, what material is missing and where should it be placed?

_____

_____

_____

_____

_____

_____

Please indicate the type of user/reader that you most nearly represent.

- ☐ Assembly language programmer
- ☐ Higher-level language programmer
- ☐ Occasional programmer (experienced)
- ☐ User with little programming experience
- ☐ Student programmer
- ☐ Non-programmer interested in computer concepts and capabilities

Name _____ Date _____

Organization _____

Street _____

City _____ State _____ Zip Code or Country _____

If you require a written reply, please check here. ☐

---------- Fold Here ----------

---------- Do Not Tear - Fold Here and Staple ----------

FIRST CLASS
PERMIT NO. 33
MAYNARD, MASS.

BUSINESS REPLY MAIL
NO POSTAGE STAMP NECESSARY IF MAILED IN THE UNITED STATES

Postage will be paid by:

digital

Software Communications
P. O. Box F
Maynard, Massachusetts 01754

digital

digital equipment corporation

PRINTED IN U.S.A.