ISIS: Anatomy of a Real-World Operating System

by Perry C. Hutchison
Computer Science Department
Iowa State University
Ames, IA 50010

Dear Jim: *Received: 77 Dec 23*

I am enclosing a sort of "internal logic manual" for Intel's ISIS system. This all started some months ago when I became the victim of a persistent "data CRC" error on the MDS which could not be reproduced on anything else. Several phone calls to Intel produced only the claim that, by the time the error message came out, the information as to what track and sector were involved was no longer available anywhere in the machine. This sounded rather preposterous, so I went a-hunting. About 2 weeks' work, spread over a considerably longer period, yielded the discoveries contained in this write-up. Since I don't suppose I am all that unique in needing to debug things in the context of ISIS, I decided to write it up for publication. It will probably be fairly interesting to 8080 (and Z-80) hackers, even if they don't have access to an MDS. The ISIS disk allocation and directory policies would make an excellent standard for floppy disk file systems.

This article may also be useful as context for Max Ogoston's RIMOS (DDJ 9/77).

Very truly yours,
Perry C. Hutchison

ISIS (Intel Systems Implementation Supervisor) is the floppy disk operating system developed by Intel Corporation for the Intellec MDS microcomputer system. ISIS was written entirely in PL/M, a dialect of PL/1 oriented to the 8080 microprocessor. The ISIS system is exceptionally well designed; it bears significant resemblance to Bell Labs' UNIX operating system for the PDP-11.

Intel has released neither the source code nor any internal documentation concerning ISIS. Since a knowledge of the internal workings of such systems is occasionally needed if they are to be utilized to the full extent of their capabilities, a study of the object code was undertaken. This document contains the principal findings of that study. It should be useful both as a reference for use by persons having occasion to deal with the internals of the ISIS system and as an educational example of what really goes on in a small single-user operating system. Names used are taken from the published ISIS documentation where appropriate.

The information contained herein has been derived from an examination of the object code of Version 1.2 of 32K ISIS, received in August, 1976. (The same disk also contains Version 1.1 of ASM80, 1.3 of EDIT, 1.1 of UPM, and 2.0 of ICE80.) Except where a more general applicability is specifically stated, this information should not be expected to apply to any other version of ISIS.

It is probable that, despite thorough checking, some errors will be found in this presentation. The author would appreciate being informed of them.

Environment

ISIS operates in the Intellec MDS microcomputer system, and uses the facilities of the MDS Monitor ROM (which occupies addresses F800H - FFFFH) to communicate with character-oriented peripheral devices (i.e. everything but the disk). Table I lists the entry points and functions of the principal Monitor routines. Parameters and returned values are handled as in PL/M – see the discussion below. (There are, of course, additional subroutines in the ROM, but an assembly listing of the ROM is included with the MDS and anyone interested in its internal workings can look it up. ISIS uses no Monitor entry points other than those listed in Table I.)

A detailed description of the MDS disk controller is contained in the Intel *MDS-DOS Hardware Reference Manual* and will be only briefly summarized here. The controller occupies I/O ports 78H - 7FH, and is controlled principally by a program-generated "I/O Parameter Block" (IOPB) in memory. The address of the IOPB is supplied to the controller by outputting its low and high bytes to output ports 79H and 7AH. The IOPB identifies the drive, the operation to be performed, the sector count, the track address, the starting sector address, and the starting memory address for reading or writing. The disk controller contains a DMA controller which is used for reading the IOPB as well as for transferring data.

Organization

ISIS consists of some 57 PL/M procedures, including 8 which merely provide access to the Monitor ROM. The remaining 49, plus two compiler-generated subroutines, are listed in Table II; Figures 1 and 2 depict the caller-called relationships. (Many of the routines of Fig. 1 call one or more routines of Fig. 2, but the spaghetti effect would render a combined drawing incomprehensible.) The heavy lines in Fig. 1 lead to routines which directly correspond to and implement the various ISIS system calls. (The procedures ERROR and XEQIOPB are actually Fig. 2 routines, but the direct calls to them from ISIS are shown in Fig. 1 for completeness.)

Table II should be largely self-explanatory, with the exception of a few abbreviations and notational conventions. The second column contains the Entry Point address of the procedure. The columns labeled "T" signify the Type of the associated quantity: A (Address) denotes a 16-bit value, B (Byte) an 8-bit value, and L (Logical) a Byte value used in a True-False sense. (An odd value is taken as True, an even value as False.) The column labeled "ADDR" is used for two related purposes: in the case of procedures which have parameters, it gives the memory address at which each parameter is stored; in the case of non-parametric procedures it gives the address at which the first of that procedure's local variables is stored. (Local variables of parametric procedures are stored immediately following the last parameter.) A dash in this column signifies that the procedure has no local variables and, except in the case of the compiler-generated subroutines, no parameters.

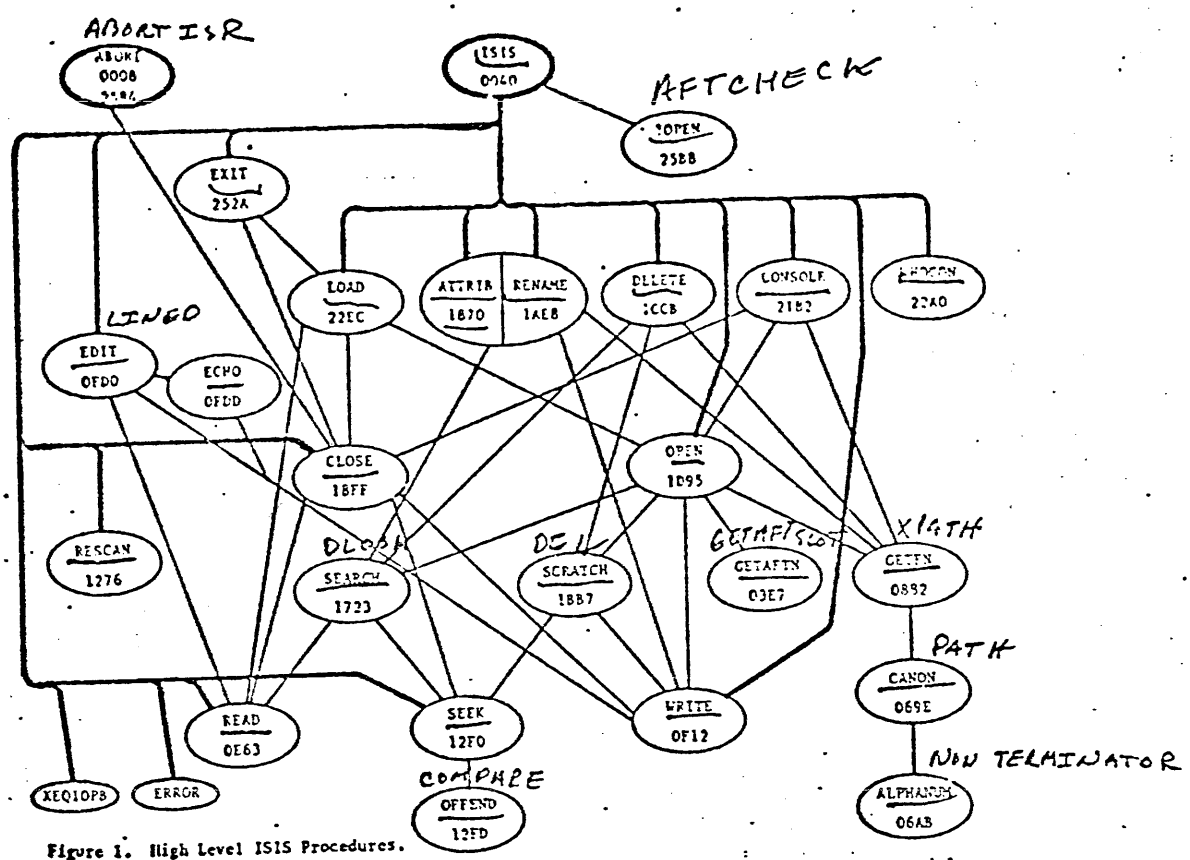


Figure 1. High Level ISIS Procedures.

The combining of ATTRIB and RENAME is a clutter-reducing notational convenience having no special significance. Both call the same three subroutines.

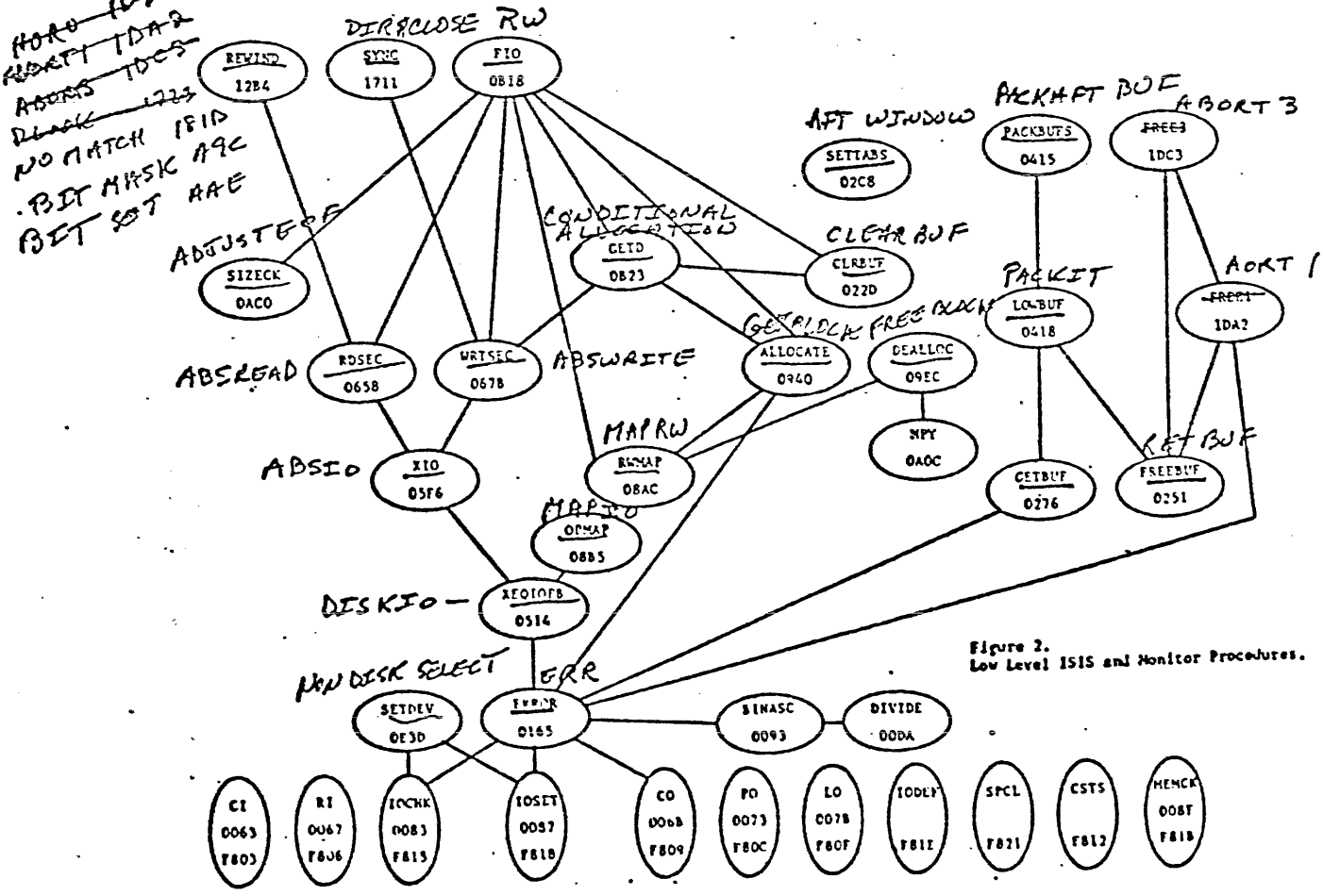


Figure 2.
Low Level ISIS and Monitor Procedures.

When one procedure is entirely contained within another, the table entry for the inner procedure immediately follows that for the outer one and the inner name is indented two spaces.

Data Structures

One of the principal motivations for this study has been the desire to obtain information of diagnostic usefulness. To that end, Table III presents a list of memory addresses whose contents may assist in diagnosing troubles. In addition, parameter values in active procedures may be useful. A traceback of procedure calls is fairly easily obtained from the stack, since entries other than return addresses are rare. The locations of the parameters may then be found from Table II.

The central in-memory data structure of ISIS is the File Structure, located in memory addresses 29E5 through 2B1B inclusive and summarized in Table IV. The File Structure consists of 18 tables, each containing 10 entries; each table has associated with it a "current pointer" which contains the address of the entry in that table which pertains to the current file (whose table index may, in turn, be found in 29E5). The first two entries (numbered 0 and 1) in each table always refer to the directories of disks 0 and 1; in a sense the directories are always open, although they do not always have buffers allocated. Entries 2 and 3 are the console files :CO: and :CI:. This leaves entries 4 through 9 for the six files which a user program may have open at any one time. Each open file corresponds to one entry in each table, but most of these entries are unused if the file is not on a disk.

The parameter AFT which many of the ISIS procedures require is simply a number in the range of 0 to 9 identifying a set of entries in the tables of the File Structure. This differs from the AFTN supplied by the user (and returned by OPEN) in that AFT = AFTN + 2.

The I.V. (Initial Value) shown in Table IV is the value which appears in the particular File Structure member for a disk file which has just been opened.

Disk Layout

The principles of the ISIS disk organization were rather well described by David Yulke in the December 1977 issue of *Kilobaud* (although he didn't mention ISIS by name), but will be included here for completeness and to provide additional details.

In ISIS, every disk sector falls into one of three groups: Data blocks, Linkage blocks, and Free blocks; this statement applies to system data as well as to user files. An ISIS file sonsists of Linkage blocks and Data blocks. Data blocks contain the bytes which compose the file, while Linkage blocks (Yulke calls them Map blocks) tell how the Data blocks are to be linked together. The first two bytes of a Linkage block contain the disk address of this file's preceding Linkage block (zero if this is the first). All disk addresses in ISIS are stored as a Block Number and a Track Number. Tracks are numbered from 0 to 76 and the blocks of each track are numbered from 1 to 26. The next two bytes contain the disk address of the following Linkage block. The remaining 124 bytes contain the disk addresses of up to 62 data blocks.

Every ISIS-format disk contains four system files, collectively referred to as the Format files. These may be opened for input and read just like any other file, but may never be written into (except by special system routines). Tables VI and VII list the Format files and describe their contents.

The system does not always bother to look the Format files up in the directory; it assumes that their locations are known. Thus the first Linkage block of the directory itself is always assumed to be Track 1 Block 1, and the Data blocks of the Allocation Bit Map are assumed to be Track 2 Blocks 2 and 3. The boot ROM expects to find the System Initialization Program starting at Track 0 Block 1 and occupying as much of Track 0 as may be needed; the System Initialization Program then assumes that the first Linkage block of the file ISIS.BIN (which contains the system proper) will be found at Track 2 Block 4.

Secret "XEQIOPB" System Call

The XEQIOPB system call is not described in the published ISIS documentation. Its "command" value is 68 decimal (44 hex); the "parameter block" must contain the following 3 words: 534BH, drive, .iopb (where drive is 0 or 1 and .iopb is the memory address of the IOPB to be executed). This is presumably intended to be used by programs like FORMAT which need to perform disk operations not needed by "normal" programs. The IOPB is to be set up as if it were to run on drive 0, regardless of which drive is specified; XEQIOPB takes care of inserting the drive identification in the proper places. Normal system handling of disk errors is provided. Note that this call bypasses all directory accessing, file mapping, and protection flags, and must therefore be used only with extreme caution. One possible use would be in a program which does a sector-for-sector copy of a disk. This would be much faster than FORMAT SA since the normal disk allocation mechanism involves considerable overhead. Such a program could even copy a non-ISIS disk, e.g., DEC RT11, CP/M, etc., if needed.

Similar calls exist in Version 1.6 of 16K ISIS (the 534BH entry in the parameter block is not examined, but the drive and the IOPB address are still taken from the second and third words) and in Version 2.2 of ISIS-II (the parameter block contains only the drive and the IOPB adress – the 534BH is omitted); however since this is an unpublished feature, any program using it must be considered system-dependent. The possibility exists that such programs will not work with some future version of ISIS.

User-Program Use of ISIS Procedures

A user program may occasionally need to perform a processing task similar or identical to that performed by one of the ISIS procedures. In such a case, it may be desirable to call the system routine rather than having to write and debug code to perform the same function. In order to do this, a few details of the implementation of PL/M must be understood.

Up to two parameters of a procedure are passed in via registers. (The first thing a parametric procedure does is to store the appropriate registers into the memory locations reserved for those parameters.) A procedure having one parameter will expect the parameter to be in the C register (if a Byte) or in the BC register pair (if an Address). If there are two or more parameters, the last will be in the E (or DE) and the next-to-last will be in the C (or BC). The CALLER must store all other parameters into the proper memory locations before the procedure is entered. (This sounds messy, and it is. Newer PL/M compilers pass extra parameters on the stack, which is much nicer.)

A procedure which returns a Byte value will return it in the A register; one which returns an Address will return the more-significant byte in the B register and the less-significant byte in the A. (Note: this does not apply to the compiler-generated MPY and DIVIDE routines – see Table II.) Except for such returned values, the contents of the registers upon return from a procedure cannot in general be depended upon.

The benefits of using a system routine must of course be weighed against the extreme system dependency which results: the routines will almost certainly be in different places in different versions of the system and there is no guarantee that, even after they have been found in a new version, their results will be the same as before.

Conclusion

Users of the ISIS operating system have heretofore been hampered by the lack of documentation on its internal operation. While such documentation is not necessary most of the time, the need for it occasionally becomes critical. This presentation should partially fill that need for Intel customers while also assisting hobbyists in understanding how a first-rate operating system and file handler is organized.

Table I. Principal MDS Monitor Subroutines

NAME	ROM E.P.	ISIS E.P.	PARAMETER T	RETURNS T	PURPOSE
CI	F803	0063		CHARIN B	Returns character read from Console input device.
CO	F809	0068	CHAROUT B		Outputs character to Console output device.
CSTS	F812	--		STATUS L	Returns TRUE if Console has a character ready.
IOCHK	F815	0083		IOBYTE B	Returns current value of I/O control byte.
IOCL?	F81E	--	PGMID ENTRY	B A	Defines extensible I/O entry point (see MDS Operator's Manual).
IOSET	F818	0087	IOBYTE B		Sets I/O control byte.
LO	F80F	007B	CHAROUT B		Outputs character to List device.
MEMCK	F811	008F		MEMSIZE A	Returns highest available memory address.
PO	F80C	0073	CHAROUT B		Outputs character to Punch device.
RI	F806	0067		CHARIN B	Returns character read from Read device.
SPCL	F821	--			Entry point reserved for future expansion.

CP/M USERS' GROUP

Dear Jim: *Received: 77 Dec 16*

Just a note to inform you that there exists a CP/M Users' Group which is active both in user software exchange and also in group purchasing of proprietary software. Although designated CP/M Users' Group, we also naturally welcome users of IMSAI DOS-A and M-DOS and of the Cromemco CDOS, and soon expect to see a TDL FDOS which is similarly compatible with CP/M in terms of program load point, DOS call convention and diskette allocation and directory format.

Kindest regards,
Anthony R. Gold

345 E 86 Street
New York, Ny 10028

NORTH STAR EXECUTIVE SOFTWARE

News Release *Received: 77 Sept 13*

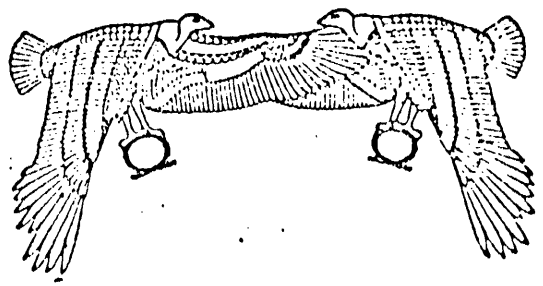
XEK, a complete system executive package for North Star users, is now available from the Byte Shop of Westminster.

The XEK package contains a disassembler capable of creating files that may be left in memory when changing from the disassembler to the executive package for re-assembly. The monitor software has the ability to accept input from cassette tapes and paper tape as either source or object files, as well as from the North Star diskette system. In addition, the assembler features a new auto-line editor for the creation of source files. This editor also extends to the modification of existing object files.

Another feature is the XEK's ability to handle up to six named files at once that may be consecutively assembled to form one object file. The assembler, monitor, and disassembler come with complete documentation, both on disk and as a manual. Total price, including first class postage, insurance and California resident's sales tax, is $48.00. For further information and ordering, contact: The Byte Shop of Westminster, 14300 Beach Boulevard, Westminster, CA 92683, (714) 894-9131.

Table II. Internal ISIS Procedures

NAME----	E.P.	PARAMETER	T	ADDR	RETURNS	T	PURPOSE----------	CALLS---
ABORT	2584			2DFF			Terminates execution and reboots system; usual entry via interrupt 1 vector at address 0008.	CLOSE XEQIOPB
ALLOCATE	0940	DRIVE	B	2CC2	TRKSEC	A	Allocates new disk block and returns its address.	ERROR RWMAP
ATTRIB	1870	.FILENAME	A	2D6A			Sets file attributes.	ERROR GETFN SEARCH SETTABS SYNC WRITE
BINASC	0093	NUMBER	A	2992			Converts binary NUMBER to Ascii representation in requested BASE starting at WHERE.	DIVIDE
		BASE	A	2994				
		WHERE	A	2996				
		NDIGITS	B	2998				
CANON	069E	.FILENAME	A	2D9A	ERRNUM	B	Converts FILENAME to internal form in INTFN.	ALPHANUM
		.INTFN	A	2D9C				
ALPHANUM	6AB			--	LETDIG	L	Returns TRUE if byte at (2D9A) is a letter or a digit.	
CLOSE	18FF	AFT	B	2D73			Closes specified file.	FREEBUF MPY READ REWIND RWMAP SEEK SETTABS SIZECK SYNC WRITE WRTSEC
CLRBUF	022D	.BUFFER	A	29DE			Clears 128-byte BUFFER to zeros.	
CONSOLE	2182	.INFILE	A	2DD0			Changes console file assignments.	CLOSE ERROR GETFN OPEN
		.OUTFILE	A	2DD2				
DEALLOC	09EC	DRIVE	B	2CC9			Releases disk block.	MPY RWMAP
		TRKSEC	A	2CCA				
DELETE	1CCB	.FILENAME	A	2D9E			Deletes a file given its name.	ERROR GETFN SCRATCH SEARCH SETTABS
DIVIDE	00DA	(BC)	A	--	(BC)	A	Compiler-generated division routine. Divides BC by DE; returns quotient in BC, remainder in DE.	
		(DE)	A	--	(DE)	A		
EDIT	0FD0	AFT	B	2D11			Called in place of READ when line-editing is required.	ECHO ERROR READ WRITE
		.BUFFER	A	2D12				
		COUNT	A	2D14				
		.ACTUAL	A	2D16				
ECHO	0FDD	CHAR	B	2D28			Writes CHAR on echo file.	SETTABS WRITE
ERROR	0165	FLAG	B	29A2			If FLAG = 0, returns CODE to user. If FLAG = 1, prints message. If FLAG = 2, prints message and aborts.	BINASC CO IOCHK IOSET
		CODE	B	29A3				
EXIT	2S2A			2DFE			Closes files and returns to system.	CLOSE LOAD RWMAP
FIO	0B18	AFT	B	2CCF			Transfers COUNT bytes between disk file buffer and caller's BUFFER; READFLAG is TRUE to read from file.	ALLOCATE CLRBUF GETD RDSEC RWMAP SIZECK WRTSEC
		.BUFFER	A	2CD0				
		COUNT	A	2CD2				
		.ACTUAL	A	2CD4				
		READFLAG	L	2CD6				
GETD	0D23			--			Allocates new data blocks as needed.	ALLOCATE CLRBUF WRTSEC
FREEBUF	0251	.BUFFER	A	29E2			Releases BUFFER to buffer pool.	
GETAFT	03E7			2B1E	AFT	B	Assigns AFT for file to be opened.	ERROR
GETBUF	0276			29E4	.BUFFER	A	Allocates a buffer and returns its address.	ERROR
GETFN	0882	.FILENAME	A	2DA4			Passes params to CANON; calls ERROR if needed.	CANON ERROR
		.INTFN	A	2DA6				



NAME----	E.P.	PARAMETER	T	ADDR	RETURNS	T	PURPOSE----------	CALLS---
ISIS	0040	COMMAND	B	2921			Principal user entry point to operating system.	ATTRIB CLOSE CONSOLE DELETE EDIT ERROR EXIT LOAD OPEN READ RENAME RESCAN SEEK WHOCON WRITE XEQIOPB ?OPEN
		BLOCK	A	2922				
LOAD	22EC	.FILENAME	A	2DE8			Loads and optionally executes an absolute binary file.	CLOSE ERROR MEMCK OPEN PACKBUFS READ
		BIAS	A	2DEA				
		RETSW	A	2DEC				
		.ENTRY	A	2DEE				
MPY	0A0C	(BC)	A	--	(DE)	A	Compiler-generated multiplication routine. (DE) = (BC) * (DE)	
		(DE)	A	--				
OPEN	1D95	.FILENAME	A	2DA2	AFTN	B	Opens a file given its name; assigns and returns index in file table.	ALLOCATE CLRBUF ERROR FREE1 FREE3 GETAFT GETBUF GETFN REWIND RWMAP SCRATCH SEARCH SETTABS SYNC WRITE WRTSEC
		ACCESS	A	2DA4				
		ECHOAFT	A	2DA6				
FREE1	1DA2	ERRNUM	B	2DB0			Frees line-edit buffer (if any), then calls ERROR(0, ERRNUM).	ERROR FREEBUF
FREE3	1DC3	ERRNUM	B	2DB1			Frees link and data buffers, then calls FREE1(ERRNUM).	FREEBUF FREE1
PACKBUFS	0415			2B20	LOWLIM	A	Moves buffers currently in use to lowest possible locations; returns lowest vacant address.	LOWBUF
LOWBUF	0418	.BUFPTR	A	2B22			Moves buffer whose address is in BUFPTR to a lower position, if one is available.	FREEBUF GETBUF
RDSEC	0658	TRKSEC	A	2B44			Reads requested disk block into BUFFER, from proper drive for current file.	XIO
		.BUFFER	A	2B46				
READ	0E63	AFT	B	2CDD			Reads characters from file into BUFFER until COUNT is satisfied or end-of-file is reached.	CI ERROR FIO RI SETDEV SETTABS
		.BUFFER	A	2CDE				
		COUNT	A	2CE0				
		.ACTUAL	A	2CE2				
RENAME	1AE8	.OLDNAME	A	2D8E			Changes the name of a file.	ERROR GETFN SEARCH SETTABS SYNC WRITE
		.NEWNAME	A	2D90				
RESCAN	1276	AFT	B	2D29			Sets the next-byte pointer of a line-edited file to the beginning of the current line.	ERROR SETTABS
REWIND	12D4				--		Initializes several file tables and reads in first link block.	RDSEC
RWMAP	08AC	DRIVE	B	2CC0			Reads or writes bit map on requested drive, using internal buffer.	OPMAP
		OPCD	B	2CC1				
OPMAP	08B5				--		Constructs IOPB'S for RWMAP and has them run.	XEQIOPB
SCRATCH	1B87	DRIVE	B	2D95			Deletes all disk space allocated to file whose directory entry is at 2D4A.	DEALLOC RWMAP SEEK SETTABS WRITE XEQIOPB
SEARCH	1723	.INTFN	A	2D5C			Looks up file represented by INTFN in appropriate directory.	FREEBUF GETBUF MPY READ REWIND SEEK SETTABS

NAME----	E.P.	PARAMETER	T	ADDR	RETURNS	T	PURPOSE----------	CALLS---
SEEK	12F0	AFT	B	2D2D			Repositions the next-byte pointer of a disk file.	ALLOCATE CLRBUF ERROR OFFEND RDSEC REWIND RWMAP SETTABS SIZECK WRTSEC
		MODE	A	2D2E				
		.BLOCKNO	A	2D30				
		.BYTENO	A	2D32				
OFFEND	12FD	BLKNUMA	A	2D44	LARGER	L	Returns TRUE if requested seek would enlarge file.	
		BLKNUMB	A	2D46				
		BYTENUMA	B	2D48				
		BYTENUMB	B	2D49				
SETDEV	0E3D			--			Sets up RON's "IOBYT" to steer I/O properly.	IOCHK IOSET
SETTABS	02C8	AFT	B	2B1D			Sets up "current pointers" into file tables.	
SIZECK	0AC0			--			Updates file length in tables.	
SYNC	1711			--			Forces delayed write of current data block.	WRTSEC
WHOCON	22A0	AFT	B	2DE1			Copies into BUFFER the name of the requested console file.	
		.BUFFER	A	2DE2				
WRITE	0F12	AFT	B	2CF7			Writes COUNT characters on file from BUFFER.	CO ERROR FIO LO PO SETDEV SETTABS
		.BUFFER	A	2CF8				
		COUNT	A	2CFA				
WRTSEC	067B	TRKSEC	A	2B48			Writes requested block from BUFFER, on proper drive for current file.	XIO
		.BUFFER	A	2B4A				
XEQIOPB	0514	DRIVE	B	2B29			Runs given IOPB on requested drive.	ERROR
		.IOPB	A	2B2A				
XIO	05F6	OPCODE	B	2B34			Constructs IOPB to perform requested operation, and has it run.	XEQIOPB
		DRIVE	B	2B35				
		TRKSEC	A	2B36				
		.BUFFER	A	2B38				
XOPEN	2588			--			Increments user-supplied AFTN value by 2 to obtain internal value; if this does not correspond to an open file, calls ERROR(0, 2).	ERROR

Table III. ISIS Variables of Diagnostic Significance

ADDRESSES	CONTENTS----------
28C0-2920	Unused area, reserved for future expansion. Might be used for temporary debugging code, or for "patches."
2928-2931	Copy of current command parameter block.
294C-294D	User's Stack Pointer.
294E-298D	ISIS stack.
299C	Debug Switch (Logical). When True, fatal errors will invoke the MDS Monitor instead of aborting the program.
299D	"CONSOLE" command flag. Value is 2 while processing a CONSOLE system call, and zero at all other times. This is what causes all errors detected by CONSOLE to be fatal.
299E-299F	"Result Byte" and "Result Type" generated by disk channel in connection with a "Hard" (non-recoverable) disk error. For details, see the MDS-DOS Hardware Manual. (These are printed as part of the "ERROR 24" message.)
29CB-29DD	Buffer Table, containing one byte for each of 19 possible buffers. The possible values of each byte are: 0 Free. 1 Space preempted by LOAD since the buffer area contains part of the user program. 2 In use as a buffer. The buffers themselves begin at address 2E00; each buffer is 128 bytes long.
29E5-2B18	File Structure (see Table IV).
2B29	Disk drive most recently accessed.
2B2A-2B2B	Address of IOPB.
2B2C	Result Byte.
2B2D	Result Type.
2B4C-2B7S	Table of 2-byte device names (each name stored backwards).
2B86-2B8F	Internal-format name of last file sought by SEARCH. 2B86 contains the device identification (index in table at 234C), 2B87-2B8C contain the file name, 2B8D-2B8F contain the name extension.
2BA8-2CA7	Bit Map Buffer.
2CA8	Drive to which this bit map belongs.
2CA9	(Logical) True if bit map has been modified since it was last written.
2D4A-2D59	Directory Entry for last file found by SEARCH, as it then appeared. (This is not kept current with respect to growth of the file.)

ADDRESSES	CONTENTS
2DB2-2DC0	Name of current :CI: file.
2DC1-2DCF	Name of current :CO: file.
2DF6-2DF7	Address field of last logical record read by LOAD. (Except during a LOAD, this turns out to be the entry point of the last file LOADed.)
2E00-	Buffer Area.

Table IV. ISIS File Control Structure (in memory)

NAME	BASE ADDR	T	CUR. PTR.	I.V.	DESCRIPTION
CLOSED	2A0E	L	29EA		True if this file is not open.
DEVICE	2A18	B	29EC		Device identification (see Table V).
ACCESS	2A22	B	29EE		Value of OPEN's ACCESS parameter.
ECHOAFT	2A2C	B	29F0		AFT of echo file; zero if non-edit.
EBUF	2A36	A	29F2		Address of edit buffer.
DBUF	2A4A	A	29F4		Address of data buffer (copy of current element at 29E8).
BYTENO	2A5E	B	29F6	128	Byte counter in data buffer.
DNUM	2A68	A	29F8		Position in directory.
LBUF	2A7C	A	29FA		Address of link-block buffer (copy of current element at 29E6).
DPTR	2A90	B	29FC	1	Word pointer in link block.
LASTBYTE	2A9A	B	29FE		Number of bytes in last data block.
ALLOC	2AA4	L	2A00	F	True if allocation has been done for this file.
DIRTY	2AAE	L	2A02	F	True if current data block has been modified.
BLKCNT	2AB8	A	2A04		Number of data blocks in this file.
BLKNO	2ACC	A	2A06	0	Sequential number of current data block within this file.
LADDR	2AE0	A	2A08	(2AF4)	Track-Sector address of current link block.
L1ADDR	2AF4	A	2A0A		Track-Sector address of file's first link block.
DADDR	2B08	A	2A0C	0	Track-Sector address of current data block

Table V. ISIS Device Identification Codes

NUMBER	DEVICE
0	:F0:
1	:F1:
2	:TI:
3	:TO:
4	:VI:
5	:VO:
6	:I1:
7	:O1:
8	:TR:
9	:HR:
10	:R1:
11	:R2:
12	:TP:
13	:HP:
14	:P1:
15	:P2:
16	:LP:
17	:L1:

Table VI. ISIS Format Files

FILE NAME	HEX DISK ADDRESSES LINK	---DATA---	CONTENTS
ISIS.TO	0,18	0,1 - 0,17	System Initialization Program.
ISIS.LAB	0,19	0,1A	Disk Label.
ISIS.DIR	1,1	1,2 - 1,1A	Disk Directory (see Table VII).
ISIS.MAP	2,1	2,2 - 2,3	Allocation Bit Map.

Table VII. ISIS Directory Entry

BYTES	CONTENTS
00	Flag: 00 = active 7F = never used FF = deleted
01-06	File Name
07-09	Name Extension
0A	Attributes: Bit 0 = Invisible Bit 1 = System Bit 2 = Write-protect Bit 7 = Format
0B	Number of bytes in last data block
0C-0D	Number of data blocks
0E-0F	Disk Address (Block, Track) of first linkage block

Intel Corporation, 3065 Bowers Avenue, Santa Clara, California 95051

ISIS.DIR FORMAT

Ø	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
ØØ	1	S	1	S			T	Ø		B7					
Ø Ø	1	S	1	S			M	A	P	B7					
ØØ	1	S	1	S			D	1	R	B7					
ØØ	1	S	1	S			L	A	B	B7					
ØØ	C	R	E	D	1	T				Ø6					
FF	M	Y	P	R	O	G	O	B	J	Ø4					
7F															
7F															

Ø - ACTIVITY
ØØ - OPEN CODE
FF - DELETED
7F - NEVER USED

1-6 FILE NAME 6 BYTES ASCII

7-9 EXTENSION 3 BYTES ASCII

A - ATTRIBUTES 1 BYTE

BIT	7	6	5	4	3	2	1	Ø
	F	X	X	X	X	W	S	I

CODE

B- NUMBER OF BYTES IN LAST DATA BLK. SO ISIS WILL KNOW WHERE THE END OF FILE IS HEX

C-D NUMBER OF DATA BLKS USED BY THE FILE HEX

C	D
LSB	MSB

E - SECTOR OF 1ST LINKAGE OR HEADER BLOCK

F - TRACK OF 1ST LINKAGE OR HEADER BLOCK

ALL REFERENCES ARE BASE HEX AND 'H' MUST BE USED WITH GONIF AS IT DEFAULTS TO BASE 10

— LINKAGE BLOCK FORMAT —

DSK ADDRESSES OF DATA BLOCKS ON DISKETTE

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E
0	PRECEDING		FOLLOWING												
1															
2															
3															
4															
5															
6															
7															

SEC	TRK	SEC	TRK	SEC	TRK	SEC	TRK	. . . .
ADDRESS OF PRECEDING LINKAGE BLK. SET TO 0 IF THIS IS FIRST BLK		ADDRESS OF FOLLOWING LINKAGE BLKS SET TO 0 IF THERE ARE NONE		ADDR OF DATA BLOCKS				
2 BYTES		2 BYTES		124 BYTES				

FILE	LINK BLK ADDR TRK	SECTOR	DATA BLK ADDR FROM TRK	SEC	TO TRK	SEC	
ISIS.T0	0	18H	0	1	0	17H	
ISIS.DIR	1	1	1	2	1	1AH	VARIABLE
ISIS.MAP	2	1	2	2	2	5	
ISIS.LAB	0	19H	0	1AH	0	34H	
			1	1BH	1	34H	
ISIS.BIN	2	6	2	7	2	34H	
	[illegible]		3	1	3	10H	
	3	11H	3	12	3	2FH	
ISIS.CLI	3	30H	3	31H	3	34H	
			4	1	4	10H	
SIGNON MSG.	0	16H					

ISIS.MAP DISPLAYS THE NUMBER AND LOCATION OF BLOCKS USED (SET TO FF) AND NUMBER AND LOCATION OF FREE SPACE (SET TO 00) (SEE ADDITIONAL INFO)

EXAMPLE:

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E
0	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	
1	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	F	
2	FF	FF	FF	FF	00	00	00	00	00	00	00	00	00	00	
3	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
4	00	00	00	00	00	00	00	00	00	00	00	00	00	00	

ISIS BREAKDOWN

TRACK	SECTOR	USE
0	1-52	ISIS.T0
1	2-52	ISIS.DIR
1	1	HEADER BLOCK FOR ISIS.DIR
2	1	HEADER BLOCK FOR ISIS.MAP
2	2-5	ISIS.MAP
2	6	HEADER BLOCK FOR ISIS.BIN
3	30H	HEADER BLOCK FOR ISIS.CLI

ISIS.LAB CONTAINS THE DISK LABEL ONLY. SECTORS 1AH-34H ON TRACK 0, SECTORS 1BH-34H ON TRACK 1 HAVE BEEN ALLOCATED FOR ISIS.LAB.. THATS QUITE A LOT OF FREE SPACE.....

ISIS.MAP: THE SECTORS USED PER GIVEN TRACK (OR DISK) ARE LISTED SEQUENTIALLY STARTING AT TRACK 2 SECTOR 2, BIT 0 OF BYT. 0. (BIT 0 OF BYTE 0 WOULD BE SECTOR 1 OF TRACK 0) IF A GIVEN BIT IN ISIS.MAP IS SET TO A ONE, THE CORESPONDING SECTOR ON THE DISKETTE IS BEING USED. THE BIT PATTERNS ARE READ SEQUENTIALLY AND 52 SECTORS ARE ALLOWED FOR DOUBLE DENSITY, 26 FOR SINGLE..

ISIS-II Diskette Operating System Folklore, V1.0

Steve Kreuscher

This discussion is intended to answer a few questions regarding the ISIS-II operating system and its diskette structure. It is not intended to be a complete breakdown of ISIS-II. It is not complete; any inputs for additions will be appreciated.

1.0 ISIS.DIR

1.1 Purpose - Directory of files for use by ISIS.CLI

1.2 Location

Linkage block - Sector 1, Track 1

Data blocks - from Sector 2, Track 1
through Sector 1AH, Track 1

1.3 Description

This is not to be confused with the diskette file called DIR. DIR is a program which accesses ISIS.DIR to list the names of the currently open files on the diskette. ISIS.DIR is actually a table. Each entry consists of 16 bytes. When GANEF is used to display a block of ISIS.DIR, the CRT display consists of the 128 hex bytes that make up the block and their ASCII equivalents:

STAT	NAME						EXT			ATT.	BRAM	BLOCKS USED LO	BLOCKS USED HI	LINK SEC	LINK TRK	
0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0123456789ABCDEF
00	49	53	49	53	00	00	4D	41	50	81	80	04	00	01	02	.ISIS..MAP.....
00	49	53	49	53	00	00	54	30	00	81	80	17	00	18	00	.ISIS..T0.....
00	43	52	45	44	49	54	00	00	00	06	26	36	02	14	09	.CREDIT....&6...
FF	4D	59	50	52	4F	47	4F	42	4A	04	08	02	00	1B	06	.MYPROGOBJ.....
7F	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
7F	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
7F	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
7F	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....

Byte 0 - Current status of file
00 = in use
7F = never used
FF = deleted
Note that when a file is deleted, the file itself remains intact; only this byte of ISIS.DIR changes.

Bytes 1 through 6 - File name

Bytes 7 through 9 - Extension

Byte A - Attributes
bit 0 set = invisible file
bit 1 set = system file
bit 2 set = write protected file
bit 7 set = format file
(bits 3 through 6 not used)

Byte B - Number of bytes in last data block. This is used to determine the location of the end of file.

Bytes C and D - Number of data blocks used by the file.
byte C = least significant byte
byte D = most significant byte

Byte E - Sector of the first linkage block.

Byte F - Track of the first linkage block.

Last Data Block

ISIS-II only stores data on diskettes in 128-byte blocks. The number of bytes in a file will not often be an exact multiple, but the system will occupy blocks in integer increments only. Thus, the system needs to know the number of bytes used in the last block of the file, in order to determine the whereabouts of End-of-File.

Linkage Blocks

Since every file is broken up into 128-byte pieces and strewn about wherever there is room on the diskette, ISIS-II needs to know where the pieces are, and in what order to load them. That information is contained in the linkage block. Each file has a linkage block for every 124 data blocks. It lists the sector and track of each data block in the order in which they are to be loaded. Why only 124? Because the first two bytes of the block show the sector and track of any preceeding linkage block, and the next two bytes show the sector and track of any subsequent linkage block. For any file that is less than 124 blocks, these will be 00.

2.0 ISIS.T0

2.1 Purpose - Track 0 Loader

2.2 Location

Linkage block - Sector 18H, Track 0

Data blocks - from Sector 1, Track 0
through Sector 17H, Track 0

2.3 Description

This is the file pulled in by the system's monitor when the diskette system is initialized. It begins at

sector 1 of track 0 ("home" position for the drive), so it is immediately loaded as the first action of the diskette operating system.

3.0 ISIS.MAP

3.1 Purpose - Bit Map of Diskette Blocks

3.2 Location

Linkage block - Sector 1, Track 2

Data blocks -	from	Sector 2, Track 2
	through	Sector 5, Track 2.

3.3 Description

This file is a bit map of all diskette blocks, used to determine which blocks are in use. Each of the 4004 blocks of the diskette is assigned one bit in the map. A set bit indicates that the block is in use.

Sector 1 of Track 0 is represented in the map by Bit 7 of Byte 0 of ISIS.MAP. Sector 2 is represented by Bit 6, Sector 3 by Bit 5, etc. Since there are 4004 blocks on a double-density diskette, ISIS.MAP needs to store 4004 bits. Thus, it occupies four blocks (4096 bits) of data.

4.0 ISIS.LAB

4.1 Purpose - Diskette Label

4.2 Location

Linkage block - Sector 19H, Track 0

Data blocks -	from	Sector 1AH, Track 0
	through	Sector 34H, Track 0
	from	Sector 1BH, Track 1
	through	Sector 34H, Track 1

4.3 Description

This space of 53 blocks (6784 bytes) is reserved for the label of the diskette, which is limited by ISIS to nine ASCII characters (six for name, three for extension). The label may only be issued at the time of diskette initialization, and can only be altered using GANEF (or equivalent). RENAME won't work.

Yes, this does seem like a lot of space for storing nine characters. Actually, the version number of ISIS-II V4.0 is also stored here, so that brings the

total to eleven. Room for future expansion?

5.0 ISIS.BIN

5.1 Purpose - Binary file of Operating System

5.2 Location

Linkage blocks - Sector 6, Track 2
Sector 11H, Track 3

Data blocks -	from	Sector 7, Track 2
	through	Sector 34H, Track 2
	from	Sector 1, Track 3
	through	Sector 10H, Track 3
	from	Sector 12H, Track 3
	through	Sector 2FH, Track 3

5.3 Description

This is the main brains of ISIS-II. This is the program that actually runs the system when ISIS-II is operating.

6.0 ISIS.CLI

6.1 Purpose - Command Line Interpreter

6.2 Location -

Linkage block - Sector 30H, Track 2

Data blocks -	from	Sector 31H, Track 3
	through	Sector 34H, Track 3
	from	Sector 1, Track 4
	through	Sector 11H, Track 4

6.3 Description

This is the portion of ISIS-II which accepts command lines from the operator and determines what to do with them. If you issue an invalid command, this is where it gets rejected.

ISIS-II Version 4

EXTERNAL REFERENCE SPECIFICATION

Revision 2

January 2, 1979

by

S. Fung

R. Harper

T. Stolz

Intel Corporation
3065 Bowers Avenue
Santa Clara, California 95051

TABLE OF CONTENTS

PREFACE

I. Substantive differences between the ERS for ISIS-II Version 3, revision 2, and the ERS for ISIS-II Version 4, revision 0, are:

1. Section 1.3.1. Addition of hard disk drives.
2. Section 3.7. Addition of Monitor work area to memory layout diagram.
3. Section 3.8. Addition of explanation of buffer allocation.
4. Section 3.8.15. Addition of hard disk devices to device number table.
5. Section 3.9. Addition of hard disk platter format information. Deletion of Attachments A and B.
6. Section 3.10. Addition of directory information for the hard disk.
7. Section 3.12.1. Addition of switches 6,7,8,9,0,T.
8. Section 3.12.4. Allow display of file attributes without having to specify a switch setting.
9. Section 3.12.6. Addition of K switch and target devices :F6:, :F7:, :F8:, :F9:.
10. Section 3.12.10. Specification of nesting levels of SUBMIT.
11. Section 3.12.11. Addition of K switch.
12. Section 3.12.12. Addition of HDCOPY.
13. Section 3.13. Addition of hard disk configurations.
14. Section 3.14. Description of changes to T0BOOT.
15. Section 4.1. Addition of cold start procedures in a hard disk environment.
16. Appendix A. Addition of correct PL/M external declarations.
17. Appendix E. Addition of description of selected ISIS-II internal data structures.

II. Differences between ISIS-II Version 4, Revision 0 and Revision 1, (other than typographical corrections) fall into two categories. The first category concerns those changes which amplify or clarify the operation of ISIS-II but which do not represent a change in the operation of ISIS-II. Major changes in this category include:

1. Section 1.3.1 "four hard disk drives (= 2 disk boxes)" changed to "two hard disk drives (4 platters)".
2. Section 2.4 Remove references to PL/M version numbers.
3. Section 3.2 "Control-Z has no echo." changed to "Control-Z is echoed as CR, LF."
4. Sections 3.6, 3.12, 3.12.1-3.12.12 BNF notation revised.
5. Section 3.6 Insert paragraph concerning relationship of uppercase to lowercase letters.
6. Section 3.6 "starting address" changed to "load address".
7. Section 3.7 Clarified relationships of UOP,TOB.

8. Section 3.7 "one or more I/O buffers" changed to "three or more I/O buffers".
9. Section 3.7 "the more buffers ISIS-II may allocate for the user's benefit" changed to "the more buffers ISIS-II may allocate for the user's benefit (up to the maximum of 19)".
10. Section 3.8.2 Rewritten.
11. Section 3.8.3 Change "RAM area" to "memory area".
12. Section 3.8.4 Added description to Mode=2.
13. Section 3.8.7 Rewritten.
14. Section 3.8.8. Added description of FILE$POINTER.
15. Section 3.8.8. Expanded description of RETSW=1 and RETSW=2.
16. Section 3.8.15. Corrected Byte Bucket description. Deleted references to nulls in filename and filename extension.
17. Section 3.9. Expanded description of hard disk mapping.
18. Section 3.10. Added EDIT to list of ISIS-II cusps.
19. Section 3.11.5. "is closed" changed to "is opened or closed".
20. Section 3.12. Wildcard description moved to Section 3.6.
21. Section 3.12. Clarified explanation of invalid pathname and switch processing.
22. Section 3.12.1 "of the files in the specified disk directory" changed to "of the filenames in the specified disk directory".
23. Section 3.12.1. Clarified Pause Switch.
24. Section 3.12.3.1.A 2nd paragraph reworded. Corrected output messages.
25. Section 3.12.4. Attributes modified message is not displayed.
26. Section 3.12.5. Corrected output messages.
27. Section 3.12.10.B "SUBMIT will use the default extension .CSD" is changed to "SUBMIT will assume the default extension .CSD".
28. Section 3.12.10.3. Added cautionary note.
29. Section 3.12.10.4. Added clause concerning nested SUBMITs.
30. Section 3.12.11. Last paragraph amended.
31. Section 3.14. T0BOOT will output the appropriate error message when an incompatible hard disk controller is used.
32. All references to COLONEL changed to KERNEL.
33. Appendix E. Added cautionary note concerning changeability of internal data structures.

The second category concerns those changes which do represent a change in the operation of ISIS-II. Major changes include:

1. Section 3.12.6 FROM integer added. Drive 0 is now a valid target device.
2. Section 3.12.11 FROM integer added.
3. Section 3.12.12 HDCOPY revised - added verification switch and backup switch.
4. Section 3.13 The hard disk fixed platters will be :F0: and :F2: instead of :F1: and :F3:.
5. Section 4.3 and Appendix C. Disk error messages changed from "FDCC" TO "STATUS".

III. Differences between ISIS-II (Version 4) Revision 1 and Revision 2 (other than typographical corrections and minor rewording) are:

1. Section 3.2 If the console echo file is :VO: a Rubout will be echoed as a backspace, blank, backspace.
2. Sections 3.6, 3.12.1, 3.12.6, 3.12.10.2, 3.12.11 Brackets were mistakenly left off the BNF representation of the syntax.
3. Section 3.12.1 "Z" switch added.
4. Section 3.12.12 Specification of HDCOPY error messages. Also, processing will continue upon detection of a miscompare in the verification process instead of exiting. More information will be displayed during BACKUP operation.

1.0 PRODUCT IDENTIFICATION

1.1 Name, Mnemonic, and Project Number

Name: Intel Systems Implementation Supervisor

Mnemonic: ISIS-II V4.n

Project Number: 2816

1.2 Product Abstract

ISIS-II provides a set of services normally required to execute programs on an Intellec MDS or Intellec Series 2 development system such as: supervisory function, logical input/output facilities, and file management capabilities. ISIS-II must be used in conjunction with a ROM-Resident Monitor and it requires at least one floppy disk device configured in the hardware system.

1.3 Product Use Environment

1.3.1 Hardware

ISIS-II operates in an Intellec MDS or Intellec Series 2 microcomputer system with at least 32K bytes of RAM memory, a flexible disk drive, and a console device (such as TTY or CRT). ISIS-II can support up to six floppy disk drives and two hard disk drives (4 platters) in certain configurations, a full complement of 64K RAM memory and all peripherals currently offered in the Development Systems line.

1.3.2 Software

ISIS-II requires access to the I/O system of either the Intellec MDS Monitor, Version 2.0, or the Intellec Series 2 Monitor, Version 1.2.

2.0 FUNCTIONAL SPECIFICATIONS

2.1 General Characteristics and Scope of Product

ISIS-II is a disk operating system intended to simplify a microcomputer development effort by providing a convenient environment for source editing, assembly/compilation, linking, locating, debugging, and simulation. Major emphasis is placed on ease of use of ISIS-II, based on the assumption that the typical user is not a senior systems programmer, but more likely, an electronic engineer involved in his first software project.

2.2 Description of All Major Functions Performed

For purposes of exposition, ISIS-II is considered to comprise two components: KERNEL and a collection of programs called "user programs."

KERNEL may be conceived as a subroutine collection which is permanently resident in low memory during ISIS-II execution.

KERNEL provides facilities for loading and executing user programs, and serves the I/O needs of user programs. User programs may be Intel-supplied (e.g., the Editor, Assembler, PL/M), or they may be written by the ISIS-II user.

Whenever the ISIS-II user is communicating to software (i.e., by typing at the console device), he is in communication with a user program, never with the KERNEL.

User programs achieve I/O by making calls ("system calls") to subroutines within the KERNEL. All I/O occurs to/from "files." A program normally "opens" a file, then "reads" from it or "writes" to it, and finally "closes" it. All I/O is status-driven, not interrupt-driven, and except for disk I/O is achieved by use of subroutines in the Monitor.

There is a system call (EXIT, see Section 3.8.7) which user programs make to terminate their execution (and RAM-residency). This system call causes KERNEL to load and run a special user program called CLI ("Command Language Interpreter").

CLI reads and interprets command lines provided by the user. Command line syntax is described in Section 3.6. Briefly, each command contains a generalized keyword which specifies either a user program or CLI command. If the former, CLI causes that program to be loaded and run; otherwise CLI performs the indicated command and reads another command line.

2.3 Performance Characteristics

The most important requirement placed on ISIS-II is the maximum size of its data and code areas which cannot exceed the combined size of 3000H (12K) bytes. This requirement will be satisfied but, as a result, speed of execution may suffer. It should also be pointed out that the system response time will depend on the bus speed and the data throughput of the particular disk controller to which the disk on which ISIS-II resides is connected.

The hardware does not permit ISIS-II to protect itself from user programs. Operation of ISIS-II conformance to this ERS is predicated on the condition that the user program does not modify RAM below its origin point (UOP, see Section 3.7) or within the Monitor work area.

2.4 Applicable Standards

ISIS-II is written in 8080 Resident PL/M. All system calls from a user program to ISIS-II conform to the parameter calling sequence of PL/M.

Throughout ISIS-II, the character set used is ASCII (USAS X 3.4-1968). The flexible diskettes and controllers and hard disk platters and controller used by ISIS-II support a soft-sectored format with 128 bytes of data per sector.

ISIS-II supports absolute load files of the Object Module Formats.

2.5 Syntax Description Conventions

Standard Backus Normal Form is used. The symbols "<", ">", "::=" and "!" are the usual meta-linguistic symbols; when they are required as terminal symbols, they are enclosed by braces ("{" and "}"); these braces are also used to enclose concepts defined informally in English.

2.6 Nomenclature

This document distinguishes meanings for "diskette", "platter", "drive" and "disk":

"Diskette" is the recording medium for floppies while "platter" is the recording medium for the hard disk; "drive" is the mechanism on which the medium is mounted; "disk" is the drive together with a mounted diskette/platter. Where the meaning is unambiguous, "disk" is often used in place of "diskette" and "platter."

"MONITOR", "MDS MONITOR", and "SERIES 2 MONITOR" are also used interchangeably, but where necessary, distinctions are made between the MDS Monitor and the Series 2 Monitor.

3.0 INTERFACE SPECIFICATIONS

3.1 ISIS-II File Structure

A "file" is an abstraction of an I/O device, and may be considered to be a collection of information, usually in machine-readable form. Throughout this document, a file is formally defined as a sequence of 8 bit values called "bytes".

ISIS-II usually places no semantic interpretation on the byte values of a file. The single exception is lined files (see Section 3.2). However humans, programs, and devices will frequently assume that the bytes represent ASCII values, and thereby characters.

Programs receive information by "reading" from an "input file", and transmit information by "writing" to an "output file".

A major purpose of ISIS-II is to implement files (called "disk files") on diskettes/platters. Every disk file is identified by a name unique on its diskette/platter, which has 2 parts: a filename and an optional extension. A disk file's filename is a sequence of from 1 to 6 ASCII characters; an extension is a sequence of from 1 to 3 ASCII characters. To facilitate name specification within command strings, these ASCII characters are constrained to be letters and/or digits.

For every non-disk device supported by ISIS-II, there are one or more associated files, each identified by a name consisting of a pair of ASCII characters between colons (see Appendix D for a complete list). Disk drives also have names which are prefixed to filenames to specify on which disk the file resides.

No file can exist on more than 1 physical device. In particular, a disk file must reside entirely on one diskette/platter.

Three files (:BB:, :CI: and :CO:) deserve special mention:

ISIS-II supports a virtual input/output device known as a "Byte Bucket" (:BB:). This device acts as an infinite sink for bytes when written to, and a file of zero length when read from. Multiple opening of :BB: is allowed, each open returns a different AFTN. (See Section 3.8.1).

ISIS-II supports a virtual teletype known as "the Console," which is implemented as 2 files, an input file (:CI:) and an output file (:CO:). These 2 files are always "open" (see Section 3.8.1); :CI: is always a "lined file", :CO: is its associated echo file (see Section 3.2). Each of :CI: and :CO: is a pseudonym for some file corresponding to an actual physical device. After a cold start of ISIS-II (see Section 4.1.1), :CI: and :CO: will reference either the teletype (:TI:

and :TO:) or the video terminal (:VI: and :VO:), which will be called the "cold start Console"; however user programs may "move" the two halves of the Console from one physical device to another (see Section 3.8.12)

Whenever an end of file is encountered on :CI:, then both :CI: and :CO: are automatically "moved" to the cold start Console.

It is always from the current Console that CLI obtains its command lines. (See Section 3.6).

3.2 Input Line Editing

Programs, when opening a file for input, may optionally request ISIS-II to "filter" the input data through a module called line editor. This option caters to the frequent situation where a human is typing input for a program in real time at a keyboard console but is not restricted to such situations.

Files read in this fashion are called "lined files." (Note that a file is so characterized not because of any attribute intrinsic to the file, but merely by its current method of access).

Every lined file has associated with it an output file, which is the file on which the echo is printed. This allows programs to ignore complications of input echoing; ISIS-II does it automatically. If no echo is desired, as may be the case when a disk file is a lined file, the associated echo file may be :BB:.

ISIS-II interprets bytes in a lined file as 7-bit ASCII codes (the high order bit is ignored); furthermore, special interpretations (described below) are placed on the following byte values:

VALUE	CHARACTER
0AH	LF (Line Feed)
0DH	CR (Carriage Return)
12H	Control R
18H	Control X
1AH	Control Z
1BH	ESC (Escape)
7FH	Rubout
05H	Control E
10H	Control P

LF and ESC are defined as "break characters", with semantics defined below.

Lined files are conceptually partitioned into segments, called "logical lines," by the following rules:

1. A LF is inserted following every CR, and then all LF's immediately following a break character are removed;

2. A logical line is defined to be all characters between break characters, together with the terminating break character;

3. If all logical lines comprise no more than 122 "uncancelled" characters (by the editing transformations defined below), the partitioning is complete; otherwise the "long" logical lines are themselves further partitioned into 2 segments: the left segment comprises the largest possible number of characters such that no proper substring of those characters comprise more than 121 "uncancelled" characters; the right segment comprises the remaining characters;

4. Rule (3) is applies as many times as necessary to eliminate "long" logical lines.

A READ call (see Section 3.8.2) returns bytes from only one logical line at a time; thus READ's of lined files often transfer fewer bytes than requested by COUNT.

A READ system call returns no characters from a logical line until the line has been input in its entirety. Thus, during physical input, the logical line is accumulated in an internal buffer; no information in the buffer is transferred to the READing program until the termination character(normally a LF --- see Rule 1) is seen. Therefore ISIS-II has the opportunity to modify buffer contents conditionally on values entertaining the buffer. This is the mechanism of line editing, which permits the following manipulations:

A CR character entering the buffer has the effect that a LF character is automatically appended to it (this is rule 1 above), and both are echoed. Thus the CR character may be used at a keyboard to terminate an input line.

A LF character as the first (and therefore only) byte in a line has no effect (this is also rule 1 above); it is discarded; there is no echo. This permits disk files with CRLF line terminators to be used as lined files; the CR generates an LF, yielding CRLFLF, but then the 2nd LF is removed from the buffer and is ignored.

A Rubout character has the effect that it cancels both itself and the most recent uncancelled byte in the buffer. In general a Rubout is echoed as the character it cancels. However, in the case that the console echo file is :VO: a Rubout is echoed as a

sequence of backspace (08H), blank character (20H), and a backspace. If there is no uncancelled character remaining in the buffer, the Rubout has no effect and is echoed as a Bell (ASCII 7).

A Control-X character cancels all characters in the buffer, including itself, thereby erasing the current input line read in so far. It is echoed as a '#', CR and LF.

A Control-R echoes a CR and LF, followed by the current uncancelled contents of the buffer.

A Control-Z cancels all characters in the buffer including itself, and causes the READ call to return immediately without transferring any bytes, thus simulating an end of file. The remainder of the logical line, if any, may be read by a further READ call. This is the only way to obtain an end of file indication on keyboard input devices. Control-Z is echoed as a CR and LF.

An ESC character is echoed as a dollar sign ('$') character.

The function of a Control-E is described in Section 3.4.

The function of a Control-P is described in Section 3.5.

3.3 Operator Controlled Pause

ISIS-II provides a pause facility for all console output devices (:VO:, :TO:, :CO:), to allow the operator to stop scrolling of output, inspect the display, and then continue scrolling. Two control keys are used as follows:

1. If Control-S (X-OFF) is entered from the keyboard of the physical device which corresponds to the current :CO: device, the display stops. The display will remain stopped until a Control-Q (X-ON) is entered from the same device.

2. All intervening characters entered between control-S and Control-Q are ignored.

3. The above operations have no effect on any input operations. Control-S and Control-Q are not considered to be line editing characters.

Note that stopping the display also stops the program generating the display, along with the rest of the system. Entering a Control-S will cause the program executing to pause at its next console output.

Since this feature is associated with physical devices below the file level, a pause is controlled from the physical device

paired with the output device. Thus, pauses on :VO: are controlled from :VI:, no matter where the console input (:CI:) is originating.

3.4 Ability to Exit and Reenter Disk Console Input

In many cases, it is useful to interrupt a SUBMIT (see Section 3.12.10) job temporarily and accept input from the Cold Start Console and then continue the SUBMIT job. This is useful when a sequence of standard operations (e.g. ICE80 initialization) is followed by an interactive session (e.g. ICE80 debugging of a user program) which is then followed by another standard sequence (saving the modified program image, exiting). ISIS-II line-edited input logic checks for a Control-E (ENQ) which causes an exchange of the console input from the currently assigned file to the cold start console. A subsequent Control-E will perform the converse, since the Control-E is interpreted as a toggle. Control-E is echoed as an up arrow followed by an E (↑E) but is not returned in the input buffer.

At cold start time, whenever a CONSOL call is made, and whenever a fatal error or interrupt 1 occurs, the alternate console (the console to be made active when the next Control-E is encountered) is set equal to the cold start console. Therefore, entering a Control-E from the cold start console when no submit job has been suspended, results in a null operation.

3.5 Use Of A Quote Or Literal Character In Line-Edited Files

There are occasions when it is necessary and desirable to override the line-edited input conventions and input the literal value of a character (cr, lf, Control-R,...) which would otherwise be interpreted as a line-editing character. The Control-P (DLE) character is used to indicate that the following character is to be treated literally and to be placed in the input buffer.

3.6 Command Language

When ISIS-II (CLI) is ready to accept a command it prompts with a dash character ("-") at the beginning of a new line on the current Console output device (:CO:). Normally, the user now gives a command to CLI by typing a sequence of characters, followed by a carriage return. This character sequence, including the carriage return (and the automatically appended line feed) is a "command line", as defined further below. In general, CLI reads 1 line of input from the current console input device (:CI:). All line editor features are available at command-input time (see Section 3.2).

CLI also performs a conversion of the command head from lower case into uppercase characters. Commands (i.e. programs) must decide for themselves whether to treat uppercase and lowercase characters in the command tail as being equivalent. The ISIS-II cusps (Section 3.12), for example, convert all lowercase characters in the command tail into their uppercase equivalents. A command line must conform to the following syntax:

```
<command line> ::= <command head> <command tail> !
                   <comment line>
<command head> ::= [ DEBUG ]<command>
<command> ::= <pathname>
<command tail> ::= {a sequence of 0 or more characters,
                    not including a LF or ESC} [<terminator>]
<comment line> ::= ; {a sequence of 0 or more characters, not
                   including a LF or ESC} [<terminator>]
<terminator> ::=  CR ! LF ! ESC
<pathname> ::= <device> ! <fid> ! <device> < fid>
<device> ::=  : <c><c> :
<fid> ::= <filename> [ .<extension>]
<filename> ::= {a sequence of 1 to 6 <c> s}
<extension> ::= {a sequence of 1 to 3 <c> s}
<c> ::=  A!B!C!D!E!F!G!H!I!J!K!L!M!N!O!P!Q!R!S!T!U!V!W!X!Y!Z!a!b!
         c!d!e!f!g!h!i!j!k!l!m!n!o!p!q!r!s!t!u!v!w!x!y!z!0!1!2!3!
         4!5!6!7!8!9
```

Examples of command lines:

```
;THIS IS AN EXAMPLE
COPY  :F1:MYPROG.HEX TO  :HP:
IDISK  :F1:NEW.DSK
:TR:
:F1:SIMULA BROWN$L
EDIT JIM
DEBUG MYPROG :LPL$N
```

A command line must contain 122 or fewer characters. The terminating LF in the command tail will normally be inserted automatically by the line editing mechanism (see Section 3.2) when it encounters the CR.

The DIR, ATTRIB, DELETE, and COPY system commands incorporate a wild card facility in the command line. The syntax of the wildcard pathname element is as follows:

```
<wildcard pathname> ::= <device> ! <wildcard fid>!        --
                        <device><wildcard fid>
<wildcard fid> ::= <wildcard filename>[ .<wildcard extension> ]
<wildcard filename> ::= {a sequence of 1 to 6 <char>s}
<wildcard extension> :: {a sequence of 1 to 3 <char>s}
<char> ::= <c> ! * ! ?
```

1. In either filename or extension, the character '?' can be used as a token to match any valid non-null character.

   AB?.HEX matches ABC.HEX, ABX.HEX,...but not AB.HEX.

2. The character '*' is used to match all or the remainder of either filename or extension field. '*' can be interpreted as filling the rest of the field with don't care tokens.

   AB*.HEX matches ABC.HEX, ABCD.HEX, AB.HEX, ...
   A*.* matches ABC.HEX, A, ...
   A*C.HEX is illegal.

3. A device specifier, :Fx:, may be used to prefix a wildcard filename, but must be fully specified. (i.e. :F?: is not allowed).

A pathname normally specifies a file as follows: the device specifies one of the physical devices listed in Appendix D. If device is not specified, then the system disk (:F0:) is specified by default. If a disk is specified, then an fid must be specified. If a non-disk device is specified, than an fid may be specified, but has no significance.

Comment lines can be input interspersed with command lines. A comment line starts with ';' as its first nonblank character.

If the command line does not conform to the above syntax, an error message is sent to the current console output device (:CO:), together with another prompt. Otherwise the command specifies a file which is interpreted by ISIS-II as a file in ISIS-II Absolute Object Module Format (see Absolute Object File Formats, Intel Technical Specifications, 9800183B), which is to be loaded. After loading, one of two actions occurs, depending on debug: If DEBUG is not specified, then the DEBUG TOGGLE (see Section 4.3) is reset, and the program is executed. If DEBUG is specified, then the DEBUG TOGGLE is set, thereby entering debug mode, and control is transferred to the Monitor with the starting address of the program displayed, at which point the entire debugging facility of the Monitor is at the

user's command. There are 4 ways to leave debug mode (thereby resetting the DEBUG TOGGLE): (1) a user program can call EXIT (see Section 3.8.7), (2) a user program can call LOAD with RETSW=1 (see Section 3.8.8), (3) the user may press interrupt switch #1 while the user program is running (see Section 4.4), or (4) the user may execute a G8 command from the Monitor.

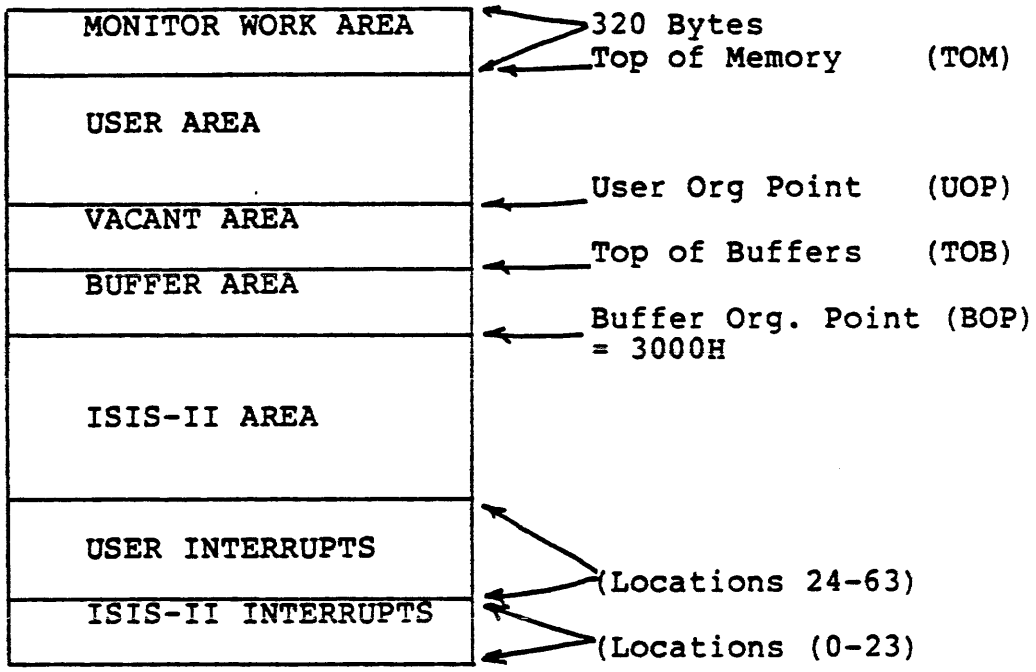
If the file specified by command does not exist, or if it has an illegal ISIS-II Absolute Object Module Format, or if its load address lies within the ISIS-II memory or buffer area, an error is issued (see Section 4.3) and another prompt is given. If the file does not correspond to a main program, then results are undefined.

By the use of RESCAN and READ calls (see Section 3.8), CLI arranges that the loaded program may read the command tail by normal READ calls on the current console input device (:CI:); the first character read will be the first character following the command head. If desired the command head may also be examined by use of the RESCAN system call in conjunction with READ.

The loaded program is free to implement its own semantics on the command tail. (Note: if the loaded program fails to READ the entire command tail, the unread portion will be flushed by CLI before processing the next command line.)

3.7 Memory Layout

Intellec memory is logically compartmented into 7 sections by ISIS-II: interrupt areas, the ISIS-II area, the buffer area, a vacant area, the user area, and the Monitor work area. This logical partitioning of memory is described by the following diagram:



Relative sizes of these memory areas are constrained by the following equations:

UOP greater than or equal to 3180H
TOB greater than or equal to 3180H (varies dynamically)
TOB less than or equal to UOP
TOB less than or equal to 3980H
TOM = 32K, 48K or 62K (less 320 bytes for Monitor work area)

ISIS-II reserves interrupts 0 through 2 for use by itself and the Monitor, and leaves interrupts 3 through 7 available to the user.

The buffer area contains three or more I/O buffers of 128 bytes each (the permanent buffer is the line edited buffer for the Console). Buffers are dynamically allocated and deallocated according to the I/O needs of the user program (see OPEN, CLOSE, LOAD, DELETE, RENAME, and ATTRIB, Sections 3.8.1 through 3.8.11); allocation of buffers may cause the Buffer Area to grow at the expense of the Vacant Area, thus causing TOB to vary dynamically.

The number of buffers required by a user program can vary from a minimum of 3 to a maximum of 19. The following rules can be used to determine the required number of buffers:

1. Each open disk file requires two buffers until the file is closed.

2. An open line-edited file including :CI: requires one buffer until the file is closed. For a disk file, this buffer is in addition to the two required in rule 1.

3. A system call that accesses a disk directory (LOAD, DELETE, RENAME, ATTRIB, CONSOL when it specifies a disk file) requires two buffers during the processing of the call. The buffers are released on return to the calling program.

4. When the CONSOL system call assigns the console input or output device to a disk file, three buffers are required for the console input file and two buffers are required for the console output file. These buffers are required until end-of-file. A program called by a system command in a SUBMIT file must also allow for these buffers in determining its origin point.

User programs run in the user area. The user origin point (UOP) is specified by the user (via the translators, LOCATE, HEXOBJ, etc) and must be at least as large as BOP+180H. The higher the user program is origined, the more buffers ISIS-II may allocate for the user's benefit (up to the maximum of 19). Roughly speaking, more buffers allow more simultaneously open disk files. If the value of TOM is of interest to the user program, it may be obtained by a call on the MEMCK routine in the Monitor (see Intellec MDS Operator's Manual or Intellec Series 2 Model 210 Users Guide).)

Note: the lower 32K of memory must be RAM. The remaining 30K of address space may be occupied by any combination of memories and/or non memory. The last 2K of 64K address space is occupied by the Monitor ROM. The Monitor Work Area is located at the highest 320 bytes of contiguous RAM.

3.8 System Calls

A system call is a subroutine call: the call is in the user program, the subroutine is within ISIS-II (KERNEL). The ISIS-II subroutine will use its own stack as necessary; thus the depth of the user stack is not affected by subroutine calls within ISIS-II.

The following subsections demonstrate the various system calls in PL/M schemata, and define the action of the subroutines. (The interface to assembly language user programs is given in Appendix B.)

The schemata share the following conventions:

1. Every variable in a system call is of type ADDRESS (never of type BYTE).

2. Errors in hardware or in user programs, or certain hardware- or software-imposed limitations, may prevent the successful completion of system calls. These situations are identified by "error numbers," (listed in Appendix C.). Errors are classed as non-fatal or fatal. Fatal errors are handled as in Section 4.3, non-fatal errors are described below.

   Most system calls specify the address of a variable into which ISIS-II will place an error number if control returns to the user program. This variable is denoted by "STATUS" in the schemata below. If STATUS=0 on return, no error was encountered.

   The following terminology is used throughout Sections 3.8n; (a) "Non-fatal error occurs": no action was performed by the system call; control returns to the user program; the nature of the difficulty is indicated by the value of STATUS on return, which is an "error number." (b) "Fatal error issues:" a message is printed on the cold start Console device, and control passes to the Monitor or to CLI, as described in Section 4.3.

   If a hardware error or a user programming error prevents successful output of an error message to the cold start Console device, the system may hang without the normal error message, in which case the system must be restarted (see Section 4.1.1).

3. A pathname is specified by a PL/M variable which points to the first of a string of bytes in memory.

   This byte string must conform to the syntax of pathname as given in Section 3.6, but may have leading ASCII space characters, and must be terminated by a character which is neither a <c> nor ':' nor '.'.

If the byte string does not conform to this description, a fatal error may issue or a non-fatal error may occur, depending on the system call.

4. To clarify the effect of certain system calls upon files, we can imagine that there are 2 integer quantities, LENGTH and MARKER, associated with each file.

   The LENGTH associated with a file is the number of bytes in the file. For many input files, such as sequences of bytes being read from a teletype keyboard, the LENGTH is potentially infinite; for other input files, such as paper tape, the LENGTH is unknown until the end of the tape is read in. The LENGTH of an output file typically increases as it is written.

   MARKER is associated only with open files (see OPEN system call), and is the number of bytes in the file which precede the byte to be read or written next. The range of MARKER is between 0 and LENGTH, inclusive.

3.8.1 Open

```
CALL OPEN(.AFTN,FILE$POINTER,ACCESS,ECHOAFTN,.STATUS);
```

The OPEN system call establishes a connection between a user program and a file. No input from or output to a file may occur until such a connection is established. A file for which such a connection is established is said to be an "open file." Two files, :CI:, :CO: are always open.

When a file is opened via the OPEN system call, ISIS-II returns an integer value between 0 and 255 inclusive in "AFTN" (Active File Table Number). This value is used in future system calls to specify an open file. (The values 0 and 1 are used in such system calls to specify :CO:, and :CI:, respectively.)

If the specified file is already open, a non-fatal error occurs (unless the file is :CI: or :CO:, in which case the appropriate value is returned in AFTN.) An attempt to open a device currently serving as a Console file will cause a non-fatal error to occur. In both these non-fatal cases, the value used to identify the already open file is returned in AFTN.

No more than 6 files (exclusive of :CI: and :CO:) may be simultaneously open; an attempt to open more than this number of files simultaneously causes a non-fatal error to occur. FILE$POINTER is the address of the first of a string of bytes which satisfy the restrictions listed under point 3 in Section 3.8.

The ACCESS parameter has value 1, 2 or 3. (Otherwise a non-fatal error occurs.)

If ACCESS=1, the file is being opened for input. The associated MARKER is set to 0, LENGTH is unchanged. No attributes (see Section 3.10) of the file are changed. If FILE$POINTER specifies a non-existent file, a non-fatal error occurs.

If ACCESS=2, the file is being opened for output. If the file specified is a non-existent disk file, then a disk file so specified is created, with all attributes (see Section 3.10) reset. If the file specified is an existing disk file, then the Format and Write-Protect attributes must be in a reset state (otherwise a non-fatal error occurs). In either case the associated MARKER and LENGTH are both set to 0.

If ACCESS=3, the file is being opened for update (reading and/or writing, which may be interleaved). If the file already exists, LENGTH is unchanged; if the file does not exist, then a new file is created (as above) and LENGTH is set to 0. In either case, MARKER is set to 0. If the file specified is not a disk file with Format and Write-Protect attributes in a reset state, a non-fatal error occurs.

If the hardware characteristics of the device being opened (see Appendix D) are not compatible with the access modes specified by ACCESS, a non-fatal error occurs.

If the file is not to be opened as a lined file, then ECHOFTN must be 0; otherwise the lower byte of ECHOFTN must be the AFTN of a file already open for output, specifying the associated echo file. For example, AFTN = 0FF00H specifies the console output to be an echo file. Echoes will be interleaved with user's output to the file, if any. If a non-zero ECHOAFTN does not specify a file opened in write mode, a non-fatal error occurs.

Opening disk files causes 2 buffers to be allocated within the Buffer Area (see Section 3.7). If the file is opened as a lined file an additional buffer is allocated to it. If the Buffer Area and the Vacant Area together contain insufficient space for such buffers, a fatal error occurs.

If the file opened is a paper tape punch (:HP: or :TP:), then 12 inches of leader (ASCII null characters) are punched.

3.8.2 Read

```
CALL READ (AFTN,.BUFFER,COUNT,.ACTUAL,.STATUS);
```

This call transfers information from the open input or update

file identified by AFTN to the RAM area named BUFFER. ACTUAL is set to the number of bytes transferred, and MARKER is incremented by the same number (if the file is line-edited, MARKER is updated by the appropriate number). No more than COUNT bytes will be transferred. If ACTUAL=0 on return, then no bytes were transferred.

For all files, either COUNT or (LENGTH minus MARKER) bytes will be transferred, whichever is fewer. For lined files, there is the additional proviso than no bytes beyond the current logical line will be transferred.

If AFTN does not specify a file open for input or update, a non-fatal error occurs.

A fatal error will be issued if a read of a disk file cannot be achieved. READs of other devices cause no errors, but may (in the event of hardware difficulties) hang the system.

If COUNT = 0, then ACTUAL = 0 may or may not indicate end-of-file. End-of-file is best indicated, in the case of line-edited files and COUNT greater than 0, by ACTUAL = 0; in the case of lined files and COUNT greater than 0, it is indicated by ACTUAL less than COUNT.

3.8.3 Write

```
CALL WRITE(AFTN,.BUFFER,COUNT,.STATUS);
```

This call transfers information from the memory area addressed by .BUFFER to the open file identified by AFTN. Exactly COUNT contiguous bytes are transferred. MARKER is incremented by COUNT; if this causes MARKER to be greater than LENGTH, then LENGTH is set equal to MARKER.

If AFTN does not specify a file open for output or update, a non-fatal error occurs.

If disk hardware does not permit a successful write, a fatal error is issued. If other hardware does not permit a successful write, the system may hang (e.g. line printer), or continue without indicating the failure (e.g. teletype).

3.8.4 Seek

```
CALL SEEK(AFTN,MODE,.BLOCKNO,.BYTENO,.STATUS);
```

If AFTN specifies :BB:, this is a no-op; otherwise AFTN must specify a disk file open in either READ mode (access=1), or UPDATE mode (access=3), in which case the call sets or returns the MARKER value associated with the file.

BLOCKNO and BYTENO are address values which together specify a number, N, of bytes, by the formula N = (128*(BLOCKNO modulo 2**15) + BYTENO).

MODE must have one of the values 0,1,2,3, or 4, otherwise a non-fatal error occurs.

If MODE=0(SEEK Return), then MARKER is not changed; instead, values are returned in BLOCKNO and BYTENO such that N=MARKER.

If MODE=1(SEEK Backward), then the current MARKER value is decremented by N; if this new value of MARKER is negative, MARKER is set to 0, and a non-fatal error occurs.

If MODE=2(SEEK Absolute), then the new MARKER value is set to N. If this new value of MARKER is greater than LENGTH, then sufficient zero-value bytes (ASCII nulls) are appended to the file to make LENGTH=MARKER. If insufficient disk space remains for this extension, a fatal error will be issued, either during execution of the SEEK call, or later when an attempt is made to WRITE into the extended area of the file (which can happen at any time during the life of the file on its diskette/platter).

If MODE=3(SEEK Forward), then the current MARKER value is incremented by N. If this new value of MARKER is greater than LENGTH, then sufficient zero-value bytes (ASCII nulls) are appended to the file to make LENGTH=MARKER. If insufficient disk space remains for this extension, a fatal error will be issued, either during execution of the SEEK call, or later when an attempt is made to WRITE into the extended area of the file (which can happen at any time during the life of the file on its diskette/platter).

If MODE=4(SEEK EOF), then the current MARKER value is set to LENGTH; the values of BLOCKNO and BYTENO are ignored.

It should be observed that the editing and buffering implicit in the handling of lined files has the side effect that the current MARKER value is not always calculable by the READing program; thus the use of SEEK on lined files can have unexpected effects.

If the file specified by AFTN is not a disk file or :BB:, then a non-fatal error occurs.

Attempts to seek backward past the beginning of a file, or to extend a file opened for input (ACCESS=1, see Section 3.8.1) by seeking past the end of the file, will result in a non-fatal error.

3.8.5 Rescan

```
CALL RESCAN(AFTN,.STATUS);
```

This call affects the READing of the lined file specified by AFTN. If the file is not a lined file, a non-fatal error occurs. The effect is that the MARKER associated with the file is adjusted so that the next byte to be input by a READ command will be the first byte in the logical line from which a byte was last transferred by READ command. If RESCAN is given before any READ is given, it has no effect. This permits the READing of a line to begin anew, after some or all of it has been previously read. A RESCAN call on :BB: is a null operation.

3.8.6 Close

```
CALL CLOSE(AFTN,.STATUS);
```

CLOSE severs the connection established by the OPEN system call. All files should be "closed" when input or output is complete.

Closing a file releases all buffers allocated for it (by OPEN) in the Buffer Area.

If the file closed is a paper tape punch (:HP: or :TP:), 12 inches of trailer (ASCII null characters) are punched.

If AFTN specifies :BB:, :CI: or :CO:, or the file specified by AFTN is not an open file, CLOSE returns with no error (STATUS=0), but the action is a no-op.

3.8.7 Exit

```
CALL EXIT;
```

When a user program wishes to terminate execution, a call to EXIT is used. This call causes all currently open files (except :CI: and :CO:) to be closed and the Command Interpreter (CLI) to be loaded from the booting drive and started. At the conclusion of this operation, the DEBUG TOGGLE is in a reset state. The current Console definition is not changed.

If CLI cannot be loaded (e.g. the boot drive does not contain ISIS.CLI) then control will be passed to the MONITOR.

3.8.8 Load

```
CALL LOAD(FILE$POINTER,BIAS,RETSW,.ENTRY,.STATUS);
```

This call requests ISIS-II to load a portion of RAM as specified by the contents of an ISIS-II Absolute Object Format file specified by FILE$POINTER.

First, ISIS-II rearranges buffers currently allocated in order to make TOB (see Section 3.7) as small as possible.

Then the program (or data) is loaded into memory at addresses calculated by addition (modulo 64K) of BIAS to the load address specified by the input file. FILE$POINTER is the address of the first of a string of bytes which satisfy the restrictions listed under point 3 in Section 3.8. If FILE$POINTER does not correctly specify a file in ISIS-II Absolute Object Format, or if an attempt to load memory (other than interrupt areas 3-7) below TOB (see Section 3.7) is made, then a fatal error will be issued.

RETSW (Return Switch) must have one of 3 values: 0, 1, or 2; otherwise a non-fatal error occurs.

RETSW=0 will cause control to return to the calling program after the memory loading has been accomplished. ENTRY will be set equal to the loaded program's entry point, as given in the input file. The new UOP is the minimum of the old UOP and the lowest address loaded (exclusive of locations 24-63). Certain error conditions (internal error in object file or attempt to load RAM below TOB) will cause a fatal error to be issued.

RETSW=1 will cause control to tranfer to the newly loaded program at its start address entry point. If the loaded program is not a main program, then results are undefined.

RETSW=2 will cause control to transfer to the Monitor after loading with the PC equal to the start address.

The LOAD call can also affect the DEBUG TOGGLE (see Section 4.3). The toggle is unchanged, reset or set as RETSW is 0,1 or 2, respectively. In the event of an error, the toggle is unchanged.

ISIS-II will not permit loading into the "ISIS-II Area" or the "Buffer Area" (see Section 3.7, Memory Layout); however buffers in current use will be relocated as necessary to make the buffer area as small as possible.

Execution of this system call uses 2 transitory buffers. If the Buffer Area and the Vacant Area (see Section 3.7) together contain insufficient space for such buffers, a fatal error will be issued.

3.8.9 Delete

```
CALL DELETE(FILE$POINTER,.STATUS);
```

This call removes the disk file specified by FILE$POINTER from the disk. Disk space allocated to the file is released. If FILE$POINTER specifies a non-existent file, or if the specified file is not a disk file, or has its Write-Protect or Format attribute set (see Section 3.10), or is already open, a non-fatal error occurs. If the specified file is currently serving as :CI: or :CO:, then it is neither closed nor deleted, and a non-fatal error occurs.

Execution of this system call uses 2 transitory buffers. If the Buffer Area and the Vacant Area (see Section 3.7) together contain insufficient space for such buffers, a fatal error will be issued.

3.8.10 Rename

```
CALL RENAME(OLDFILE$POINTER,NEWFILE$POINTER,.STATUS);
```

This call changes the name of a disk file. Both FILE$POINTER's must specify files on the same disk.

If the files specified are not both disk files on the same disk, or if NEW$FILEPOINTER specifies an already existing file, or if OLD$FILEPOINTER specifies a non-existent file or a file whose Write-Protect or Format attribute (see Section 3.10) is set, then a non-fatal error occurs.

Execution of this system call uses 2 transitory buffers. If the Buffer Area and the Vacant Area (see Section 3.7) together contain insufficient space for such buffers, a fatal error will be issued.

3.8.11 Attrib

```
CALL ATTRIB(FILE$POINTER,SWID,VALUE,.STATUS);
```

This call allows the user to set or reset attributes (see Section 3.10) associated with the disk file specified by FILE$POINTER. If the specified file is a non-existent disk file, a non-fatal error occurs.

SWID (SWitch IDentification) must have the value 0, 1, 2 or 3 (else a non-fatal error occurs), which specifies the Invisible, System, Write-Protect or the Format attribute, respectively; this attribute is reset or set as low order bit of VALUE is a 0 or 1, respectively.

Execution of this system call uses 2 transitory buffers. If the Buffer Area and the Vacant Area (see Section 3.7) together contain insufficient space for such buffers, a fatal error will be issued.

3.8.12 Console

```
CALL CONSOL(CI$FILE$POINTER,CO$FILE$POINTER,.STATUS);
```

This call allows independent redefinition of the 2 halves of the virtual Console device. The physical file corresponding to the current Console input (output) device is closed, and the file specified by CI$FILE$POINTER (CO$FILE$POINTER) is opened as the new Console input (output) device. If this file cannot be opened for input (output), for any reason, a fatal error will be issued.

If CI$FILE$POINTER (CO$FILE$POINTER) specifies the logical device :CI: (:CO:), then there is no effect.

3.8.13 Error

```
CALL ERROR(STATUS);
```

This call allows a user program to cause an error message to be printed on the cold start Console (see Section 4.1.1) in the standard system format (see Section 4.3).

The low-order 8 bits of STATUS specify the value of an error number to be printed in the error message.

Authors of user programs are advised to use error numbers in conformance with Appendix C.

3.8.14 Whocon

```
CALL WHOCON(N,.BUFFER);
```

This call allows determination of what physical device is now serving as the current Console input or output device.

BUFFER must be at least 15 bytes long; into it will be placed the ASCII representation of the pathname, end-delimited by an ASCII space, of the device now serving as the input or output Console device, depending on whether the least significant bit of N is a 1 or a 0, respectively. No errors are reported.

3.8.15 Spath

```
CALL SPATH(FILE$POINTER,.ARRAY,.STATUS);
```

This call allows the caller to gain some logical information regarding the file specified by the string to which FILE$POINTER is pointing.

The information is returned in a 12 byte ARRAY in the following format:

```
ARRAY(0) = Device number
ARRAY(1) - ARRAY (6) = Filename
ARRAY(7) - ARRAY (9) = Filename extension
ARRAY(10) = Device Type
ARRAY(11) = Drive Type
```

The logical entities have the following meaning;

- Device Number is a number which specifies the physical peripheral device to which the file is associated. Valid device numbers and their corresponding peripheral devices are as follows:

```
 0, /* DISK DRIVE 0 */
 1, /* DISK DRIVE 1 */
 2, /* DISK DRIVE 2 */
 3, /* DISK DRIVE 3 */
 4, /* DISK DRIVE 4 */
 5, /* DISK DRIVE 5 */
 6, /* TELETYPE INPUT */
 7, /* TELETYPE OUTPUT */
 8, /* CRT INPUT */
 9, /* CRT OUTPUT */
10, /* USER CONSOLE INPUT */
11, /* USER CONSOLE OUTPUT */
12, /* TELETYPE PAPER TAPE READER */
13, /* HIGH SPEED PAPER TAPE READER */
14, /* USER READER 1 */
15, /* USER READER 2 */
16, /* TELETYPE PAPER TAPE PUNCH (TELETYPE) */
17, /* HIGH SPEED PAPER TAPE PUNCH */
18, /* USER PUNCH 1 */
19, /* USER PUNCH 2 */
20, /* LINE PRINTER */
21, /* USER LIST 1 */
22, /* BYTE BUCKET (A PSEUDO INPUT/OUTPUT DEVICE) */
23, /* CONSOLE INPUT */
24, /* CONSOLE OUTPUT */
25, /* DISK DRIVE 6 */
26, /* DISK DRIVE 7 */
27, /* DISK DRIVE 8 */
28, /* DISK DRIVE 9 */
```

- Filename is the ISIS-II filename.

- Filename Extension is the ISIS-II filename extension.

- Device Type is a field which defines the type of peripheral to which the file is associated. The following types (and corresponding Device Type value) have been defined:

  Sequential Input = 0
  Sequential Output = 1
  Sequential Input/Output = 2
  Random Input/Output = 3

- Drive Type is a field which is meaningful only if Device Type = 3. In this case the Drive Type semantic is as follows:

  Controller not Present = 0
  Two boards double density = 1
  Two boards single density = 2
  Integrated single density = 3
  5440 type hard disk = 4

3.9 Disk Format

All floppy disks contain 77 tracks which are divided into sectors of 128 bytes each. A single density disk has 26 sectors per track, a double density disk has 52 sectors per track.

All hard disk platters contain a minimum of 800 tracks (=400 tracks per surface) with 36 sectors per track; each sector contains 128 bytes. The ISIS-II file structure addressing mechanism is predicated upon track and sector variables defined as byte variables. Thus ISIS-II must perform a mapping of the 800 tracks, 36 sectors/track into a logical representation of 200 tracks, 144 sectors/track.

ISIS-II related files (i.e. not counting ASM80, LIB, LINK, LOCATE, PLM80.LIB, FPAL.LIB) occupy approximately 800 sectors on a system disk and 100 sectors on a non-system disk.

In the previous versions of ISIS-II information pertaining to the soft sector format of the tracks on the diskette was carried in the data area of the ISIS.LAB file. The FORMAT cusp would read the information contained within ISIS.LAB on the diskette mounted on drive 0 and format the new diskette accordingly. This is no longer possible since different densities require different interleave factors. The FORMAT and IDISK cusps each contain a table with the appropriate interleave factors for each diskette/platter density, and will

use the contents of this table to correctly format a diskette/platter.

The interleave factors for each diskette/platter will continue to be written into ISIS.LAB to provide media compatibility with earlier releases of the system (especially important in regards to OEM customers).

3.10 Directory Structure

Each diskette/platter contains a single directory which describes the files currently residing on the diskette/platter.

A diskette directory can accomodate 200 files while a platter directory can accomodate 992 files. On a system diskette/platter approximately 20 files are reserved for ISIS-II; there are 6 basic files (ISIS.DIR, ISIS.LAB, ISIS.MAP, ISIS.T0, ISIS.BIN, ISIS.CLI) and 14 ISIS-II cusps (ATTRIB, BINOBJ, COPY, DELETE, DIR, EDIT, FORMAT, HDCOPY, HEXOBJ, IDISK, OBJHEX, RENAME, SUBMIT, SYSTEM.LIB). On a nonsystem diskette/platter 4 files are reserved for ISIS-II : ISIS.DIR, ISIS.LAB, ISIS.MAP, ISIS.T0.

Each directory entry contains 4 attributes associated with the file: the "Invisible" attribute, the "Write Protect" attribute, the "Format" attribute, and the "System" attribute.

These attributes may be set and reset by use of the ATTRIB command (see Section 3.12.4) or ATTRIB call (see Section 3.8.11).

Files with the Invisible attribute set are normally not listed by the DIR command (see Section 3.12.1). All files listed above normally have the Invisible attribute set.

Files with the Write Protect attribute or the Format attribute set may not be opened for output or update, or be deleted or renamed; an attempt to do so will result in a non-fatal error. The Format attribute is set for 6 special ISIS-II files (ISIS.DIR, ISIS.MAP, ISIS.T0, ISIS.LAB, ISIS.BIN, and ISIS.CLI), and should not be changed by the user.

Files with the "System" attribute set are assumed to be an integral part of the ISIS-II system and are handled accordingly. For example a FORMAT operation with the S switch selected will format a system disk; during this process all files which reside on the source disk and have the system attribute are copied to the target disk. The user should be cautioned against the indiscriminate use of such attribute since it may result in much longer execution time for certain functions (such as FORMAT).

3.11 I/O Driver Specifications

In addition to disks, the devices supported by ISIS-II are all those (except the PROM Programmer) supported by the MONITOR. The drivers for all devices (except disks) are contained in the MONITOR ROM provided with the Intellec and their definitive description appears elsewhere (see Intellec MDS Operator's Manual or Intellec Series 2 Model 210 User's Guide).

CAUTION: ISIS-II uses the Monitor's IOSET routine to select non-disk devices; therefore ISIS-II will interfere with the user's attempts to select devices by use of the Monitor's ASSIGN command. Furthermore, user programs which call the Monitor I/O routines directly will possibly cause such I/O to be interleaved with ISIS-II I/O.

The remainder of Section 3.11 is for information only; in the event of conflict with the above 2 manual references, the references are definitive.

The hardware does not allow software to determine if a device is physically present. Thus, for example, output directed to the line printer will be lost if no line printer is on the system.

3.11.1 Teletype I/O Driver

ISIS-II supports communication with a teletype by treating it as 2 separate devices, an input device (:TI:), and an output device (:TO:). These 2 devices define logical I/O files which are distinct, although :TO: may be used as the echo device for :TI: when :TI: is accessed as a lined file.

ISIS-II uses the Monitor I/O system to access the teletype. This I/O is unbuffered; thus the teletype is unresponsive to keyboard typing except when a user program is requesting input via a READ command.

3.11.2 Video Terminal I/O Drivers

The ISIS-II video terminal I/O drivers (:VI: and :VO:), are identical to the teletype drivers (:TI: and :TO:) except for a higher transmission speed.

3.11.3 Paper Tape Punch I/O Drivers

ISIS-II supports 2 separate punch devices, a low speed device located on the teletype (:TP:), and a separate high speed punch peripheral (:HP:).

When either punch is opened for access, 12 inches of leader (ASCII null characters) are produced by ISIS-II. When either punch is closed, ISIS-II produces 12 inches of trailer (ASCII null characters).

3.11.3.1 Low Speed Punch

The ISIS-II low speed punch, :TP:, is integrated with the teletype and has no automatic on/off control. Therefore, the user must manually start the punch when it is opened, and stop it after it is closed. Furthermore, since the teletype print device, :TO:, is actually the same hardware as :TP:, the user should insure, through a combination of program logic and operations procedure, that these 2 devices don't interleave outputs.

3.11.3.2 High Speed Punch

The high speed punch device (:HP:), is driven through the Monitor I/O system.

3.11.4 Paper Tape Reader I/O Drivers

ISIS-II supports 2 paper tape reader devices, a low speed device located on the teletype (:TR:), and a separate high speed reader peripheral (:HR:). The 2 readers are identical in operation; the major difference lies in the speed of the devices. End-of-file on both devices is indicated by absence of data for a period of 250 milliseconds. When this occurs, the user receives a standard-end-of-file indication from ISIS-II (see Section 3.8.2). Note that end of tape and tape jam are treated identically!

3.11.5 Line Printer I/O Driver

ISIS-II supports a line printer (:LP:). When the line printer is opened or closed, a page eject is not generated.

3.11.6 User-Defined Devices

ISIS-II allows users to implement their own I/O drivers, as defined by the Monitor. These devices are referenced in ISIS-II by the device names listed in Appendix D.

3.11.7 Disk Driver

The ISIS-II Disk Driver will support four types of disk controllers. The types of controllers supported will be: two board single density, two board double density, integrated single density, and 5440 hard disk controller. The relationships among I/O ports and present disk controllers is contained in a table called DKCFTB which is updated at the time of system activation. (see Section 3.13).

3.12 System Commands

The general syntax of commands has been given above (see Section 3.6). This section lists the commands which constitute an essential part of the ISIS-II system. The syntax and semantics of these commands is given below. Where non-terminal symbols are not here defined, they may be found in Section 3.6. Whenever a pathname occurs in a command line, it must be followed by a character which is neither a <c> nor ':' nor '.'. One or more ASCII space characters may be used for this purpose.

If invalid pathnames or switches are detected in a command tail, the command processing is halted at that point, and a non-fatal error occurs.

The command language may be extended by the user's providing files in ISIS-II Absolute Object Format; CLI will then recognize the corresponding filenames as commands (more specifically, as <command head>'s) and invoke the corresponding program.

The command language may be tailored to a user's naming preferences by renaming the system command files. For example, DELETE could be renamed to DEL or CAREFL.

3.12.1 DIR Command

The DIR (Directory) command provides the means whereby the user can ascertain what files are on a disk directory. The syntax for the Directory Command is:

```
<command line> := [ DEBUG ] DIR [ command tail ]
<command tail> ::= <parameter list>
<parameter list> ::= <parameter list><parameter> ! <parameter>
<parameter> ::=  FOR <wildcard pathname> !
                 TO <pathname> !
                 <switch list>
<switch list> ::= <switch list> <switch> ! <switch>
<switch> ::=  0!1!2!3!4!5!6!7!8!9!I!F!P!O!T!Z
```

This command causes a list (in ASCII) of the filenames in the specified disk directory to be output. If the listing device is not specified the output will be to :CO:, else it will go to the device, or file, specified in the pathname following the TO keyword. Each entry in the list contains (a) the filename and extension, (b) the number of bytes in the file, (c) the number of disk sectors allocated for representation of the file, and (d) which attributes are set for the file.

The last line of the list will display a count of all sectors used on the diskette/platter as a fraction of the number of total available sectors. Normally, this list does not include files whose "Invisible" attribute in the directory is set. If an "I" (Invisible) switch is given, then the listing will include such files. If an "F" (Fast) switch is given, then the number of bytes, number of sectors, and set attributes are not given in the output, yielding a "faster" listing.

If the "Z" switch is specified, the only information displayed will be a count of all sectors used on the diskette/platter as a fraction of the number of total available sectors.

The "O" (One) and "T" (Two) switches specify whether the display of the directory is to be in a single (one) column format or double (two) column format, respectively. If the directory device corresponds to a floppy diskette, the default is single column. If it corresponds to a hard disk platter, the default is double column.

If a "P" (Pause) switch is given, then the system will pause and output the message

LOAD SOURCE DISK, THEN TYPE (CR)

The user can then load the disk for which a directory listing is required, type carriage return, and the directory contents of that disk will be output to the designated device. Upon

conclusion, the message

LOAD SYSTEM DISK, THEN TYPE (CR)

will be output.

If the FOR construct is followed by a pathname which does not contain any wildcard tokens the DIR cusp will list the file as being present on the disk even if the file has the invisible attribute set to true and the I switch was not specified in the DIR command.

Note that while a wildcard pathname may contain a device specifier, any numeric switch specifying a disk unit number will override the wildcard device unit number. What's more, the rightmost numeric switch found in a command line will override any previous ones. Only one "TO<pathname>" construct may appear in a command line. Only one "FOR<wildcard pathname>" may appear in a command line.

3.12.2 RENAME Command

The RENAME command provides the facility whereby the user can change the filename (and/or extension) of a disk file. The syntax is:

```
<command line> ::= [ DEBUG ] RENAME <command tail>
<command tail> ::= <oldfile> TO <newfile>
<oldfile> ::= <pathname>
<newfile> ::= <pathname>
```

The two pathnames's must specify the same disk. The file specified by <oldfile> has its filename and/or extension changed, so that it is now specified by <newfile>. If <newfile> specifies an already existing file, the message

```
:FX:FILE.EXT ALREADY EXISTS, DELETE?
```

is output on the current Console device (:CO:), and one logical line is input from the current Console device (:CI:).

If this line begins with the ASCII character "Y" or "y", then the RENAME command continues, otherwise no action is performed and control returns to CLI.

If the files specified are not both disk files on the same disk or if oldfile specifies a non-existent file or a file whose Write-Protect or Format attribut is set, then an error message is issued and control returns to CLI.

3.12.3 COPY Command

The COPY command provides the user with the facility to copy a file, or group of files, and to concatenate files. The COPY command can be used with either single disk or multi-disks configuration. The syntax of the command allows the user to specify a file copying operation using the wildcard facility. A file renaming capability using wild cards is also provided. The syntax is:

```
<command line> ::= [ DEBUG ]  COPY <command tail>
<command tail> ::= <source> TO <destination> [<switch list>]
<source> ::= <wildcard pathname> ! <concatenate file list>
<concatenate file list> ::= <concatenate file list>, <pathname>
                          ! <pathname>
<destination> ::= <wildcard pathname>
<switch list> ::= <switch list><switch> ! <switch>
<switch> ::=  U ! S ! N ! P ! Q ! C ! B
```

3.12.3.1 Description of Major Functions Performed

A. Copy with wildcard facility

The COPY command allows the user to specify a group of files to be copied by using the wildcard construct. (A pathname which contains no wildcard tokens is a proper subset of the wildcard name. Therefore copying of a single file is allowed.)

The user can specify his source files with any valid wildcard construct, however, his destination file specification has two choices. First, he can specify only the device. In this case, the filename becomes the same as the source filename, i.e.,

```
COPY  :F1:*.BAZ TO :F3:
```

is identical to:

```
COPY  :F1:*.BAZ TO  F3:*.BAZ
```

This is included for the user's convenience.

The second choice the user has for specifying his destination file is to use a wildcard name with the same mask as the source wildcard name. Having 'the same mask' is defined as follows: for every position in the source wildcard name which contains an *, the corresponding position in the destination wildcard name must contain the same * token; for every position in the destination wildcard name which contains a ?, the corresponding position in the destination wildcard name must contain a wildcard token (?,*). (Note that *.* can be thought as ******.*** for this purpose).

Also, every position in the source wildcard name that does not contain a wildcard token must have a corresponding non-wildcard token in the destination wildcard name. Therefore,

```
COPY  :F2:F??3*.* TO  :F4:A??5*.*
```

is valid, while

```
COPY  :F2:SK?LL TO  :F4:SKILL
```

is not valid.

The wildcard facility has a default scope which covers all non-format files. Accordingly,

```
COPY  :F2:*.* TO  :F1:
```

will copy all non-format files from the disk mounted on drive 2 to the disk mounted on drive 1. There are two switchs (S and N) which may be used to modify the scope of wildcards. The S switch restricts the scope of the wildcards to system files (with the exclusion of ISIS.BIN and ISIS.CLI which are also format files and therefore already expected to be on the disk), while the N switch restricts the scope of the wildcards to non-format and non-system files.

As files are copied the attributes are all set to false in the destination file. The C (copy attributes switch), if specified, will insure that the destination file, or files, will have the same attributes set which were set in the corresponding source file.

As each file is copied the message

```
COPIED  source name  TO  destination name
```

will be output to the :CO: device.

If user wants to reserve the right to decide on a file by file basis whether a file is to be copied, the Q switch affords him such facility. The Q (query) switch will cause COPY to display the message.

```
COPY <source name> TO <destination name>?
```

and expect a 'Y' (or 'y') for a positive response or anything else for a negative response. Only if a positive response is given will the file be copied, else COPY will continue its normal process.

If a destination file already exists on the destination disk, the message

:FX:FILE.EXT ALREADY EXISTS, DELETE?

is output on the current Console device (:CO:), and one logical line is input from the current Console device (:CI:). If this line begins with the ASCII character "Y" or "y", then the COPY command continues, otherwise no action is performed and processing of the next file continues.

A U (Update) switch, if present in the command line, will cause suppression of the above warning message and cause a copy over the file named newfile . In this case, the new file will be opened in update mode (ACCESS = 3, see Section 3.8.1), and thus the length of the new file, if it already exists, will not be decreased.

The switch B (Brief) is used to achieve the same effect as the U switch with the difference that the old destination file is deleted and a new file, with the same name, is created, containing the exact copy of the source file.

The COPY command allows the user to copy one file on a disk to a file on another disk using just one disk drive. To execute this function the user specifies the P (Pause) switch. When this switch is specified the following message is output to the :CO: device

LOAD SOURCE DISK, THEN TYPE (CR)

The user must mount the source disk on the disk drive specified by the source file pathname. When it is necessary to switch disks, the message

LOAD OUTPUT DISK, THEN TYPE (CR)

is output. As each file is copied a message

COPIED source name TO destination name

will be output to the :CO: device. These messages are repeated until all the files are transferred. At that point the message

LOAD SYSTEM DISK, THEN TYPE (CR)

will be output to the :CO: device, and when this is done control will pass to ISIS-II. This switch permits the user of a single drive system to back up files from one disk to another.

If the source name is equal to the destination name the P switch is implied and a single drive copy will be executed.

B. Renaming Files While Copying

The COPY command permits the user to rename the file or group of files being copied with the following syntactical rule covering the use of wildcard tokens: if a wildcard token is used in the source file name it must also be used in the same position in the destination file name. Both explicitly named portions of the source and destination file names must have the same length.

```
COPY  :F2:FOO?? TO  :F1:FXX??
```

is allowed, while

```
COPY  :F2:FOO* TO :F1:FXX?
```

and

```
COPY  :F2:FOO? TO :F1:F??X
```

are not.

C. File Concatenation

The COPY command allows the user to concatenate files together and create a new file.

This facility entails certain restrictions. The destination filename must not be the same as any of the filenames to be concatenated. Also, none of the source files, or the destination file can contain wildcard tokens.

The concatenation facility of COPY has its own operator, the ','. If there is a ',' in the command line, COPY will automatically assume that there is to be concatenation. The error checking for concatenate will be done before any of the disk operations.

Example:

```
COPY A,B,C TO D
```

will result in the concatenation of files A, B, C into one file called D. As each file is concatenated to file D, the message

```
APPENDED source filename TO destination filename
```

will displayed on :CO:.

3.12.4 ATTRIB Command

This command allows the user to examine, set, or reset attributes of a disk file. The syntax is:

```
<command line> ::= [ DEBUG ] ATTRIB <command tail>
<command tail> ::= <wildcard pathname> [ <switch list>]
<switch list> ::= <switch list> <switch element> !
                  <switch element>
<switch element> ::=  W0 ! W1 ! I0 ! I1 ! S0 ! S1 ! F0 ! F1 ! Q
```

W1 sets the write protect attribute;
W0 resets it.
I1 sets the invisible attribute;
I0 resets it.
S1 sets the system attribute;
S0 resets it.
F1 sets the format attribute;
F0 resets it.

If the Q (query) switch is entered, ATTRIB will display the message

```
:FX:FILE.EXT, MODIFY ATTRIBUTES?
```

on :CO: and expects a response on :CI:. A response of 'Y' or 'y' will cause ATTRIB to modify the attributes of the file in question. The filename along with the attributes will be displayed on the :CO: device. Any other response causes ATTRIB to leave the file's attributes unmodified and continue.

If pathname does not specify a disk file, or specifies a non-existent disk file, an error message will be issued.

If <switch list> specifies different values for the same attribute, the value rightmost in the command line takes precedence.

If <switch list> is not specified, then the filename will be displayed along with those attributes which are set. This allows the user to determine the present attributes of a file.

3.12.5 DELETE Command

The DELETE Command provides the facility to remove files from a disk, thereby freeing disk space for allocation to other files. The syntax is:

```
<command line> ::= [ DEBUG ] DELETE <command tail>
<command tail> ::= <file spec list> [ switch ]
<file spec list> ::= <file spec list> , <file spec> !
                     <file spec>
<file spec> ::= <wildcard pathname> [ Q ]
<switch> ::=  P
```

If a pathname specifies a device other than disk, or if it specifies non-existent disk file or a file whose Write-Protect or Format attribute is set, then the file is not deleted and an informative comment is printed; processing of the file spec list continues.

The Q (query) switch causes DELETE to display the message

```
:FX:FILE.EXT, DELETE?
```

and expects a 'Y' or 'y' for a positive response and anything else for a negative response.

The P (pause) switch allows deleting a file on another disk while using just one drive. When this switch is specified the following message is output to the :CO: device

```
LOAD SOURCE DISK, THEN TYPE (CR)
```

The user must load the disk containing the file(s) to be deleted and then type a carriage return. As each file is deleted, the message

```
:FX:FILE.EXT, DELETED
```

will be output to the console. When done, the following message will be output to the :CO: device

```
LOAD SYSTEM DISK, THEN TYPE (CR)
```

3.12.6 FORMAT Command

The FORMAT command allows the user to format a disk, so that it may be used by ISIS-II. The syntax is:

```
<command line> ::= [ DEBUG ] FORMAT <command tail>
<command tail> ::= <target device> <fid> [ <switch list>]
<target device> ::=  :F0: ! :F1: ! :F2: ! :F3: ! :F4: !
                     :F5: ! :F6: ! :F7: ! :F8: ! :F9:
<fid> ::= <filename>[ .<extension>]
<switch list> ::= <switch list><switch>  ! <switch>
<switch> ::=  A ! S ! K ! FROM <integer>
<integer> ::=  0 ! 1 ! 2 ! 3 ! 4 ! 5 ! 6 ! 7 ! 8 ! 9
```

The disk mounted on the target device is formatted in a soft-sectored format; four necessary files (ISIS.LAB, ISIS.DIR, ISIS.MAP, and ISIS.T0) are written on it. The label given in the command line is written in ISIS.LAB.

If the S (system) switch is present, all files on the source disk which have the system attribute will be copied to the target disk; thus, if the user has not tampered with the system attribute, a copy of a system disk will be produced (the source disk must, of course, be a system disk).

If the A (all) switch is present, then all files along with their attributes on the source disk are copied to the target disk, thus building a duplicate disk.

If the K switch is present, the target disk will be initialized using interleave factors which are optimized for operation on an IPB85-based system. The default, i.e. K switch not present, are interleave factors optimized for operation on an IPB80-based system.

> Note: The K switch should not be mentioned in any documentation intended for the outside user until the IPB85-based system is released.

The FROM <integer> switch allows the user to specify the disk drive from which the required initialization files will be taken. If no FROM <integer> is given, the default will be Drive 0. If FROM is not followed by a valid integer (0-9), the message

> UNRECOGNIZED SWITCH

will be displayed on the :CO: device. If the FROM <integer> device corresponds to the target device, the message

> CANNOT FORMAT FROM TARGET DRIVE

will be displayed on the :CO: device.

The FROM <integer> device should correspond to a system disk. If it does not and if the "S" or "A" switch is specified, the effect is to duplicate the ability to backup a nonsystem disk without having to use the "DEBUG FORMAT label A" syntax of ISIS-II V2.2D.

If no disk is mounted in the target drive, an error message is issued and ISIS.CLI is reloaded.

This command cannot be used directly on a single drive system. In such systems IDISK should be used instead.

3.12.7 HEXOBJ Command

This cusp converts hexadecimal files into a corresponding absolute subset of the Object Module Formats. HEXOBJ command syntax is as follows:

```
<command line> ::= [ DEBUG ] HEXOBJ <command tail>
<command tail> ::= <input file> TO <output file> [ <start option> ]
<input file> ::= <pathname>
<output file> ::= <pathname>
<start option> ::=  START (<addr>)
```

HEXOBJ uses the name of the output file, minus extension, for the module name of the output module. START(addr) is used to include a starting address (the address of the first instruction to be executed) in the absolute object module. The address can be specified by a hexadecimal, decimal, octal, or binary number followed by a letter (H, D, O or Q, B, respectively) indicating the base. If no letter is specified, D is assumed. If START(addr) is omitted, the starting address is taken from the end-of-file record of the hexadecimal format file, which is determined by the END assembly language statement or is determined by the PL/M compiler. If no starting address is specified in any of the above ways, zero is assumed; address zero is in the ISIS-II area and will cause an error on any attempt to load the converted file.

3.12.8 OBJHEX Command

OBJHEX is a cusp which converts the absolute subset of the Object Module Formats to hexadecimal. OBJHEX syntax is as follows:

```
<command line> ::= [ DEBUG ] OBJHEX <command tail>
<command tail> ::= <input file> TO <output file>
<input file> ::= <pathname>
<output file> ::= <pathname>
```

3.12.9 BINOBJ Command

BINOBJ is a cusp which converts ISIS V1.0-1.2 fastload format to the absolute subset of the Object Module Formats. BINOBJ syntax is as follows:

```
<command line> ::= [ DEBUG ] BINOBJ <command tail>
<command tail> ::= <input file> TO <output file>
<input file> ::= <pathname>
<output file> ::= <pathname>
```

3.12.10 SUBMIT Command

SUBMIT permits non-interactive execution of an ISIS-II command sequence. The term "command sequence" is defined to be a file consisting of an integral number of ISIS-II command lines. Each command line may be followed by an integral number of data lines, as required by the program invoked by the command line.

By making use of the ISIS-II system calls CONSOL and WHOCON, SUBMIT alters the system console input device (:CI:) to accept input from a user defined disk file containing the command sequence, returning to the previous console device when input is exhausted. SUBMIT accepts as input a command sequence definition file, consisting of a sequence of commands to ISIS-II cusps (possibly with formal parameters), and a list of actual parameters.

A command sequence invocation is somewhat analogous to a procedure Call-Execute-Return sequence. SUBMIT "calls" the user defined command sequence after saving the state of the current command sequence, "passing" parameters as required.

ISIS-II then executes the command sequence. SUBMIT is invoked again at the end of the command sequence to return (restore) to the point of call in the previous command sequence.

SUBMIT command sequences may be nested to any level.

3.12.10.1 Description of All Major Functions Performed

A. Invocation

Invocation is the process of
1. Reading the SUBMIT command line,

2. Opening and/or creating the necessary files,

3. Subsituting actual parameters found in the command line for formal parameters found in the command sequence definition,

4. Saving the current command sequence information (by adding a restoration command to the end of the new command sequence),

5. Changing :CI: to the specified command sequence,

6. Exiting to ISIS-II to actually execute the command sequence.

B. Command Sequence Definition

The input file specified in the SUBMIT command and containing the sequence of ISIS-II commands to be executed is called the

Command Sequence Definition. This file may contain formal parameters. SUBMIT will assume the default extension .CSD if no extension is supplied. The Command Sequence Definition is read and copied to the Command Sequence (extension .CS), with actual parameters substituted for formal ones.

C. Command Sequence

The Command Sequence file, formed by appending the extension .CS to the root filename of the Command Sequence Definition, is the file that will become the console input. This is a temporary file that will be deleted upon restoration.

D. Formal Parameters

SUBMIT allows for up to 10 formal parameters of the format "%n" (the two characters must be immediately adjacent), where n is a digit 0 through 9. These formal parameters may appear anywhere in the Command Sequence Definition.

E. Actual Parameters

Actual parameters are character strings (up to 31 characters), defined by their position in the parameter list (0 being the first parameter) and delimited by comma, right parenthesis, or blank. Actual parameters containing delimiter characters may be entered by embedding the parameter in quotes ('). SUBMIT will allow DLE (Control-P) in parameters, and in its input file, to quote the following character. SUBMIT parameter conventions conform to the Intel Software Standard, section 2.4.1.4. A null actual parameter may be specified by adjacent commas in the parameter list.

F. Restoration

Restoration is the process of undoing a SUBMIT command. A restoration command is a SUBMIT command with the RESTORE control set. The restoration command is placed at the end of a command sequence by SUBMIT. At restoration time, the command sequence is no longer needed and is deleted.

3.12.10.2 Command Language

A. Syntax

```
<command line> ::= [ DEBUG ] SUBMIT <command tail>
<command tail> ::= <invoke> ! <restore>
<invoke> ::= <command file> [ ( <parameter list>) ]
<restore> ::=  RESTORE <command file>
               ( <previous :CI:>[ ,block, byte ])
<command file> ::= {a disk pathname}
<parameter list> ::= <parameter list> , <parameter> ! <parameter>
```

B. Semantics

A command tail may be a single line, its length subject to the restrictions within ISIS-II, namely 122 characters.

<invoke> - SUBMIT is invoked by the user in a manner similar to other ISIS-II cusps. As can be determined from the above syntax, SUBMIT requires as input the name of a disk file to be used as the new console command file, along with an optional list of parameters to be substituted for formal parameters in the command file. <invoke> adds a <restore> command to the end of the <command file> in order to cause :CI: to be restored to its former value.

<restore> - The <restore> command is used to restore :CI: to the state it was in prior to <invoke> . <restore> is generated by SUBMIT, and need never be entered by the user. When the previous :CI: is a disk file, the file must be opened and repositioned by seeking to the line following the SUBMIT command, hence the block and byte parameters.

3.12.10.3 Interaction with ISIS-II Cold Start Console

ISIS-II provides a facility for temporarily suspending input from a console input stream and exchanging it for the cold start Console, and vice versa. If a SUBMIT Command File Definition contains a Control-E character, input is switched to the cold start Console and interactive input is enabled. While input is from the cold start Console, the command sequence file should not be modified (e.g. edited). The SUBMIT command sequence may be restarted by entering a Control-E character from the cold start keyboard.

3.12.10.4 Interaction With ISIS-II Cusps

Any program executing under ISIS-II and receiving its console commands from :CI: may be executed under SUBMIT, given sufficient buffer space (see Section 3.7). Regardless of the number of nested SUBMITs, SUBMIT requires a total of 1 open file, since the console input is a disk file. This overhead must be taken into account when determining the origin point of programs destined to run in a SUBMIT environment.

3.12.10.5 Summary of Normal Use Methodology

The invocation of a SUBMIT command sequence is best described via an example. In this example, it is desired to copy a group of files from one disk to another (in this case, disk 0 to disk 1). This can be accomplished by using a nested SUBMIT.

A. Command File Definition

First Level File, ARCHIV.CSD

```
SUBMIT COPY(FOO)
SUBMIT COPY(BAZ)
```

Second Level File, COPY.CSD

```
ATTRIB  :F1:%0 W0
DELETE  :F1:%0
COPY %0 TO :F1:%0
```

B. Invocation

```
-SUBMIT ARCHIV
```

C. Expansion

```
-SUBMIT ARCHIV
-SUBMIT COPY(FOO)
-ATTRIB  :F1:FOO W0
     FILE       CURRENT ATTRIBUTES
 :F1:FOO
-DELETE :F1:FOO
 :F1:FOO, DELETED
-COPY FOO TO :F1:FOO
COPIED  :F0:FOO TO :F1:FOO
-:F0:SUBMIT RESTORE  :F0:COPY.CS(:F0:ARCHIV.CS,0,17)
-SUBMIT COPY (BAZ)
-ATTRIB  :F1:BAZ W0
     FILE       CURRENT ATTRIBUTES
 :F1:BAZ
-DELETE  :F1:BAZ
 :F1:BAZ DELETED
-COPY BAZ TO :F1:BAZ
COPIED  :F0:BAZ TO  :F1:BAZ
-:F0:SUBMIT RESTORE  :F0:COPY.CS (:F0:ARCHIV.CS,0,35)
-:F0:SUBMIT RESTORE  :F0:ARCHIV.CS(:VI:)
```

3.12.10.6 Error Messages

SUBMIT can produce 3 error messages in addition to those produced by the ISIS-II KERNEL. All are fatal.

```
ILLEGAL SUBMIT PARAMETER
PARAMETER TOO LONG
TOO MANY PARAMETERS
```

The P (pause) switch supported by the COPY, DIR, a
cusps should not be used in a SUBMIT command string

exchanging disks while SUBMIT is being executed could generate errors in the logical structure of one or more disks. Since it is possible to correctly execute SUBMIT with the command string containing the P switch, no error message will be output, but the user should be warned of the possibility of loss of data on one or more disks.

3.12.11 IDISK Command

IDISK is used to initialize a disk. This operation is a subset of the format operation because it does not perform any general file copying operation. It can be used on either a single or a multiple drive system.

IDISK allows the user to create a non-system disk or a basic system disk. If it is used to create a non-system disk, the disk will contain the following files: ISIS.MAP, ISIS.DIR, ISIS.LAB and ISIS.T0. This last file will contain a nonsystem bootstrap. A basic system disk can be produced by using the S switch. If the S switch is used then ISIS.BIN and ISIS.CLI are put on the disk together with the files mentioned above, with the difference that ISIS.T0 will now contain the T0BOOT program.

```
<command line> ::= [ DEBUG ] IDISK <command tail>
<command tail> ::= <target device><fid> [ <switch list> ]
<target device> ::=  :F0: ! :F1: ! :F2: ! :F3: ! :F4: !
                     :F5: ! :F6: ! :F7: ! :F8: ! :F9:
<fid> ::=   <filename>[ .<extension>]
<switch list> ::= <switch list><switch> ! <switch>
<switch> ::=  S ! K !  FROM <integer>
<integer> ::= 0 ! 1 ! 2 ! 3 ! 4 ! 5 ! 6 ! 7 ! 8 ! 9
```

If the K switch is present, the target disk will be initialized by using interleave factors which are optimized for operation on an IPB85-based system. The default, i.e. K switch not present, are interleave factors optimized for operation on an IPB80-based system.

> Note: The K switch should not be mentioned in any documentation intended for the outside user until the IPB85-based system is released.

The FROM <integer> switch allows the user to specify the disk drive from which the required initialization files will be taken. If no FROM <integer> is given, the default will be Drive 0. If FROM is not followed by a valid integer (0-9), the message

UNRECOGNIZED SWITCH

will be displayed on the :CO: device.

When invoked, the IDISK program will determine whether or not it must run in a single drive mode. If the target device and the FROM integer device specify the same drive, then single drive mode will be assumed. In a single drive mode IDISK will issue the following message:

LOAD DISK TO BE FORMATTED, THEN TYPE (CR)

When required the following message will be output at the end of pass one:

LOAD SYSTEM DISK, THEN TYPE (CR)

The message to load disk to be formatted will then be repeated one more time. When the disk initialization is finished IDISK will output the following message:

FORMATTING FINISHED, LOAD SYSTEM DISK, THEN TYPE (CR)

When this is done control is returned to ISIS-II. A system with 32K of memory will require two disk swaps in order to complete the single drive format operation. Systems with 48K or more will require only one swap. Because of this requirement, IDISK is located in such a manner that it cannot be invoked from within a SUBMIT file.

IDISK cannot abort and return control to ISIS-II every time a fatal error condition occurs since IDISK cannot count on the fact that a system disk is present on the system. In this situation control will be passed to the MONITOR.

3.12.12 HDCOPY Command

The HDCOPY command provides a fast physical track-by-track copying capability from one hard disk platter to another. The syntax is:

```
<command line> ::= [ DEBUG ] HDCOPY <command tail>
<command tail> ::= <source> TO <destination> [ V ] !
                   BACKUP [ V ]
<source> ::= <drive number>
<destination> ::= <drive number>
<drive number> ::=  0 ! 1 ! 2 ! 3
```

<source> and <destination> must refer to hard disk platters but they must not refer to the same drive number (i.e. there is no single drive copy capability). If they do refer to the same device number, then a fatal error will result and the message

SOURCE DRIVE CANNOT EQUAL DESTINATION DRIVE

will be output to the :CO: device. If both devices do not correspond to hard disk platters, then a fatal error will result and the message

SPECIFIED DRIVES NOT HARD DISK

will be output to the :CO: device.

The destination disk must have been previously initialized; otherwise a fatal disk error will result.

If the V (verification) switch is specified, the destination platter will be verified by a process of read from the source, write to the destination, read from the destination, compare with the source. If the comparison process reveals a conflict, the message

DRIVE n, LOGICAL TRACK xxx, LOGICAL SECTOR yyy MISCOMPARES

will be output to the :CO: device and processing will continue. If the verification process reveals no miscomparisons, the message

VERIFICATION COMPLETE

will be output to the :CO: device.

If ISIS-II is unable to load CLI upon the completion of HDCOPY (e.g. under the environment of Section 4.1.3 and :F0: was the destination of HDCOPY and :F0: no longer contains system files), then control is passed to the Monitor.

A sample invocation is:

```
-HDCOPY 1 TO 3 V

ISIS-II HARD DISK COPY Vm.n

LOAD DISK(S), THEN TYPE (CR)
:F1:MYDISK.NOW IS SOURCE DISK
:F3:MYDISK.BAK WILL BE OVER-WRITTEN
OK TO CONTINUE?
y
:F1:MYDISK.NOW COPIED TO :F3:MYDISK.NOW
VERIFICATION COMPLETE
HDCOPY COMPLETED
-
```

If the BACKUP switch is specified, the following actions will be taken: drive 1 copied to drive 0, prompt for backup removable platter to be placed in drive 1, drive 0 copied to drive 1, prompt for system removable platter to be placed in drive 1, drive 1 copied to drive 0. If the V (verification) switch is specified, the disk platters will be verified at each step in the process. The BACKUP option applies only to drives 0 and 1 of a hard disk system. If :F0: and :F1: do not correspond to hard disk platters, an appropriate error message will be displayed on the :CO: device.

A sample invocation is:

```
-HDCOPY BACKUP

ISIS-II HARD DISK COPY Vm.n

LOAD DISK, THEN TYPE (CR)
:F1:MYDISK.NOW IS SOURCE DISK
:F0:IS00AT.SYS WILL BE OVER-WRITTEN
OK TO CONTINUE?
y
:F1:MYDISK.NOW COPIED TO :F0:MYDISK.NOW

LOAD BACKUP DISK IN :F1:, THEN TYPE (CR)
:F0:MYDISK.NOW IS SOURCE DISK
:F1:MYDISK.BAK WILL BE OVER-WRITTEN
OK TO CONTINUE?
y
:F0:MYDISK.NOW COPIED TO :F1:MYDISK.NOW

LOAD SYSTEM DISK IN :F1:, THEN TYPE (CR)
:F1:IS00AT.SYS IS SOURCE DISK
:F0:MYDISK.NOW WILL BE OVER-WRITTEN
OK TO CONTINUE?
y
:F1:IS00AT.SYS COPIED TO :F0:IS00AT.SYS
HDCOPY COMPLETED
```

3.13 Supported Configurations

ISIS-II will support the disk configurations shown in Table 3.13. The specific configuration including identification of the system drive will be determined at boot time by T0BOOT (i.e. ISIS.T0); this information will be passed to the ISIS-II KERNEL (i.e. ISIS.BIN) in an array called DKCFTB and an additional byte variable (see Appendix E).

Note: Table 13.3 does not explicitly address the situation in which the integrated drive is a double density drive (this may come about either through the purchase of an MDS701 or through modification of the Model 220). In this case, the integrated double density drive will be referenced as :F0:, :F2:, :F4:, or :F6: depending upon the presence or absence of a hard disk and/or a set of double density drives.

DRIVE NUMBER	HD +DD	HD +SD	HD +ISD	HD +DD +SD	HD +DD +ISD	HD +SD +ISD
:F0:	HD-F	HD-F	HD-F	HD-F	HD-F	HD-F
:F1:	HD-R	HD-R	HD-R	HD-R	HD-R	HD-R
:F2:	(HD)-F	(HD)-F	(HD)-F	(HD)-F	(HD)-F	(HD)-F
:F3:	(HD)-R	(HD)-R	(HD)-R	(HD)-R	(HD)-R	(HD)-R
:F4:	DD	SD	ISD	DD	DD	SD
:F5:	DD	SD		DD	DD	SD
:F6:	(DD)	(SD)		(DD)	(DD)	(SD)
:F7:	(DD)	(SD)		(DD)	(DD)	(SD)
:F8:				SD	ISD	ISD
:F9:				SD		

DRIVE NUMBER	DD	SD	ISD	DD +SD	DD +ISD	SD +ISD
:F0:	DD	SD	ISD	DD	DD	SD
:F1:	DD	SD		DD	DD	SD
:F2:	(DD)	(SD)		(DD)	(DD)	(SD)
:F3:	(DD)	(SD)		(DD)	(DD)	(SD)
:F4:				SD	ISD	ISD
:F5:				SD		
:F6:						
:F7:						
:F8:						
:F9:						

TABLE 13.3 ISIS-II SUPPORTED CONFIGURATIONS

HD = Hard disk,
F = fixed platter of hard disk
R = removable platter of hard disk
DD = Double density floppy
SD = Single density floppy
ISD = Integrated single density floppy
Parentheses () indicate optional drives within the particular configuration.

3.14 T0BOOT

The function of the T0BOOT program (=ISIS.T0) is to (a) determine the system hardware configuration and the system drive, (b) load the ISIS-II KERNEL (=ISIS.BIN) from the system drive, and (c) convey the information found in (a) to the KERNEL.

Upon operator hardware reset, T0BOOT is loaded by the MDS BOOT ROM (or Series II BOOT ROM) from a floppy disk drive. Refer to Table 13.3: For configurations involving only floppy diskettes, this "boot drive" corresponds to :F0:. For configurations involving hard disk and floppy diskettes, this "boot drive" corresponds to :F4:.

T0BOOT may also be loaded via an abort (i.e. fatal error), an Interrupt 1, or a MONITOR G8 command. Under these conditions ISIS-II must already have been loaded into the system.

Once T0BOOT has been loaded and control passed to it, it will proceed to make the assignment of physical devices to logical drives. This order of assignment is based on controller present only (as opposed to controller present and drive ready). The order of assignment is hard disk, followed by double density, then single density, and finally integrated single density. The various physical device to logical drive assignments are shown in Table 13.3. (Note the determination of whether the physical drive device is actually present is not done by T0BOOT but rather is done instead by the ISIS-II disk I/O driver at execution time.)

Next T0BOOT will determine the system drive. The system drive is that drive from which ISIS.T0, ISIS.BIN, and ISIS.CLI are loaded following an abort (=fatal error), a G8, or an Interrupt 1. The system drive is determined by the following algorithm:

1. If the hard disk controller is present and the 0-th hard disk drive is ready, then it is the system drive.

2. Otherwise, if the two board floppy controller at Port 78H is present and the 0-th floppy drive is ready, then it becomes the system drive.

3. Otherwise, the integrated single density becomes the system drive.

Note that at least one of the three steps above must be true since T0BOOT itself had to have been loaded from a disk device. Note also that under the situations where a hard disk controller is present but the 0th hard disk drive is not ready, the system drive is a floppy drive, and so in this case the system drive is not :F0: but rather :F4:.

After the above tasks are accomplished, T0BOOT will load the ISIS-II KERNEL (=ISIS.BIN) from the system drive and will then cause CLI to be loaded (also from the system drive) and control to be passed to it.

T0BOOT conveys information concerning the hardware configuration and the system drive to the ISIS-II KERNEL via a 10 byte array called DKCFTB (Disk Configuration Table) and a byte variable called SYSTEM$DRIVE (see Appendix E).

Error Conditions

1. If the T0BOOT (i.e. ISIS.T0) loaded corresponds to a nonsystem diskette, the message

   NON-SYSTEM DISK, TRY ANOTHER

   will be displayed on the Console and control will pass to the Monitor.

2. If in the course of making its device assignments, T0BOOT finds a non-single density controller at Port 88H, the message

   ILLEGAL DISK DEVICE AT PORT 88H

   is displayed on the Console and control is passed to the Monitor.

3. If in the course of making its device assignments, T0BOOT finds a non-standard 5440 hard disk controller, the message

   ILLEGAL DISK DEVICE AT PORT 68H

   will be displayed on the Console and control will pass to the Monitor.

3.15 Support of Relo Object Files In Load System Call

ISIS-II has internal code for the LOAD system call to implement loading of the absolute subset of the Object Module Formats.

The absolute subset of the Object Module Formats consists of:

1. Module Header Record.
2. Content Record, with absolute segment identifier.
3. Module End Record.

Records with record types numerically greater than or equal to 22H (RELOC) are considered errors and cause loading to be

aborted with a "BAD LOAD FORMAT" fatal error. Record types less than 22H but greater than 6H (CONTENT) are bypassed at load time and have no effect on the loaded image.

4.0 OPERATING PROCEDURE

4.1 ISIS-II Cold Start Procedure

4.1.1 Cold Start in a Floppy Disk Only Environment

In an environment involving floppy disks only (i.e. no hard disks and hard disk controller) the following sequence is employed:

1. Power on the Intellec MDS or Intellec Series 2.

2. Power on the physical disk device corresponding to drive 0.

3. Place a system diskette in drive 0.

4. (Omit this step if Intellec Series 2): Set the BOOT switch to ON.

5. Press RESET.

6. (Omit this step if Intellec Series 2 containing integrated CRT): When Interrupt 2 light comes on, type a space on either the teletype or video terminal keyboard. This determines which device will be the initial system console (i.e. cold start console).

7. (Omit this step if Intellec Series 2): Set BOOT to OFF.

8. The message

   ISIS-II, Vn.m

   will be displayed on the system console.

9. CLI will now prompt with a dash '-'; the system is now ready to accept commands.

4.1.2 Cold Start in a Hard Disk and Floppy Disk Environment (Drive 0 Is System Disk)

In an environment involving hard disk and floppies with a system hard disk platter in Drive 0, the following sequence is employed:

1. Power on the Intellec MDS or Intellec Series 2.

2. Power on the hard disk device corresponding to drive 0 and the floppy disk device corresponding to drive 4.

3. a) Place a system diskette in drive 4.
   b) Place a hard disk system platter in drive 0.
   c) Flip START switch of the hard disk, wait for ready light to come on.

4-9 Same as Steps 4-9 in 4.1.1.

Note that in this environment a system diskette is required in drive 4 whenever a hardware reset is initiated. In all other situations (aborts= fatal errors, Interrupt 1, G8), it is sufficient to have a system platter in drive 0 of the hard disk.

4.1.3 Cold Start in a Hard Disk and Floppy Disk Environment (Drive 0 Not a System Disk)

In an environment involving hard disk and controller (or just the hard disk controller) with floppy disks, the sequence of operations is exactly like that of 4.1.1 except in Step 3 a system diskette should be placed in drive 4 instead of drive 0.

This particular environment will normally be an intermediate one. The typical hard disk and floppy disk environment is that of 4.1.2. Thus in this intermediate environment the user will (after readying the hard disk) initialize and build a system disk platter on drive 0 and then transfer to the 4.1.2 environment by hitting RESET or an Interrupt 1.

4.2 Summary of Normal Use Methodology

ISIS-II is used normally in an interactive mode. The user types commands and, in response, cusps are loaded and executed under the ISIS-II KERNEL. The syntax of commands to the cusps is governed by the cusp's syntax and is covered in section 3.12.

4.3 Summary of Error Conditions

ISIS-II (KERNEL) provides a uniform method of handling error conditions. Errors are either fatal or non-fatal. Every error is designated by an "error number" (see Appendix C). Detection of a non-fatal error results in the appropriate error number being returned to the user program in the STATUS parameter (see Section 3.8).

Detection of a fatal error results in 2 actions:

1. An error message, including the error number and a memory location, is printed on the cold start Console device (as specified at cold-start time, see Section 4.1.1). The message format is

```
ERROR NNN, USER PC XXXX
```

where NNN is one of the error numbers listed in Appendix C, and XXXX is the return address to the user program which made the system call resulting in the error condition. (If NNN specifies the disk IO error (Error 24), then two additional

lines are printed:

```
STATUS = 00YY
DRIVE = ZZ
```

where "YY" represents the error value from the appropriate disk hardware and "ZZ" is the disk drive (See Appendix C).

2. Control returns to CLI or to the Intellec Monitor, as determined by the setting of a system entity called the DEBUG TOGGLE. The value of this toggle is reset by CLI (see Section 3.6), is set by the DEBUG command (see Section 3.12.7), and is modified by the LOAD system call (see Section 3.8.8).

   If the toggle is false (reset), then all open files (including :CI: and :CO:) are closed in their current state, the Console is reopened as the cold start Console (see Section 4.1.1), and CLI prompts for another command.

   If the toggle is true (set), then control passes to the Intellec Monitor, and the user PC value is displayed. A graceful return to ISIS-II may be accomplished if desired by typing the Monitor command "G8".

   User programs may announce error conditions on the cold start Console in conformance with above error message format by use of the ERROR system call (Section 3.8.13).

4.4 Operator Intervention

The operator may, at any time, interrupt ISIS-II execution by depressing Interrupt Switch #1 on the Intellec front panel. This causes all currently open files to be closed in their current state, the Console to be redefined (and reopened) as the device originally specified at cold start time (see Section 4.1.1), the DEBUG TOGGLE to be reset (see Section 4.3), and CLI to be called in to accept a command line.

By depressing Interrupt Switch #0 on the Intellec front panel, the user may transfer control to the Monitor, leaving ISIS-II in a state such that it may be resumed by a "G" command, or terminated by a "G8" command.

APPENDIX A

PL/M EXTERNAL PROCEDURE DECLARATIONS FOR ISIS-II SYSTEM CALLS

These are the PL/M external procedure declarations required in order for a program to make ISIS-II system calls. The public declarations for these calls are contained in the file SYSTEM.LIB.

```
OPEN:
  PROCEDURE (AFTPTR,FILE,ACCESS,MODE,STATUS) EXTERNAL;
    DECLARE (AFTPTR,FILE,ACCESS,MODE,STATUS) ADDRESS;
  END OPEN;

CLOSE:
  PROCEDURE (AFT,STATUS) EXTERNAL;
    DECLARE (AFT,STATUS) ADDRESS;
  END CLOSE;

DELETE:
  PROCEDURE (FILE,STATUS) EXTERNAL;
    DECLARE (FILE,STATUS) ADDRESS;
  END DELETE;

READ:
  PROCEDURE (AFT,BUFFER,COUNT,ACTUAL,STATUS) EXTERNAL;
    DECLARE (AFT,BUFFER,COUNT,ACTUAL,STATUS) ADDRESS;
  END READ;

WRITE:
  PROCEDURE (AFT,BUFFER,COUNT,STATUS) EXTERNAL;
    DECLARE (AFT,BUFFER,COUNT,STATUS) ADDRESS;
  END WRITE;

SEEK:
  PROCEDURE (AFT,BASE,BLOCKNUM,BYTENUM,STATUS) EXTERNAL;
    DECLARE (AFT,BASE,BLOCKNUM,BYTENUM,STATUS) ADDRESS;
  END SEEK;

LOAD:
  PROCEDURE (FILE,BIAS,RETSW,ENTRY,STATUS) EXTERNAL;
    DECLARE (FILE,BIAS,RETSW,ENTRY,STATUS) ADDRESS;
  END LOAD;

RENAME:
  PROCEDURE (OLDFILE,NEWFILE,STATUS) EXTERNAL;
    DECLARE (OLDFILE,NEWFILE,STATUS) ADDRESS;
  END RENAME;

CONSOL:
  PROCEDURE (INFILE,OUTFILE,STATUS) EXTERNAL;
    DECLARE (INFILE,OUTFILE,STATUS) ADDRESS;
  END CONSOL;
```

```
EXIT:
  PROCEDURE EXTERNAL;
  END EXIT;

ATTRIB:
  PROCEDURE (FILE,SWID,VALUE,STATUS) EXTERNAL;
    DECLARE (FILE,SWID,VALUE,STATUS) ADDRESS;
  END ATTRIB;

RESCAN:
  PROCEDURE (AFT,STATUS) EXTERNAL;
    DECLARE (AFT,STATUS) ADDRESS;
  END RESCAN;

ERROR:
  PROCEDURE (ERRNUM) EXTERNAL;
    DECLARE (ERRNUM) ADDRESS;
  END ERROR;

WHOCON:
  PROCEDURE (AFT,BUFFER) EXTERNAL;
    DECLARE (AFT,BUFFER) ADDRESS;
  END WHOCON;

SPATH:
  PROCEDURE (FILE,BUFFER,STATUS) EXTERNAL;
    DECLARE (FILE,BUFFER,STATUS) ADDRESS;
  END SPATH;
```

APPENDIX C

ERROR NUMBERS AND MEANINGS

0	No error detected.
1	Insufficient space in buffer area for a required buffer.
2	AFTN does not specify an open file.
3	Attempt to open more than 6 files simultaneously.
4	Illegal pathname specification.
5	Illegal or unrecognized device specification in pathname.
6	Attempt to write to a file open for input.
7	Operation aborted; insufficient disk space.
8	Attempt to read from a file open for output.
9	No more room in disk directory.
10	Pathnames do not specify the same disk.
11	Cannot rename file; name already in use.
12	Attempt to open a file already open.
13	No such file.
14	Attempt to open for writing or to delete or rename a write-protected file.
15	Attempt to load into ISIS-II area or buffer area.
16	Illegal format record.
17	Attempt to rename/delete a non-disk file.
18	Unrecognized system call.
19	Attempt to seek on a non-disk file.
20	Attempt to seek backward past beginning of a file.
21	Attempt to rescan a non-lined file.
22	Illegal ACCESS parameter to OPEN or access mode impossible for file specified.
23	No filename specified for a disk file.
24	Disk error (see below).
25	Incorrect specification of echo file to OPEN.
26	Incorrect SWID argument in ATTRIB system call.
27	Incorrect MODE argument in SEEK system call.
28	Null file extension.
29	End of file on console input.
30	Drive not ready.
31	Attempted seek on write-only (output) file.
32	Can't delete an open file.
33	Illegal system call parameter.
34	Bad RETSW argument to LOAD.
35	Attempt to extend a file opened for input by seeking past end-of-file.
201	Unrecognized switch.
202	Unrecognized delimiter character.
203	Invalid command syntax.
204	Premature end-of-file.
206	Illegal disk label.
207	No END statement found in input.
208	Checksum Error.
209	Illegal records sequence in object module file.
210	Insufficient memory to complete job.

211 Object module record too long.
212 Bad object module record type.
213 Illegal fixup record specified in object module file.
214 Bad parameter in a SUBMIT file.
215 Argument too long in a SUBMIT invocation.
216 Too many parameters in a SUBMIT invocation.
217 Object module record too short.
218 Illegal object module record format.
219 Phase error in LINK.
220 No end-of-file record in object module file.
221 Segment overflow during Link operation.
222 Unrecognized record in object module file.
223 Fixup record pointer is incorrect.
224 Illegal records sequence in object module file in LINK.
225 Illegal module name specified.
226 Module name exceeds 31 characters.
227 Command syntax requires left parenthesis.
228 Command syntax requires right parenthesis.
229 Unrecognized control specified in command.
230 Duplicate symbol found.
231 File already exists.
232 Unrecognized command.
233 Command syntax requires a "TO" clause.
234 File name illegally duplicated in command.
235 File specified in command is not a library file.
236 More than 249 common segments in input files.
237 Specified common segment not found in object file.
238 Illegal stack content record in object file.
239 No module header in input object file.
240 Program exceeds 64K bytes.

When error number 24 occurs, an additional message is output to the console:

```
STATUS=00nn
DRIVE=mm
```

where nn has the following meanings for floppy disks:

01	Deleted record.
02	Data field CRC error.
03	Invalid address mark.
04	Seek error.
08	Address error.
0A	ID field CRC error.
0E	No address mark.
0F	Incorrect data address mark.
10	Data overrun or data underrun.
20	Attempt to write on Write Protected drive.
40	Drive has indicated a Write error.
80	Operation attempted on drive which is not ready.

For hard disks, nn has the following meanings:

01	ID field miscompare.
02	Data Field CRC error.
04	Seek error.
08	Bad sector address.
0A	ID field CRC error.
0B	Protocol violations.
0C	Bad track address.
0E	No ID address mark or sector not found.
0F	Bad data field address mark.
10	Format error.
20	Attempt to write on Write protected drive.
40	Drive has indicated a Write error.
80	Operation attempted on drive which is not ready.

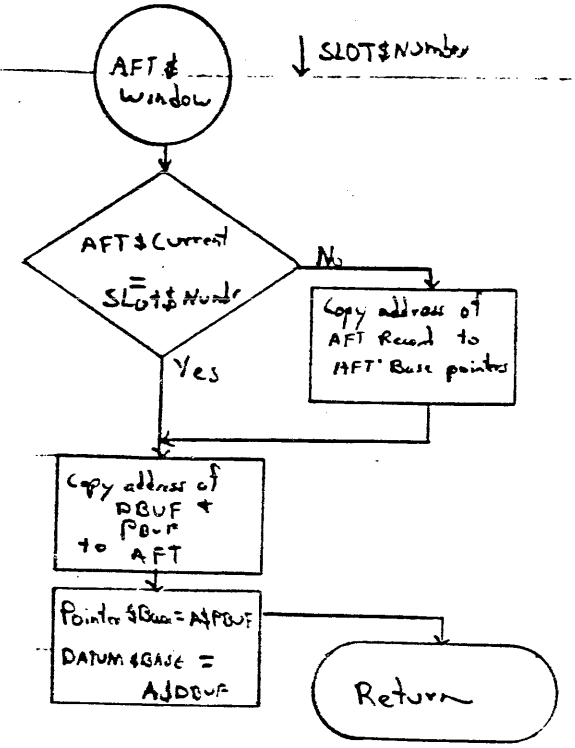
Table 1: Nonfatal Error Numbers Returned by System Calls

OPEN	3, 4, 5, 9, 12, 13, 14, 22, 23, 25, 28.
READ	2, 8.
WRITE	2, 6.
SEEK	2, 19, 20, 27, 31, 35.
RESCAN	2, 21.
CLOSE	2.
DELETE	4, 5, 13, 14, 17, 23, 28, 32.
RENAME	4, 5, 10, 11, 13, 17, 23, 28.
ATTRIB	4, 5, 13, 23, 26, 28.
CONSOL	None; all errors are fatal.
WHOCON	None.
ERROR	None.
LOAD	3, 4, 5, 12, 13, 22, 23, 28, 34.
EXIT	None.
SPATH	4, 5, 23, 28.

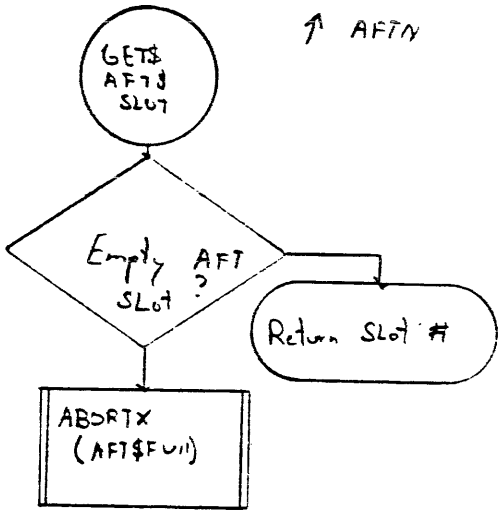
Table 2: Fatal Errors Issued by System Calls

OPEN	1, 7, 24, 30, 33.
READ	24, 30, 33.
WRITE	7, 24, 30, 33.
SEEK	7, 24, 30, 33.
RESCAN	33.
CLOSE	33.
DELETE	1, 24, 30, 33.
RENAME	1, 24, 30, 33.
ATTRIB	1, 24, 30, 33.
CONSOL	1, 4, 5, 12, 13, 14, 22, 23, 24, 28, 30, 33.
WHOCON	33.
ERROR	33.
LOAD	1, 15, 16, 24, 30, 33.
SPATH	33.

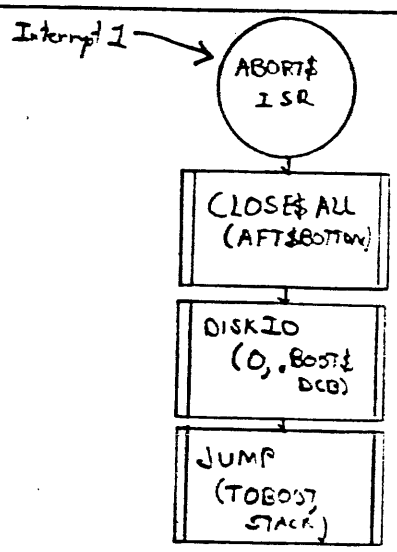
AFT$WINDOW



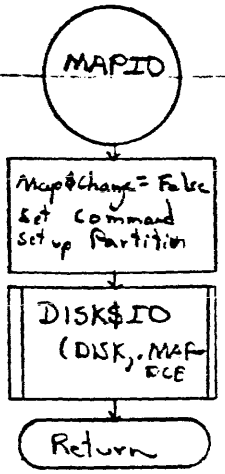
GETAFTSLOT (Function)



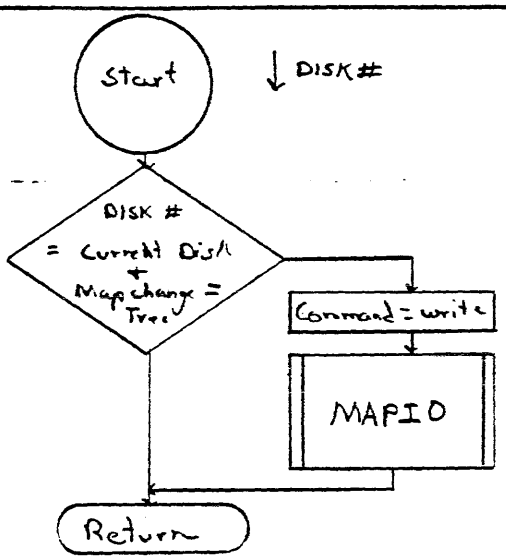
ABORT$ISR



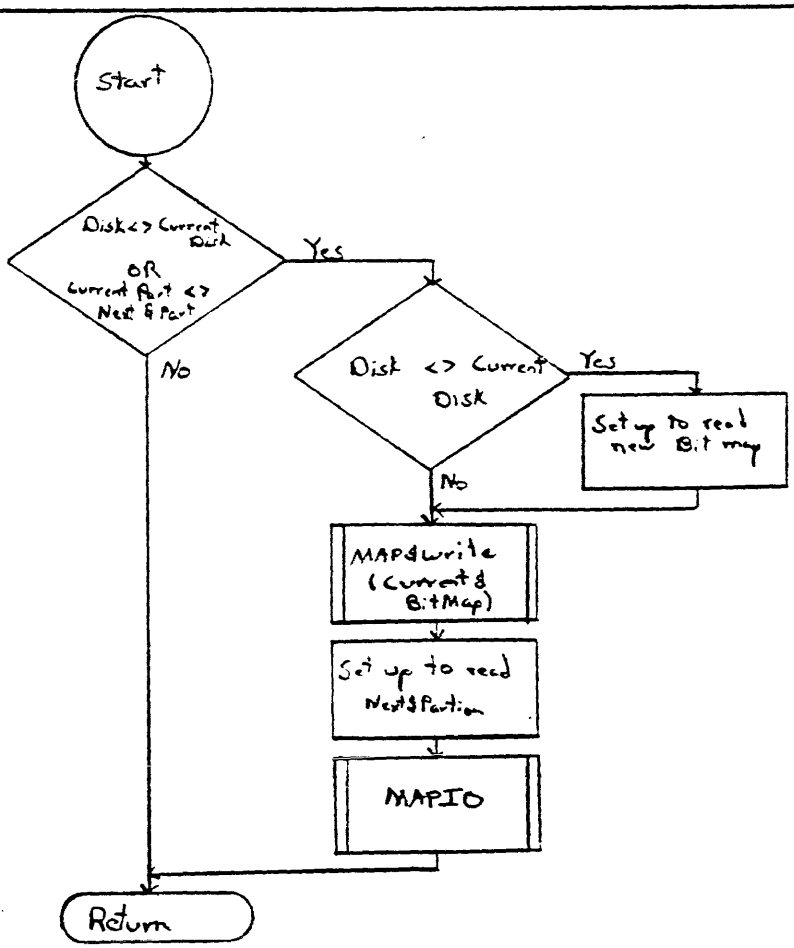
MAPIO

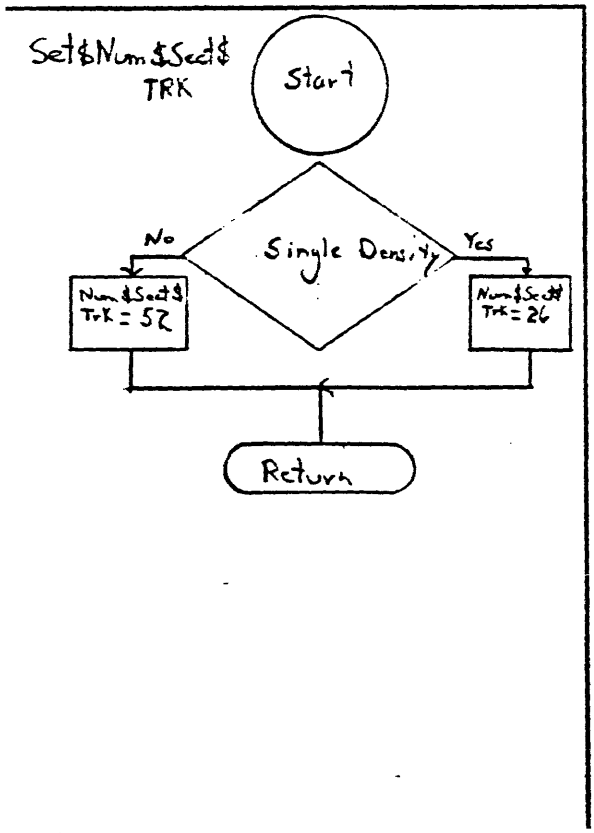


MAP$WRITE

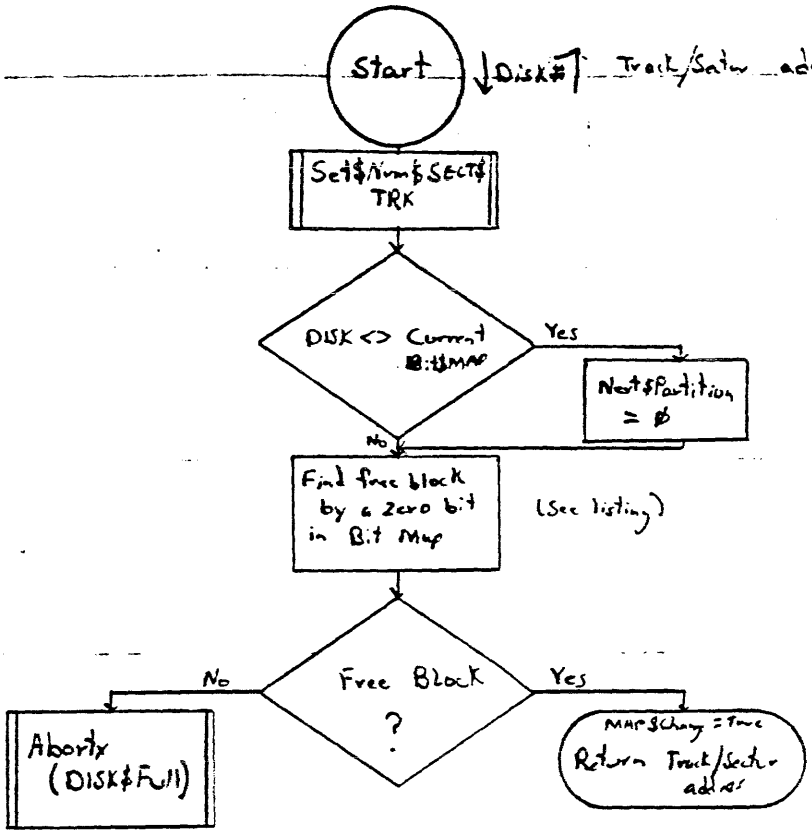


MAP$READ

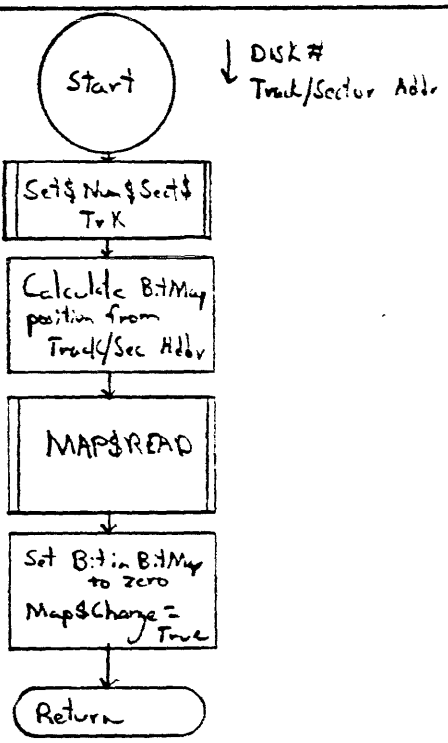




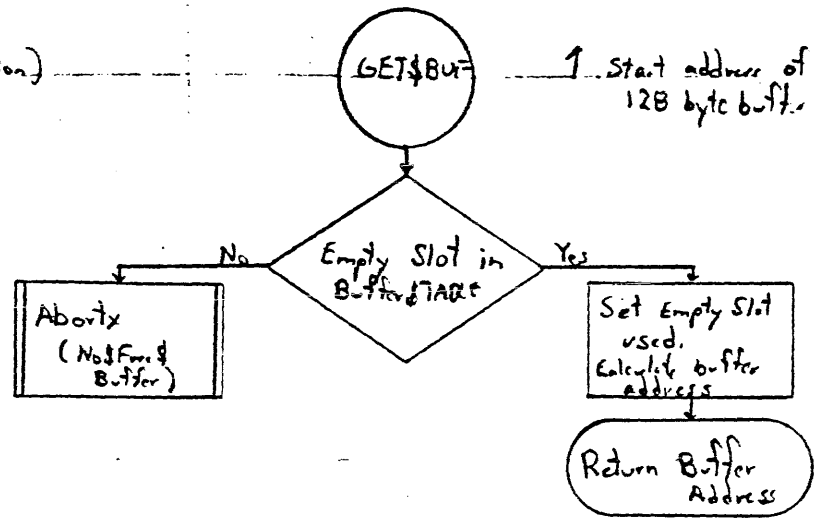
GET$BLOCK



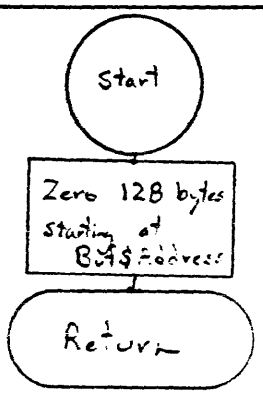
Free$BLocK



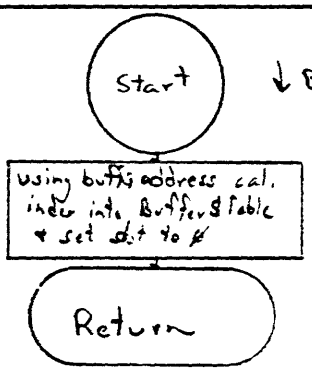
GET$BUF (Function)

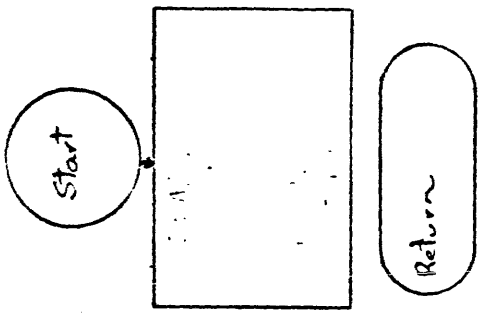


CLear$BUF



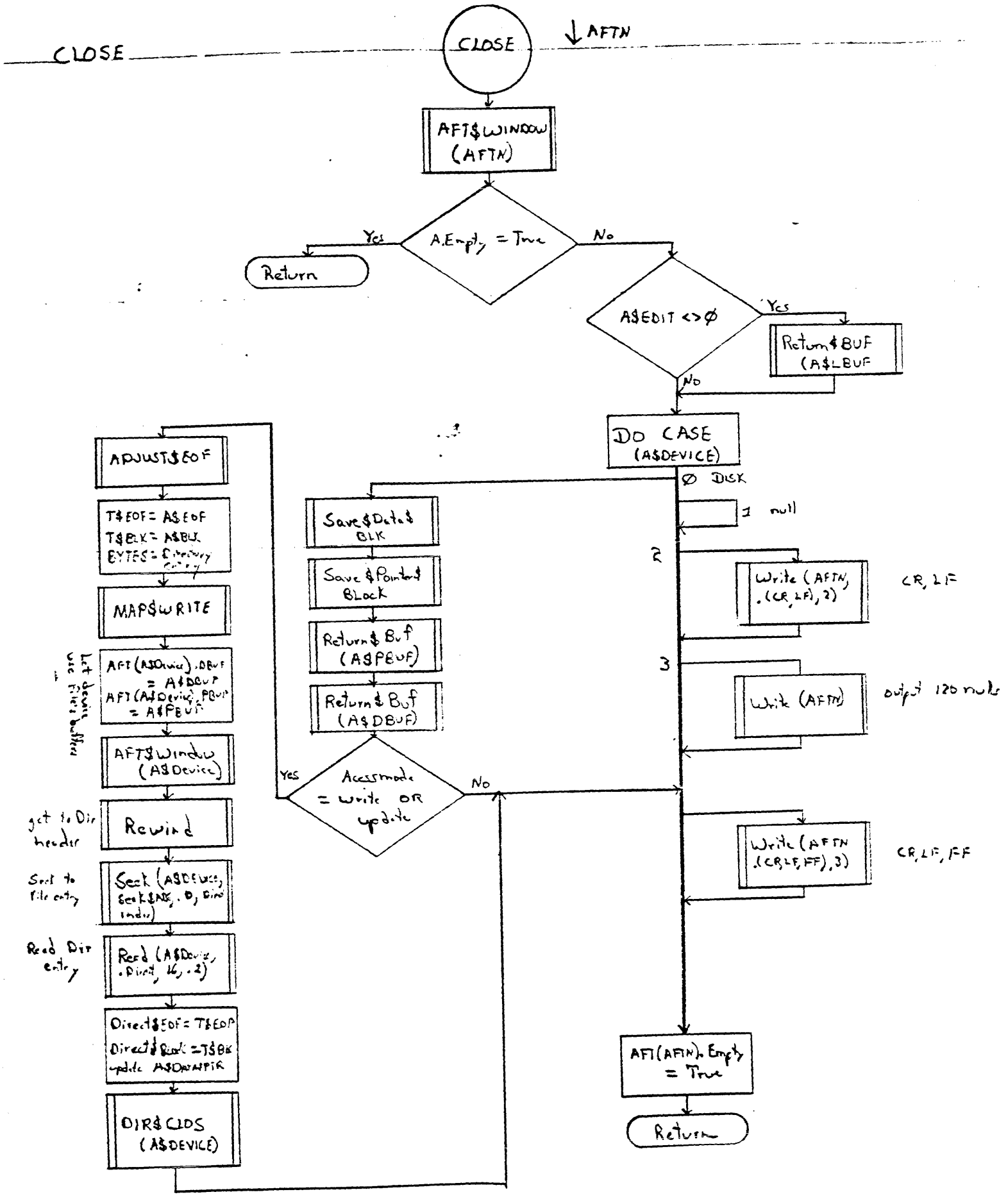
Return$Buf



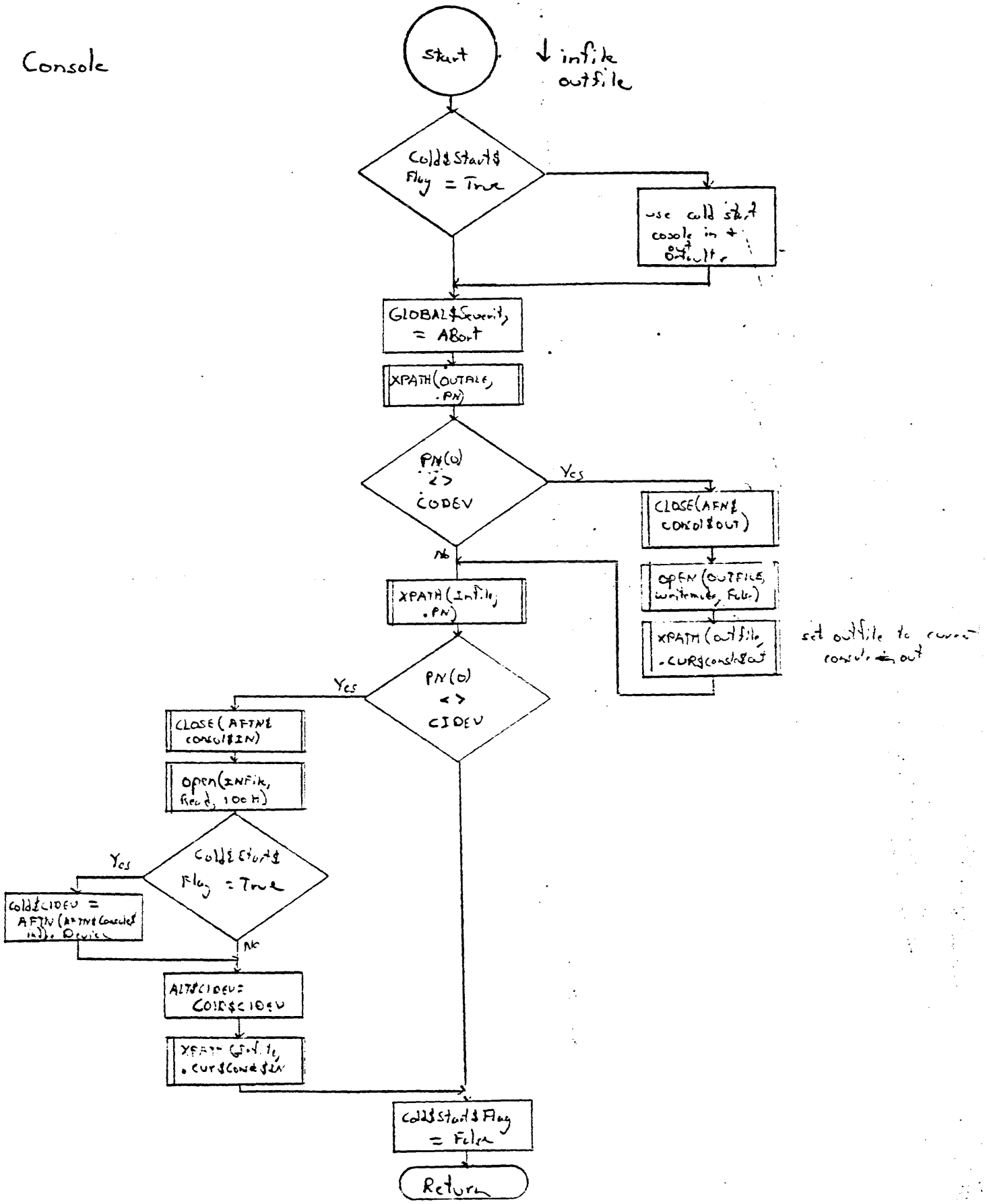


PackAFTBU

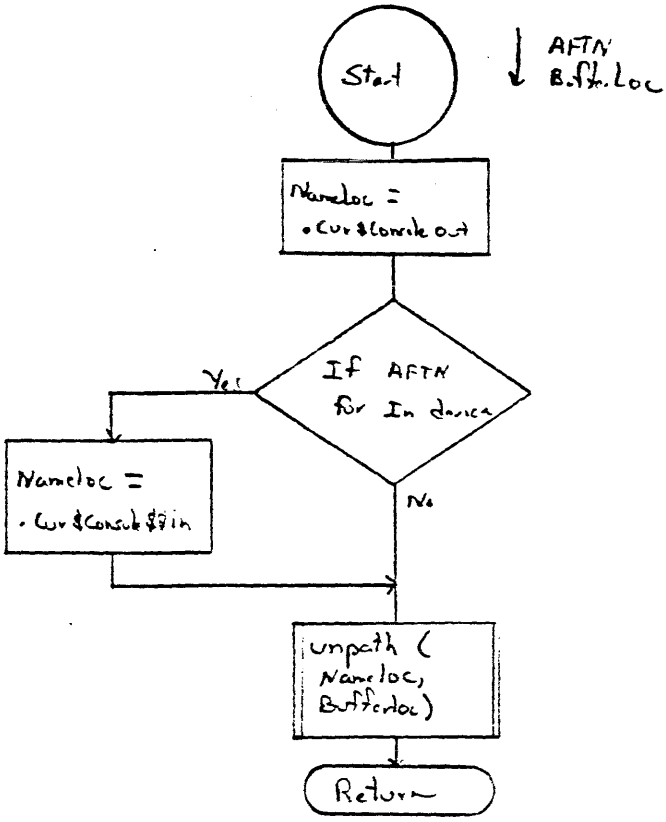
CLOSE


CLOSE
↓AFTN
AFT$WINDOW (AFTN)
A.Empty = True
Yes
Return
No
A$EDIT <> ∅
Yes
Return$BUF (A$LBUF)
No
DO CASE (A$DEVICE)
∅ DISK
1 null
2
Write (AFTN, .(CR,LF), 2)
CR, LF
3
Write (AFTN)
output 120 nuls
Write (AFTN .(CR,LF,FF), 3)
CR, LF, FF
AFT(AFTN).Empty = True
Return
Save$Data$ BLK
Save$Pointer BLock
Return$Buf (A$PBUF)
Return$Buf (A$DBUF)
Accessmode = write OR update
Yes
No
ADJUST$EOF
T$EOF = A$EOF
T$BLK = A$BLK
BYTES = Directory entry
MAP$WRITE
Let pointers use given buffers
AFT(A$Device).DBUF = A$DBUF
AFT(A$Device).PBUF = A$PBUF
AFT$Window (A$Device)
get to Dir header
Rewind
Seek to file entry
Seek (A$Device, Seek$ABS, 0, Dir index)
Read Dir entry
Read (A$Device, .Dirst, 16, .2)
Direct$EOF = T$EOF
Direct$Block = T$BLK
update A$DirAptr
DIR$CLOS (A$DEVICE)

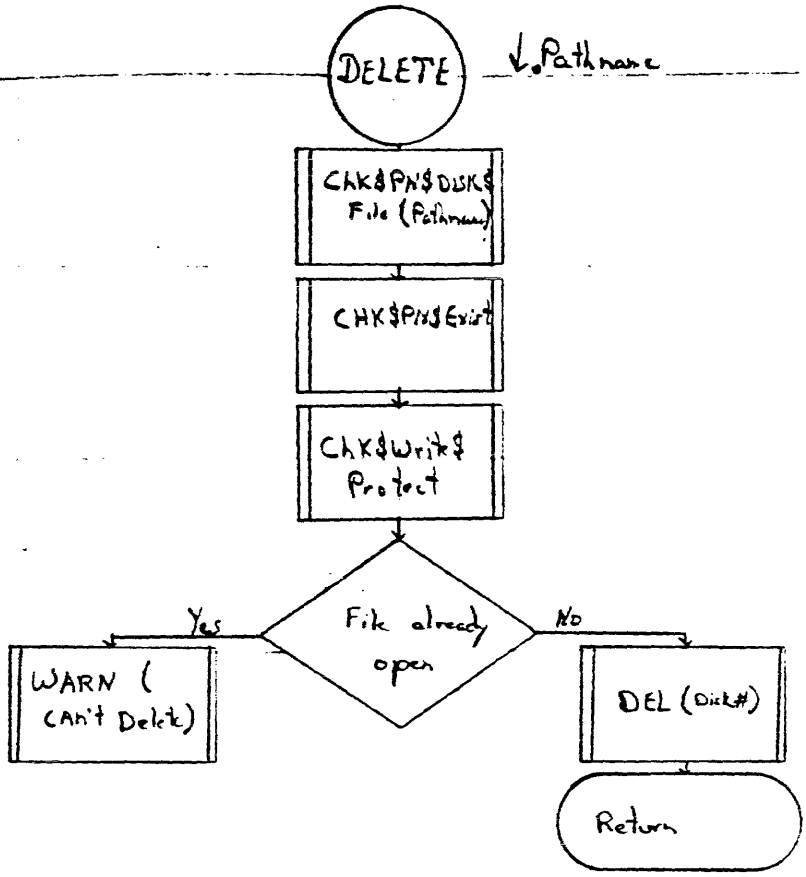
Console


Start
↓ infile
outfile
Cold$Start$ Flag = True
use cold start cosole in + out default
GLOBAL$Source$ = ABort
XPATH(OUTFILE, .PN)
PN(0) <> CODEV
Yes
CLOSE(AFN$ CONSOL$OUT)
OPEN(OUTFILE, write mode, False)
XPATH(outfile, .CUR$console$out
set outfile to current console out
No
XPATH(Infile, .PN)
PN(0) <> CIDEV
Yes
CLOSE(AFN$ CONSOL$IN)
OPEN(INFile, Read, 100H)
Cold$Start$ Flag = True
Yes
Cold$CIDEV = AFN(ALTIN Console Input Device)
No
ALT$CIDEV = CON$CIDEV
XPATH(Cur$Console$In
Cold$Start$ Flag = False
Return

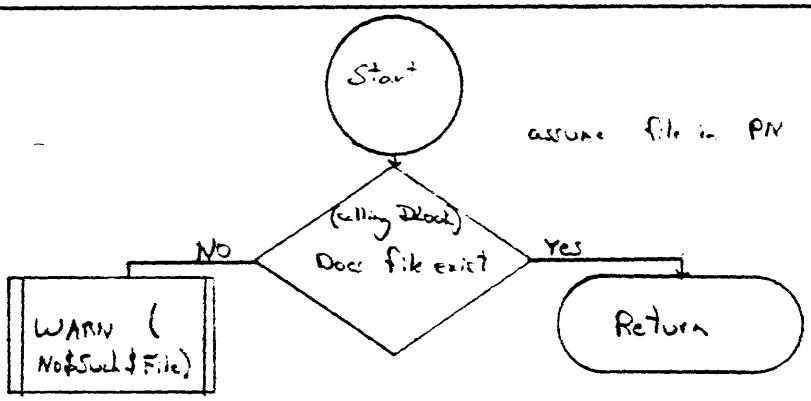
WHOCON


Start
AFTN
B.ffr.Loc
Nameloc =
.Cur$Console out
If AFTN
for In device
Yes
No
Nameloc =
.Cur$Console$Pin
unpath (
Nameloc,
Buffer loc)
Return

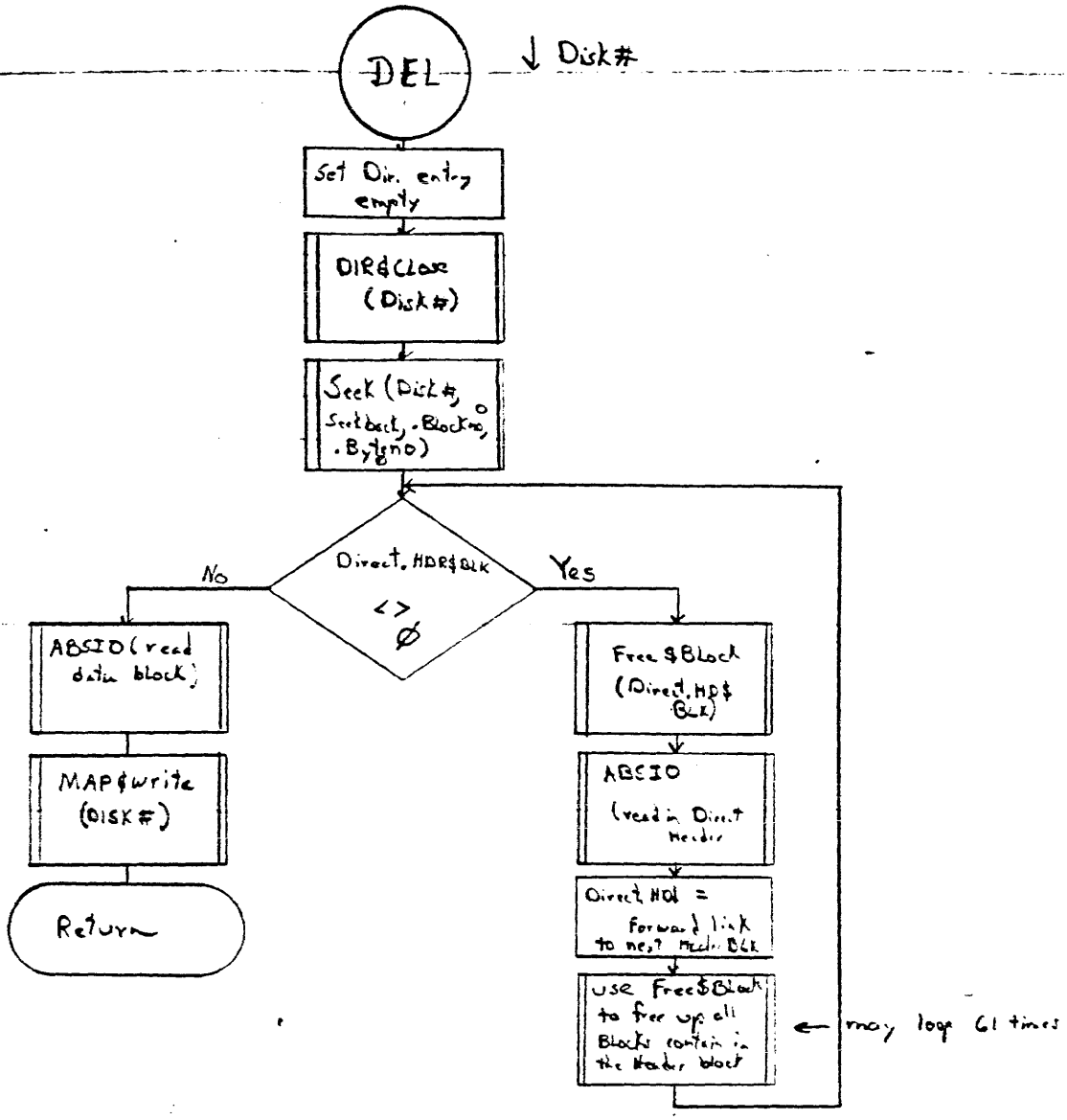
DELETE



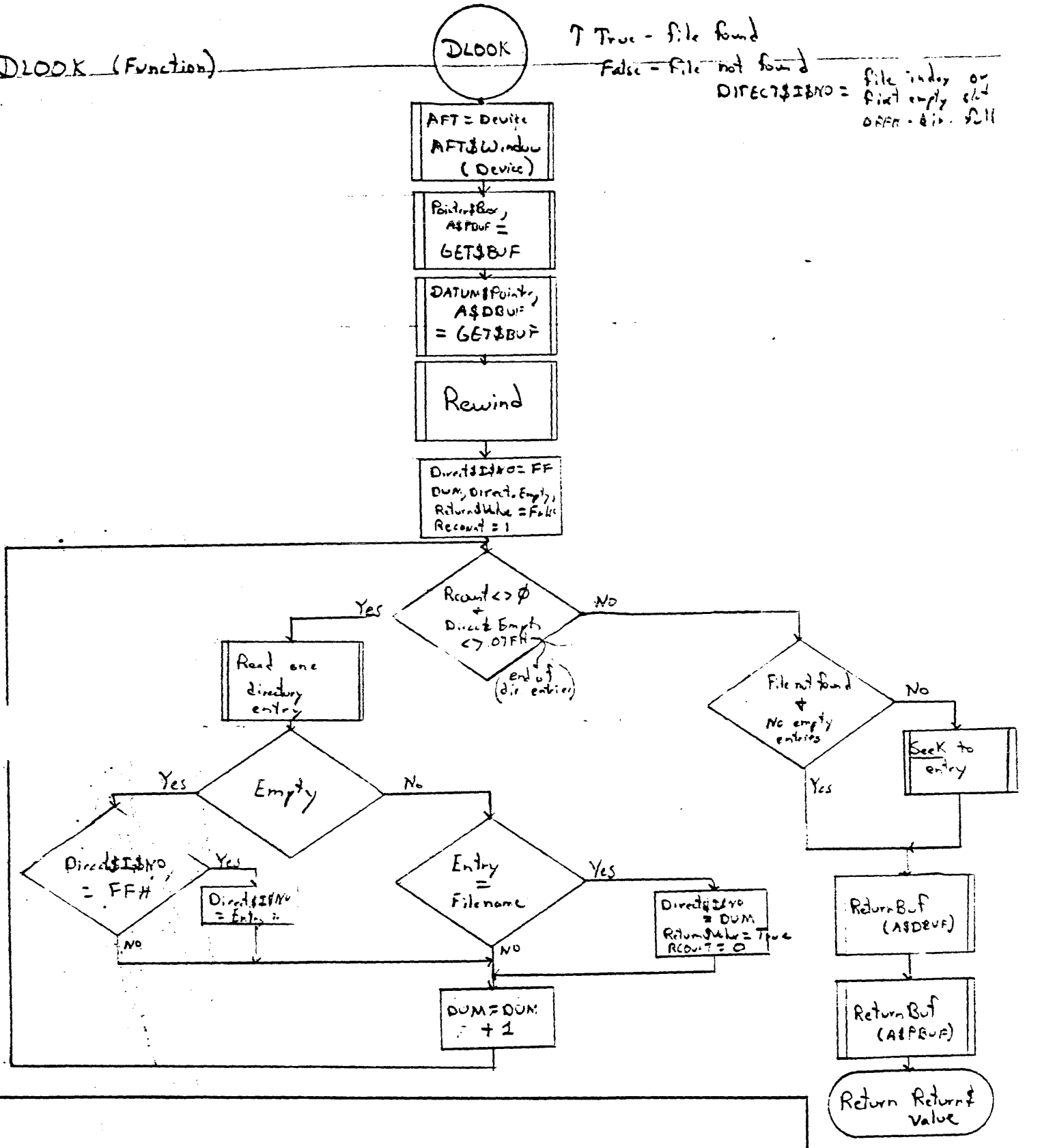
CHKPNEXIST

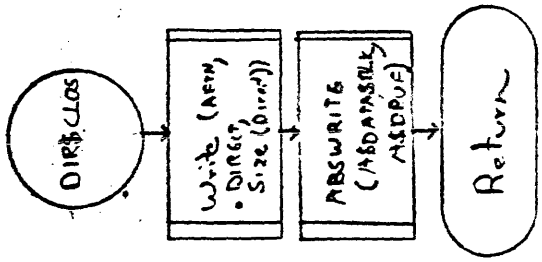


DEL


DEL
↓ Disk#
Set Dir. entry empty
DIR$Close (Disk#)
Seek (Disk#, SectorAdr, .BlockNo, .ByteNo)
Direct. HDR$Blk
<>
∅
No
Yes
ABSIO (read data block)
MAP$write (DISK#)
Return
Free$Block (Direct. HD$ Blk)
ABSIO (read in Direct Header)
Direct HDR = forward link to next Header Blk
use Free$Block to free up all Blocks contain in the Header block
← may loop 61 times

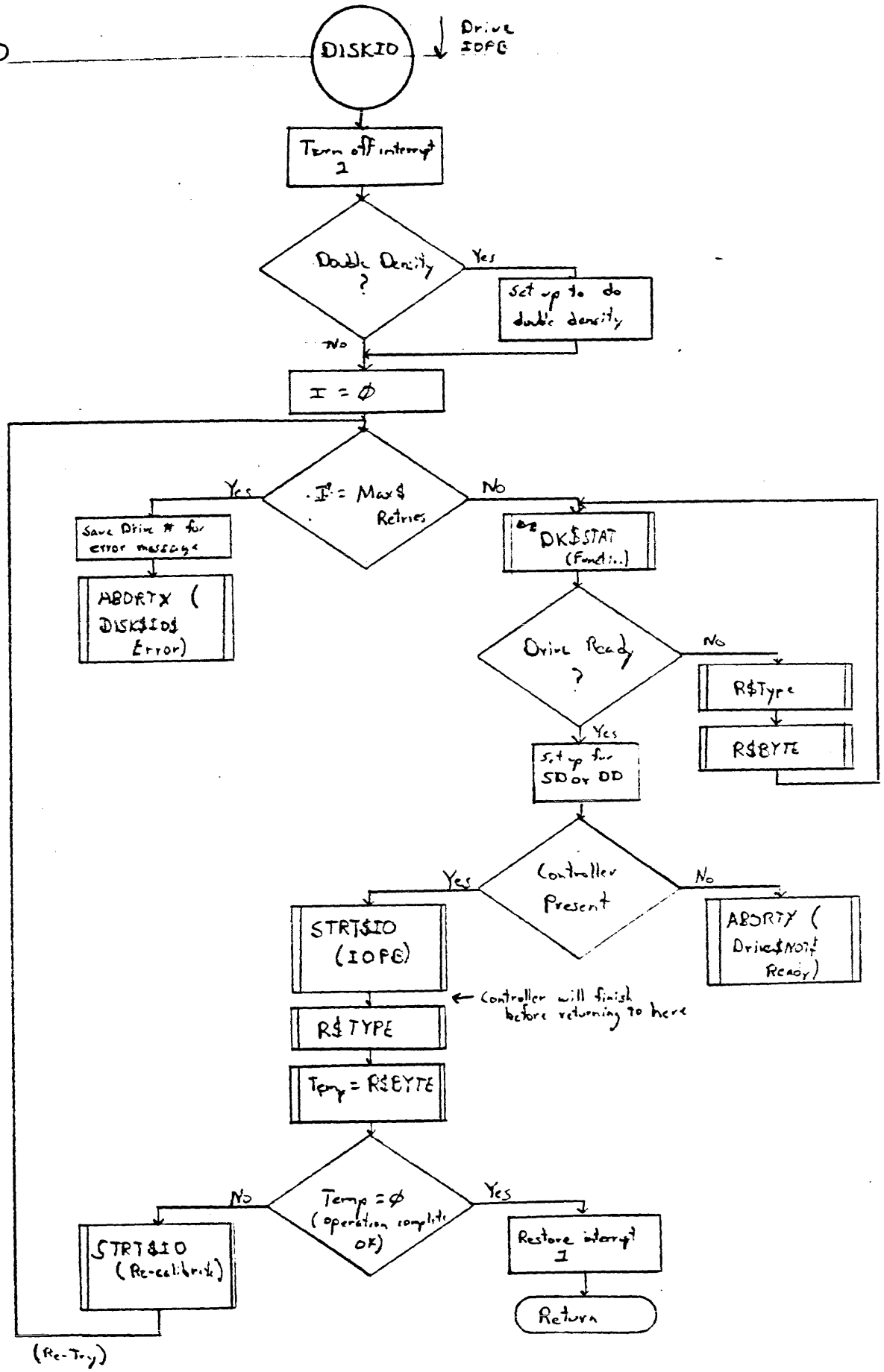
DLOOK (Function)



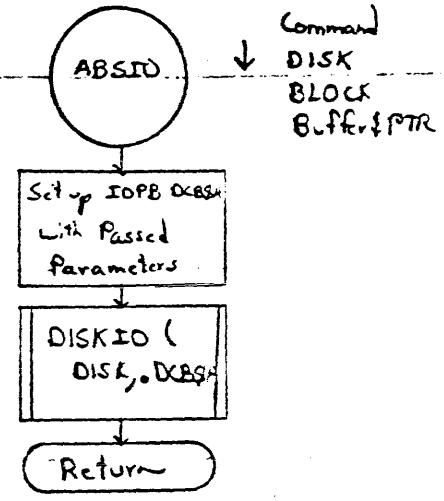


DIR$CLOS

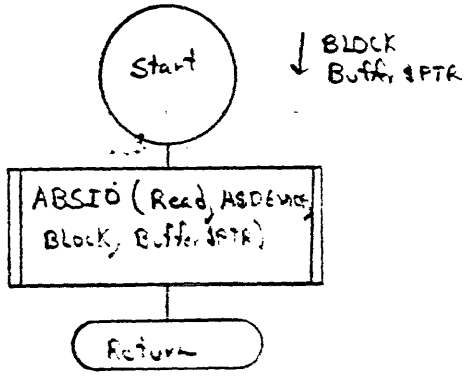
DISKIO


DISKIO
Drive IOPB
Turn off interrupt 1
Double Density ?
Yes
set up to do double density
No
I = Ø
I = Max# Retries
Yes
No
Save Drive # for error message
ABORTX (DISKIO Error)
DK$STAT (Funct:..)
Drive Ready ?
No
R$Type
R$BYTE
Yes
set up for SD or DD
Controller Present
Yes
No
STRT$IO (IOPB)
ABORTX (Drive$NOT Ready)
← Controller will finish before returning to here
R$TYPE
Temp = R$BYTE
Temp = Ø (operation complete OK)
No
Yes
STRT$IO (Re-calibrate)
Restore interrupt 1
Return
(Re-Try)

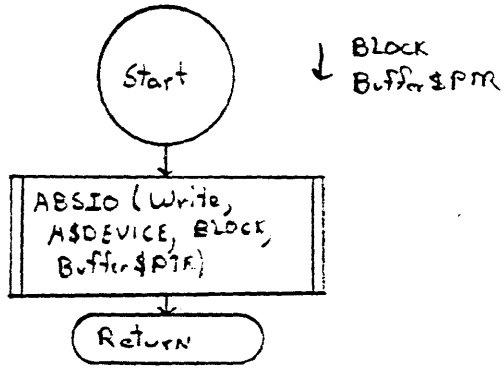
ABSIO

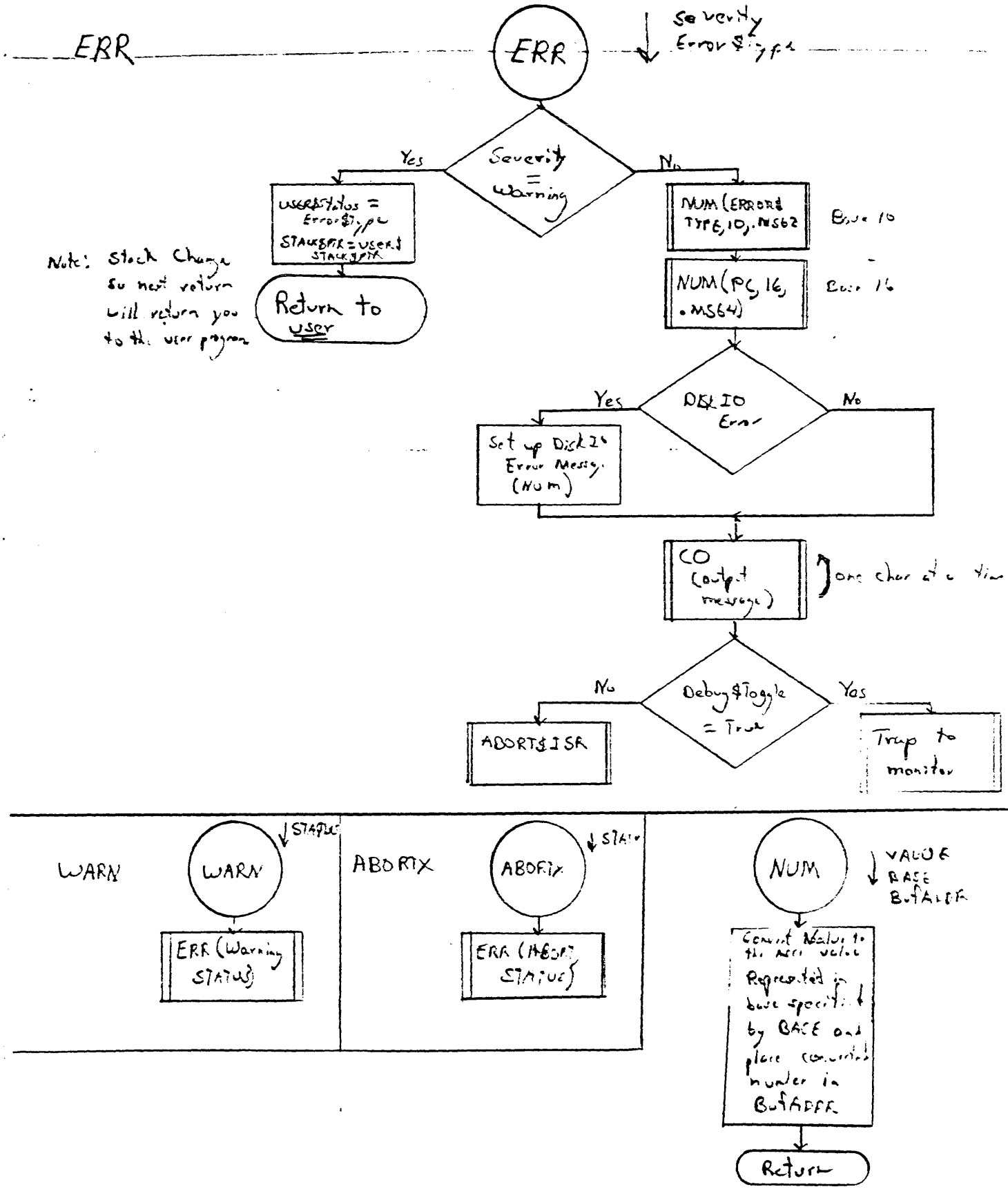


ABSREAD

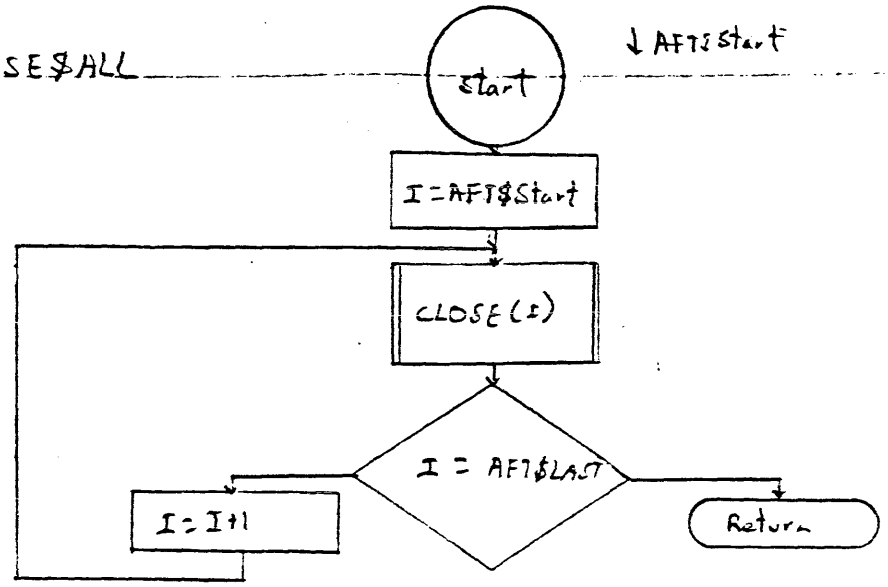


ABSWRITE

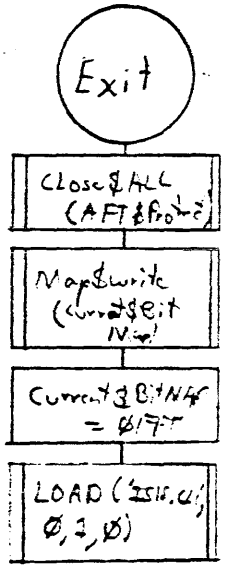



ERR
ERR
Severity
Error $Type
Severity = Warning
Yes
No
USER$Status = Error$Type
STACK$PTR = USER.1 STACK$PTR
Return to USER
Note: Stack Change so next return will return you to the user program
NUM (ERROR$ TYPE, 10, .WS62
Base 10
NUM (PC, 16, .WS64)
Base 16
DEL IO Error
Yes
No
Set up Disk IO Error Message (NUM)
CO (output message)
one char at a time
Debug$Toggle = True
No
Yes
ABORT$ISR
Trap to monitor
WARN
WARN
STATUS
ERR (Warning STATUS)
ABORT
ABORT
STATUS
ERR (ABORT STATUS)
NUM
VALUE
BASE
BUFADDR
Convert Value to the ASCII value Represented in base specified by BASE and place converted number in BUFADDR
Return

CLOSE$ALL

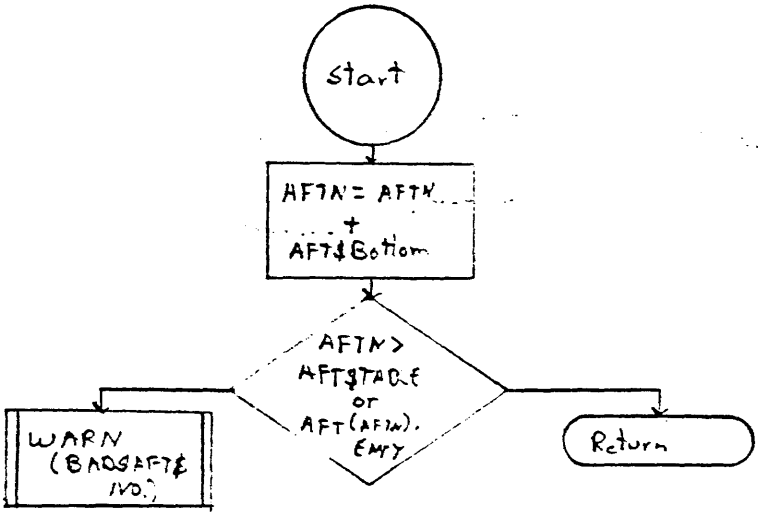


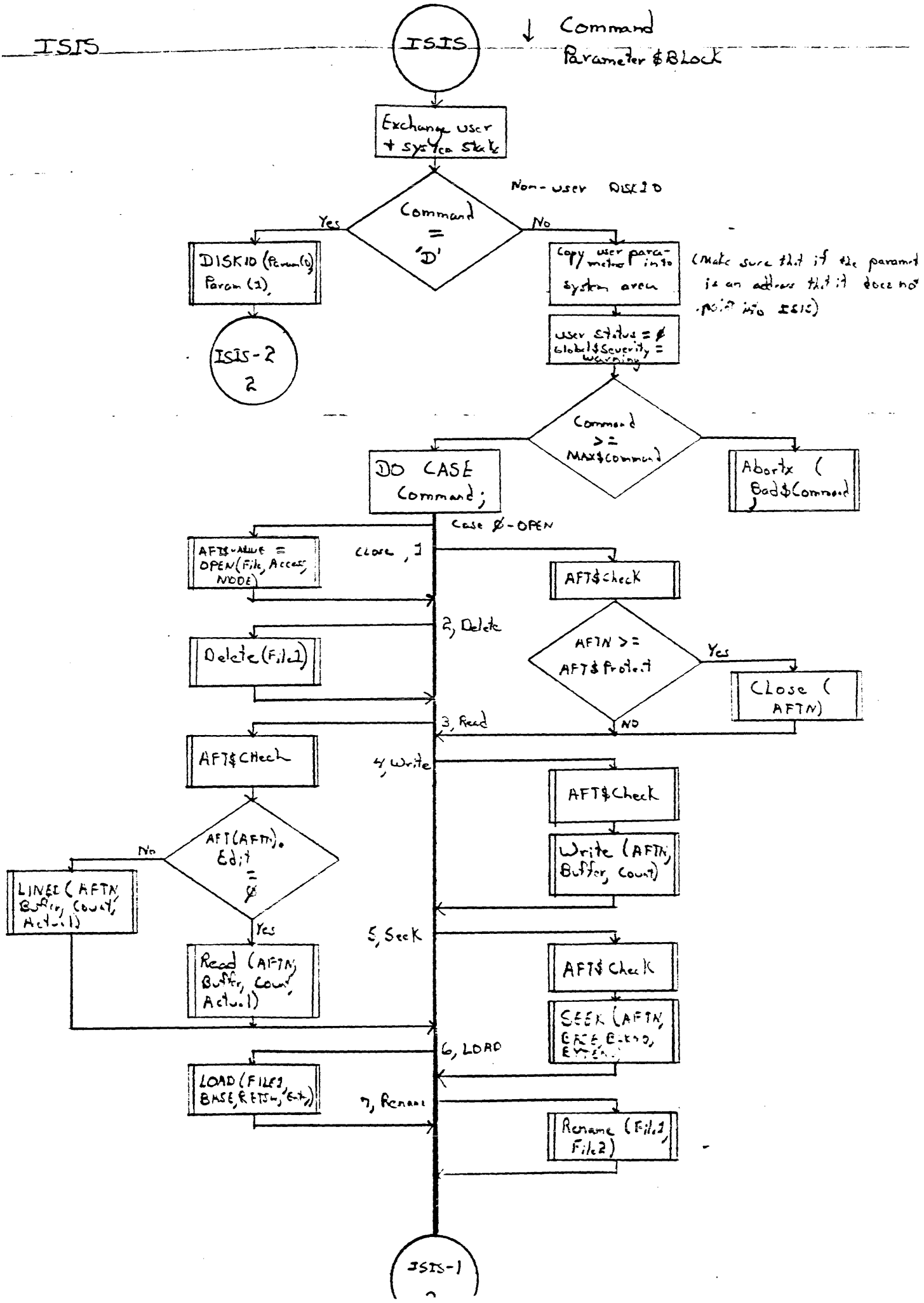
Exit



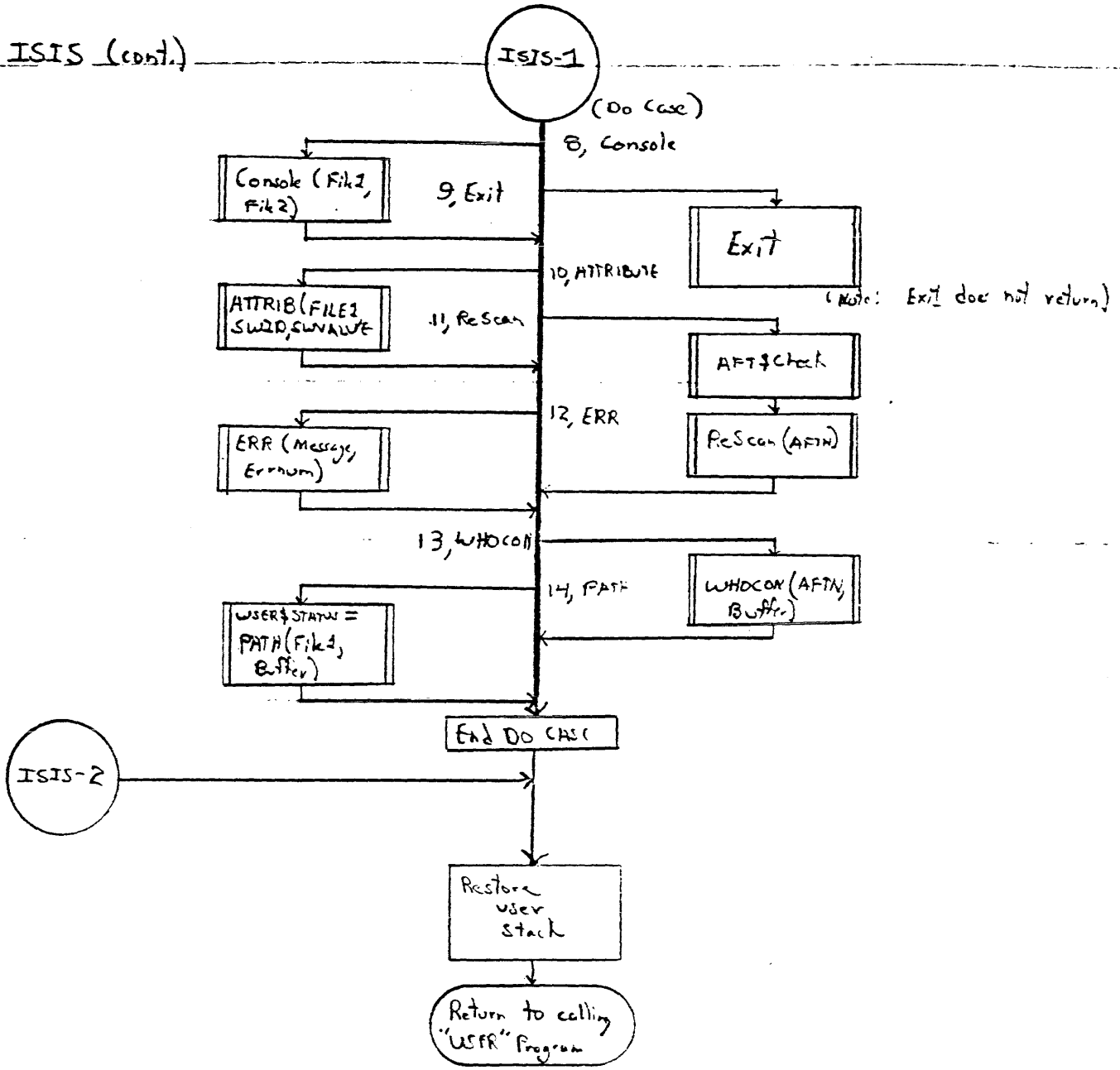
Load + run ISIS.CLI

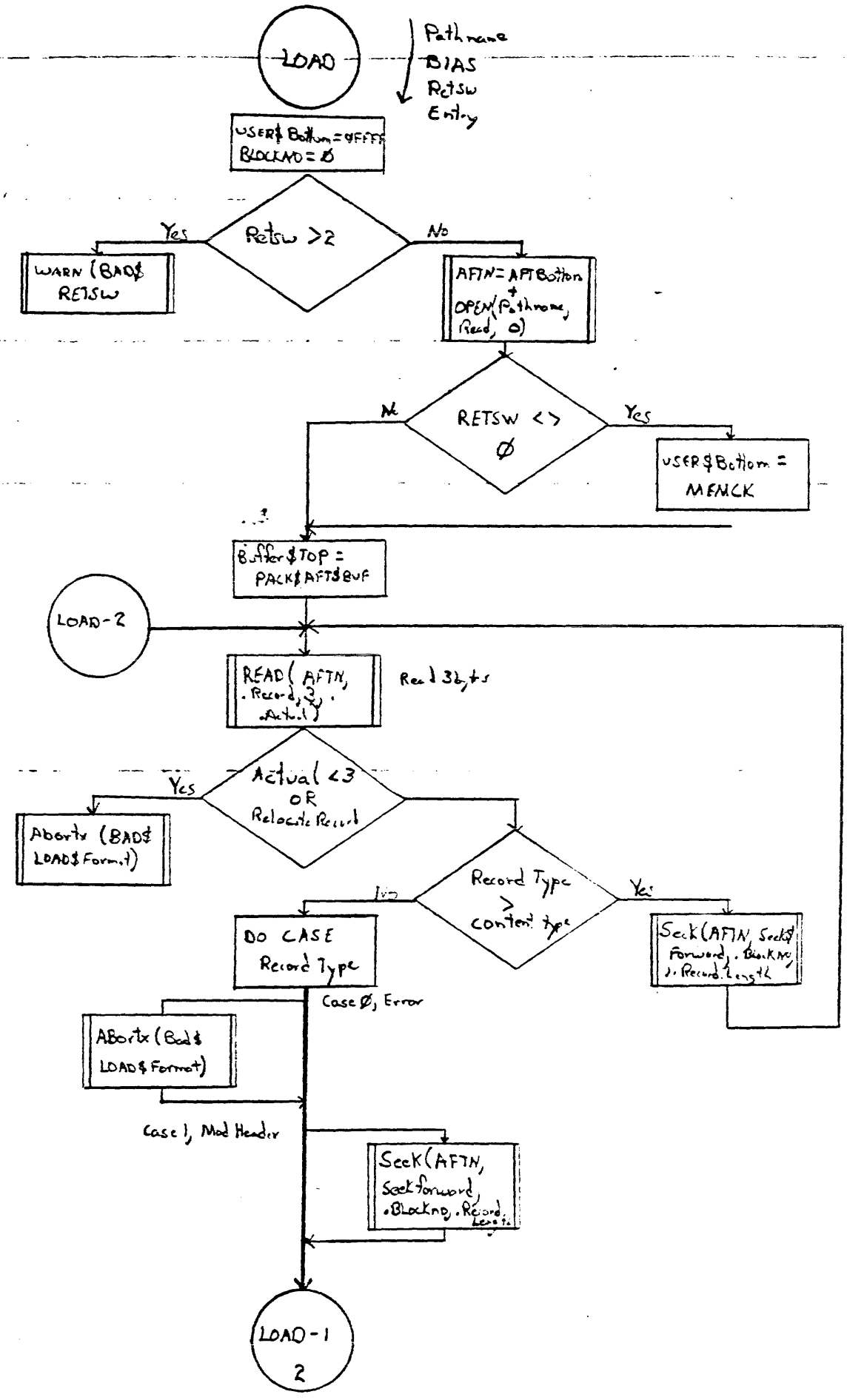
AFT$Check (ISIS)



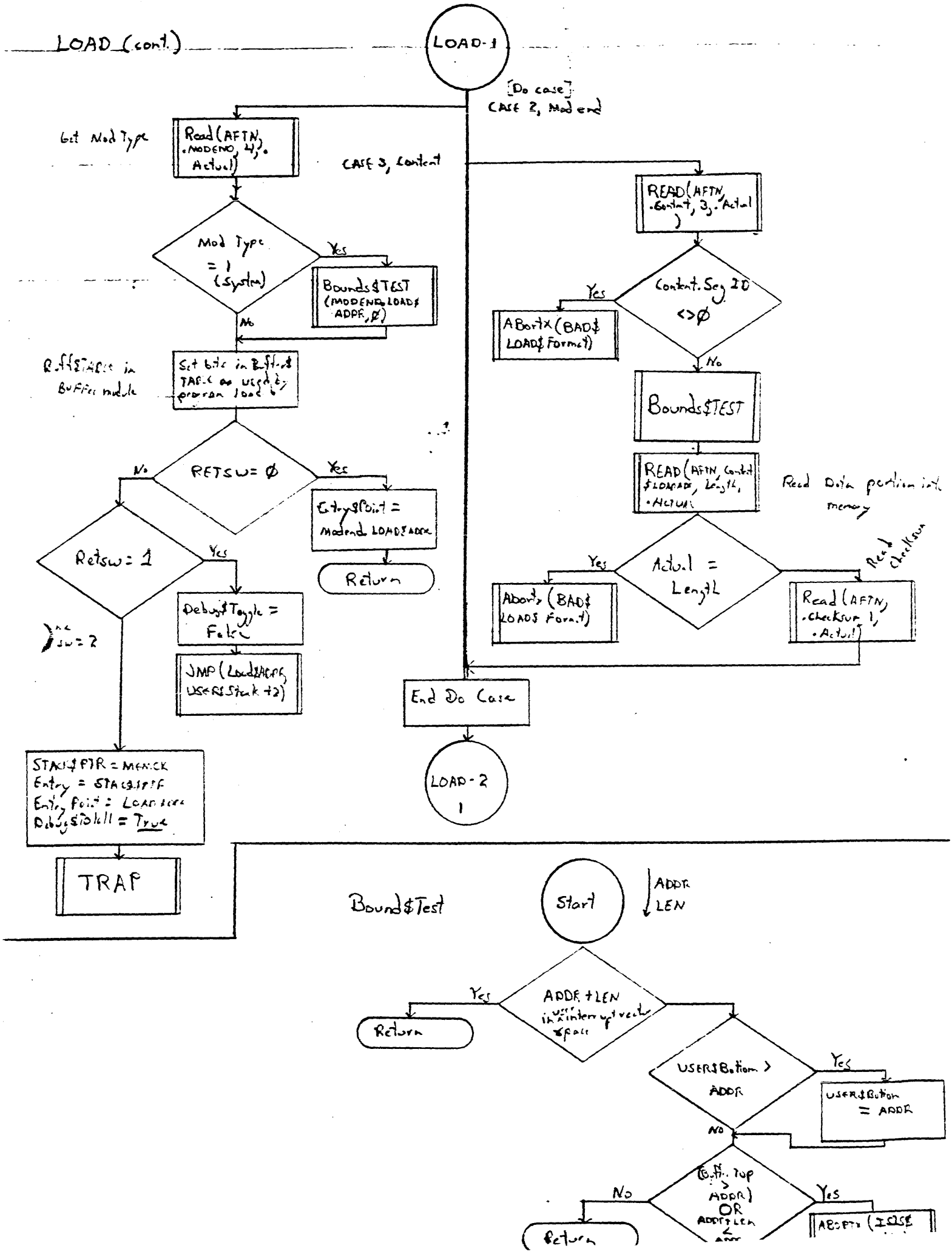

ISIS
↓ Command Parameter $Block
Exchange user + system stacks
Command = 'D'
Yes
No
Non-user DISKIO
DISKIO (Param(0), Param(1),
ISIS-2 2
Copy user parameter into system area
(Make sure that if the parameter is an address that it does not point into ISIS)
user status = 0
Global$Severity = warning
Command >= MAX$command
Abortx (Bad$Command)
DO CASE Command;
Case 0 - OPEN
AFT$Alive = OPEN(File, Access, MODE)
Close, 1
AFT$Check
2, Delete
Delete(File)
AFTN >= AFT$Protect
Yes
Close (AFTN)
No
3, Read
AFT$CHeck
4, Write
AFT$Check
Write (AFTN, Buffer, Count)
AFT(AFTN). Edit = 0
No
LINEI (AFTN, Buffer, Count, Actual)
Yes
5, Seek
Read (AFTN, Buffer, Count, Actual)
AFT$Check
SEEK (AFTN, ORIG, BLOCK, BYTE)
6, LOAD
LOAD (FILE, BASE, RETN, Ext)
7, Rename
Rename (File1, File2)
ISIS-1 0

ISIS (cont.)


ISIS-1
(Do case)
8, Console
Console (File1, File2)
9, Exit
Exit
(Note: Exit does not return)
10, ATTRIBUTE
ATTRIB(FILE1 SW1D,SWVALUE
11, ReScan
AFT$Check
12, ERR
ERR (Message, Errnum)
ReScan (AFTN)
13, WHOCON
14, PATH
WHOCON (AFTN, Buffer)
USER$STATUS = PATH(File1, Buffer)
End DO CASE
ISIS-2
Restore user stack
Return to calling "USER" Program

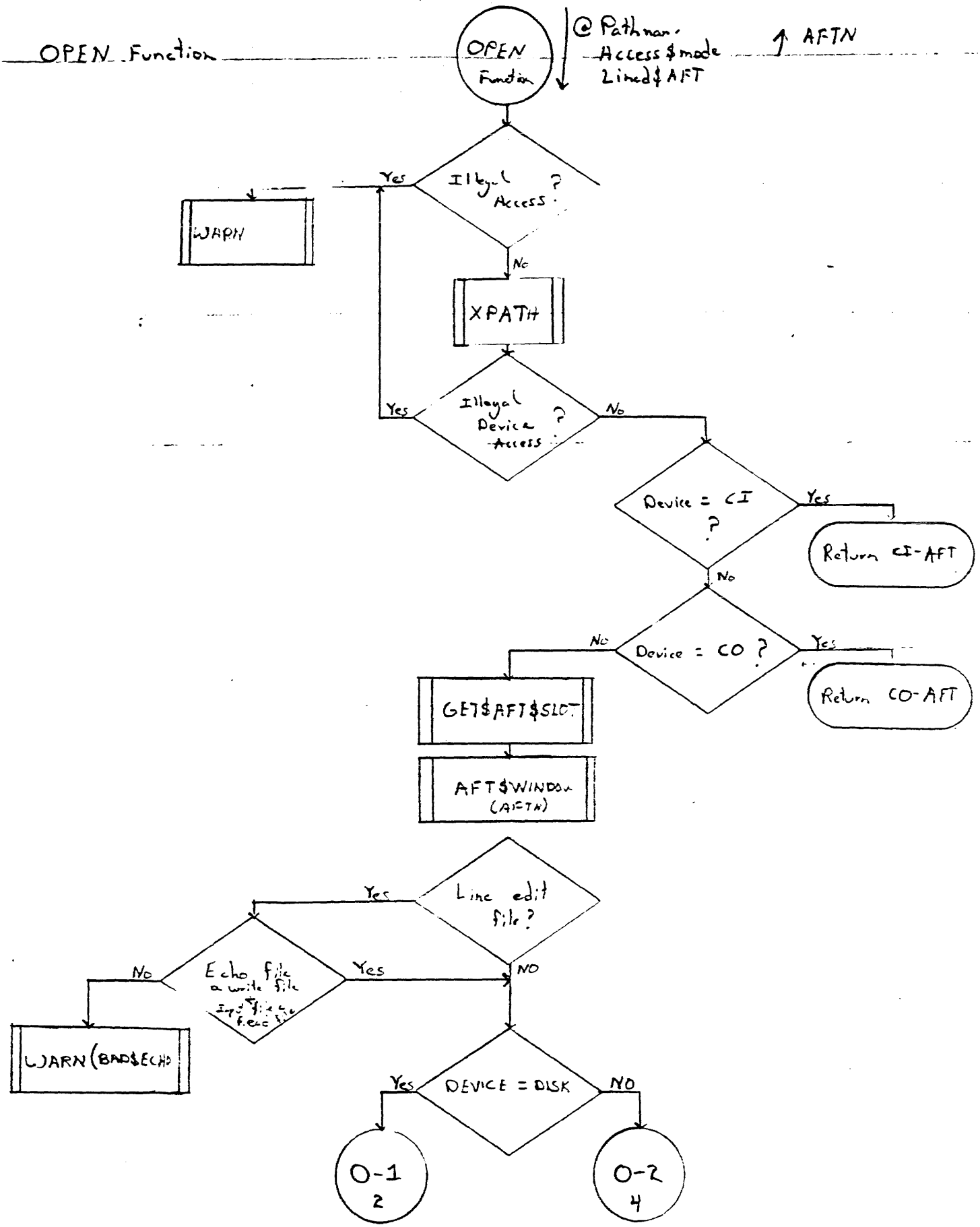

LOAD
Pathname
BIAS
Retsw
Entry
USER$Bottom = 0FFFF
BLOCKNO = 0
Retsw >2
Yes
No
WARN (BAD$ RETSW
AFTN = AFTBottom + OPEN(Pathname, Read, 0)
RETSW <> 0
No
Yes
USER$Bottom = MEMCK
Buffer$TOP = PACK$AFT$BUF
LOAD-2
READ(AFTN, .Record, 3, .Actual)
Read 3 bytes
Actual <3 OR Relocate Record
Yes
Abortx (BAD$ LOAD$Format)
Record Type > content type
No
Yes
DO CASE Record Type
Seek(AFTN, Seek$ Forward, .BlockNo, .Record.Length
Case 0, Error
Abortx (Bad$ LOAD$Format)
Case 1, Mod Header
Seek(AFTN, Seek forward, .Blockno, .Record.Length
LOAD-1
2

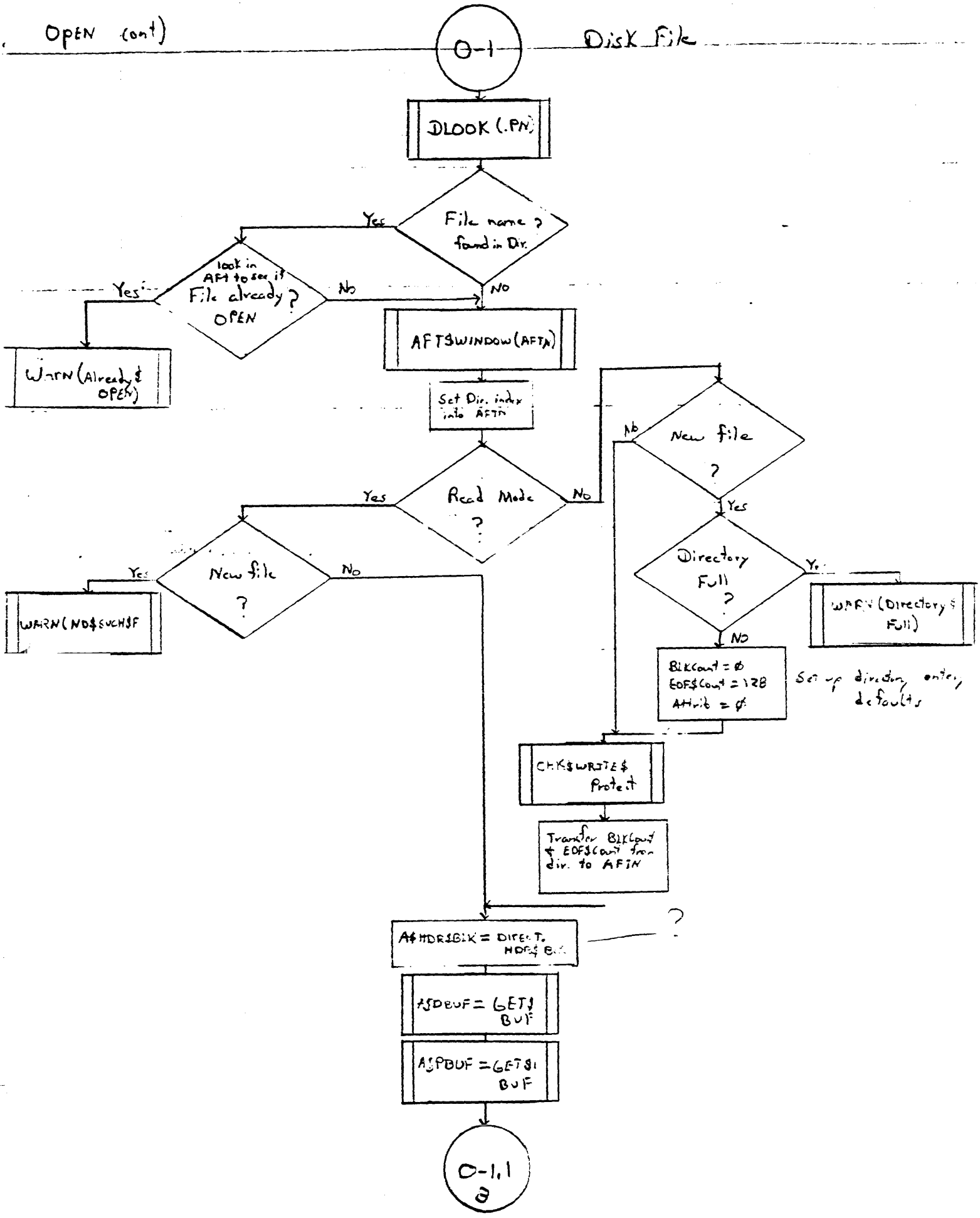
LOAD (cont.)



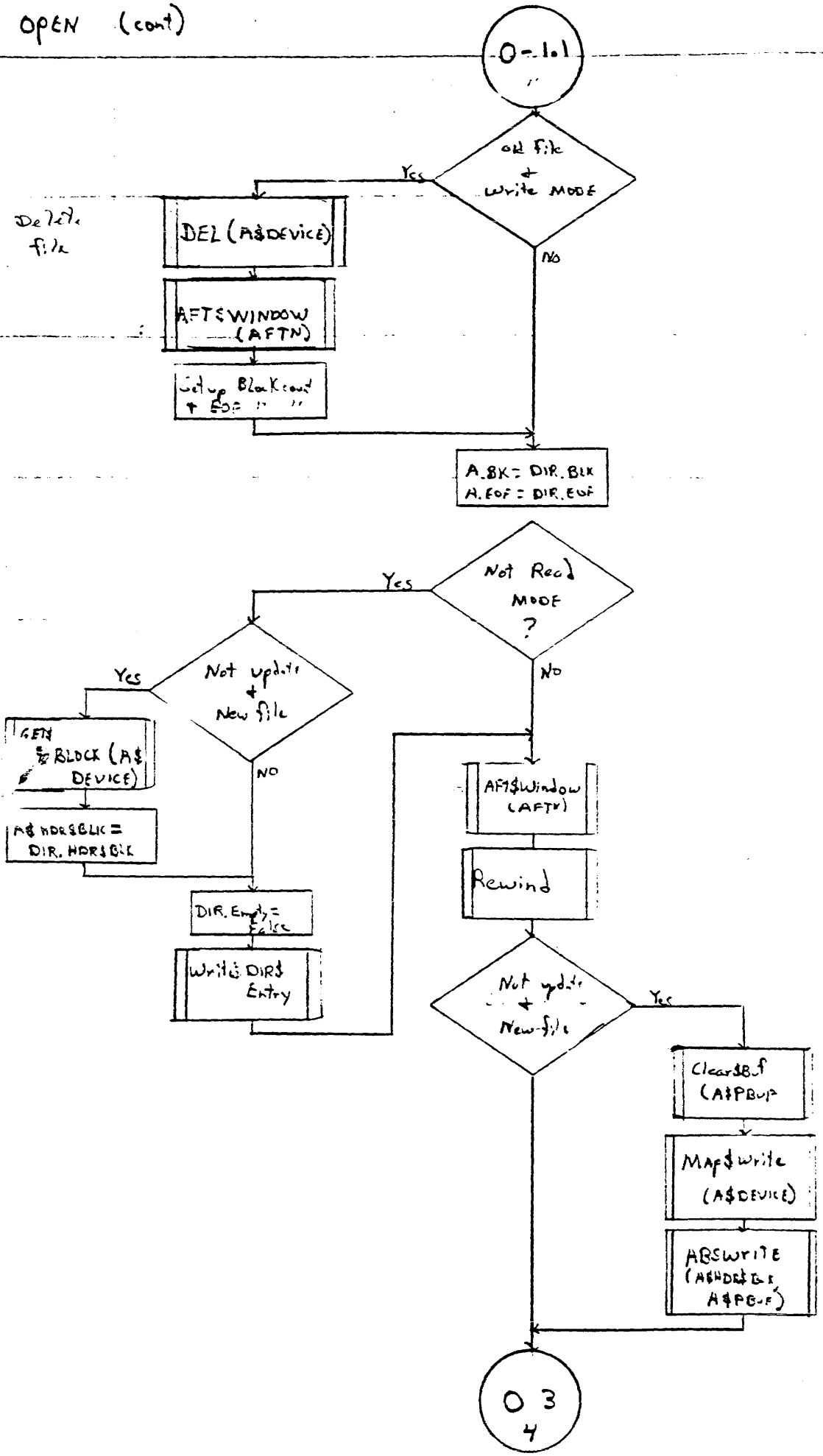
Bound$Test

OPEN Function

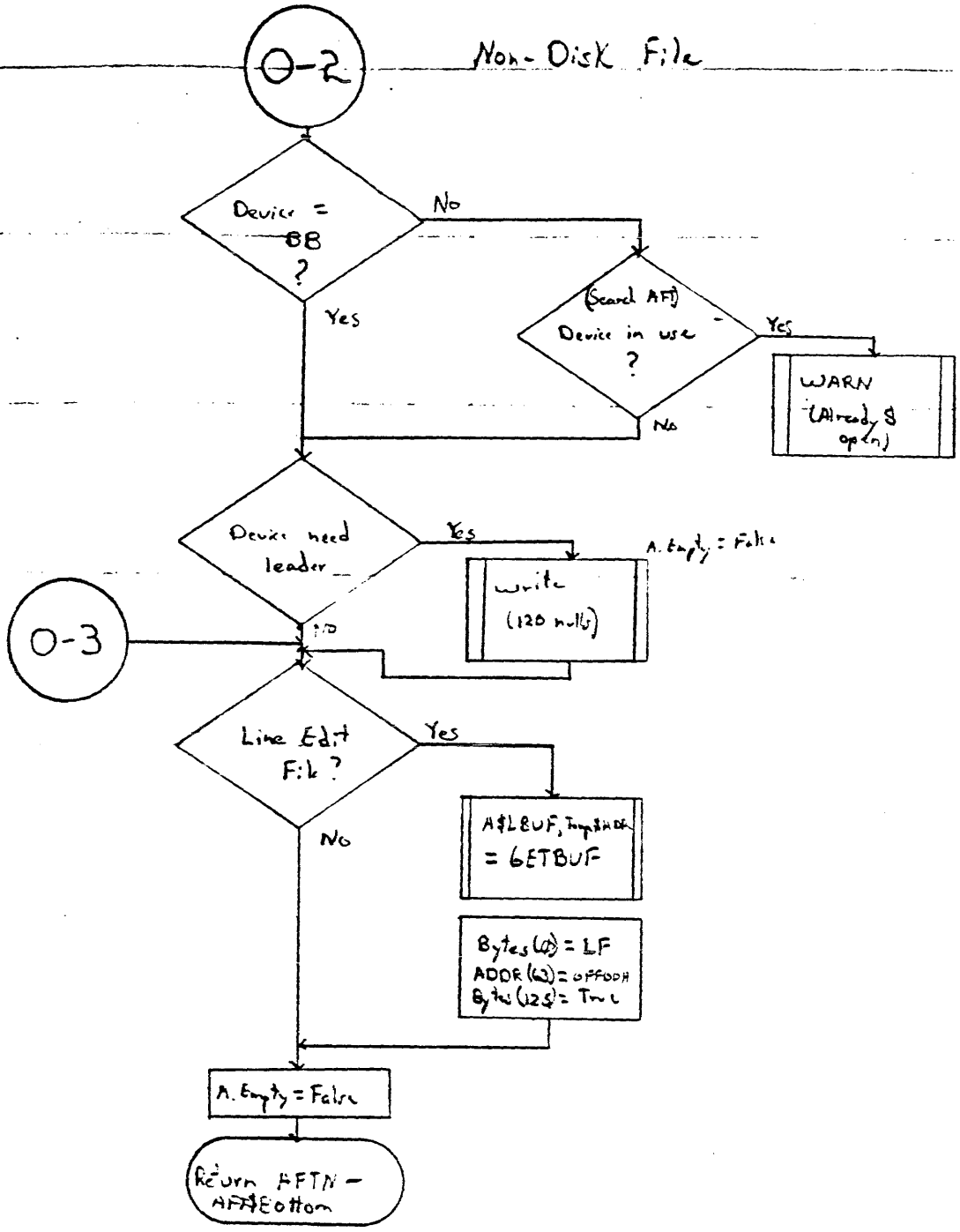



DLOOK (.PN)
File name ? found in Dir.
Yes
No
look in AFT to see if File already ? OPEN
Yes
No
WARN (Already OPEN)
AFT$WINDOW (AFTN)
Set Dir. index into AFTN
Read Mode ?
Yes
No
New file ?
Yes
No
WARN (NO$SUCHF
New file ?
No
Yes
Directory Full ?
Yes
WARN (Directory Full)
No
BLKcount = 0
EOF$Count = 128
Attrib = 0
Set up directory entry defaults
CHK$WRITE$ Protect
Transfer BLKcount & EOF$Count from dir. to AFTN
AHDRBLK = DIRECT. HDR$ BLK
A$DBUF = GET$1 BUF
A$PBUF = GET$1 BUF
O-1.1

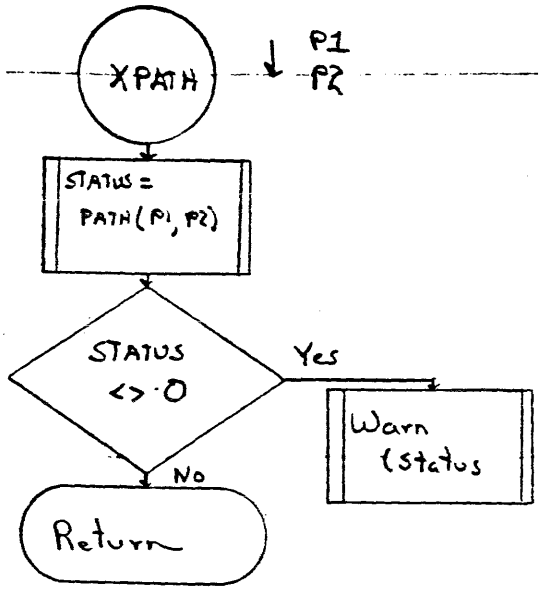
OPEN (cont)


0-1.1
Old file + write MODE
Yes
No
Delete file
DEL (A$DEVICE)
AFT$WINDOW (AFTN)
Set up Block count + EOF
A.BK = DIR.BLK
A.EOF = DIR.EOF
Not Read MODE ?
Yes
No
Not update + New file
Yes
No
GET $BLOCK (A$ DEVICE)
A$ HDR$BLK = DIR.HDR$BLK
DIR.Empty = False
Write DIR$ Entry
AFT$Window (AFTN)
Rewind
Not update + New file
Yes
Clear$BUF (A$PBUF)
MAP$write (A$DEVICE)
ABSWRITE (AHDRBLK, A$PBUF)
0 3
4

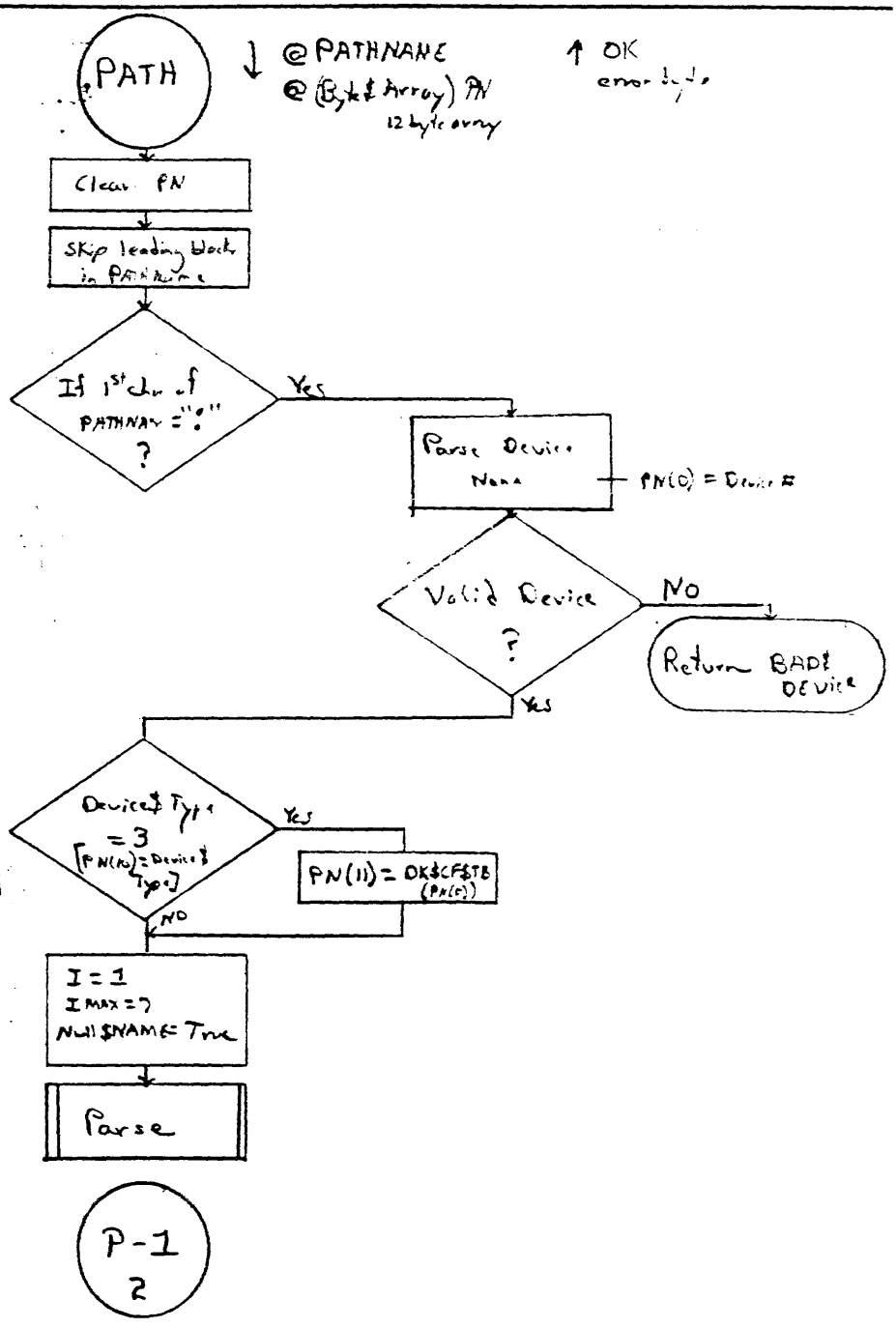
OPEN (cont)

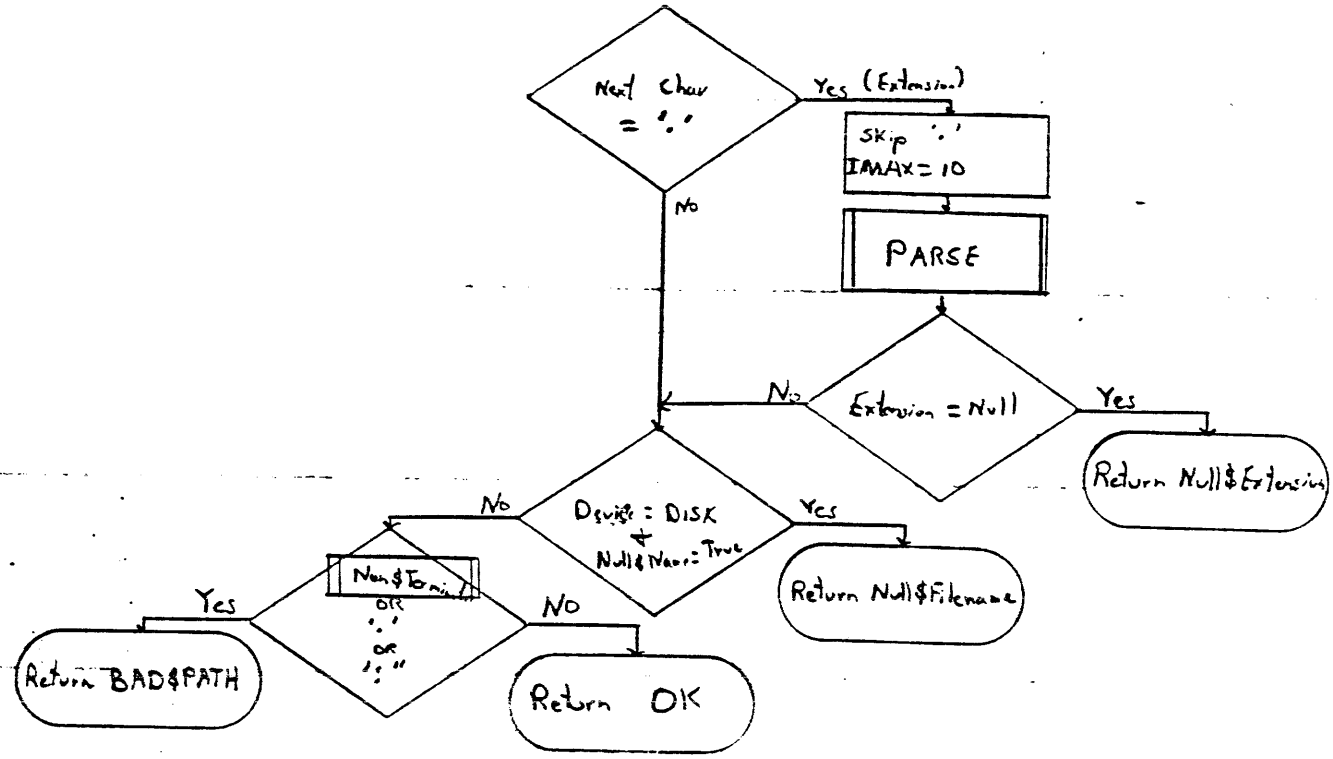

O-2
Non-Disk File
Device = 88 ?
No
Yes
(Scan AFT) Device in use ?
Yes
No
WARN (Already Open)
Device need leader
Yes
A.Empty = False
write (120 nulls)
No
O-3
Line Edit File ?
Yes
No
A$LBUF, TmpHDR = GETBUF
Bytes(0) = LF
ADDR(0) = OFFOOH
Bytes(125) = Tmp
A.Empty = False
Return AFTN - AFTBottom

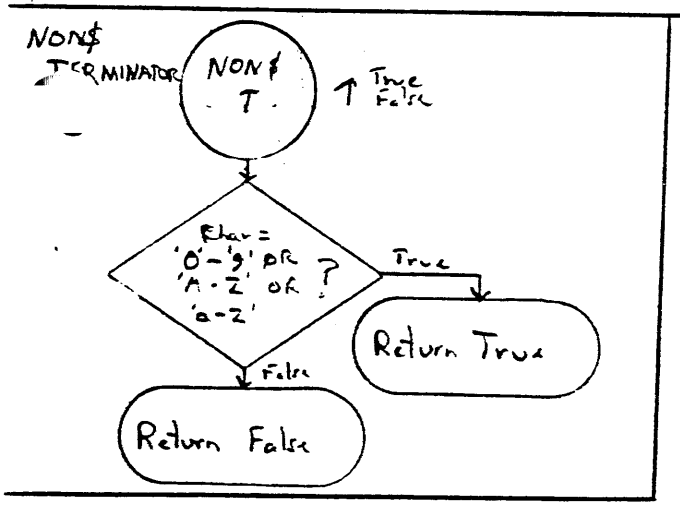
XPATH

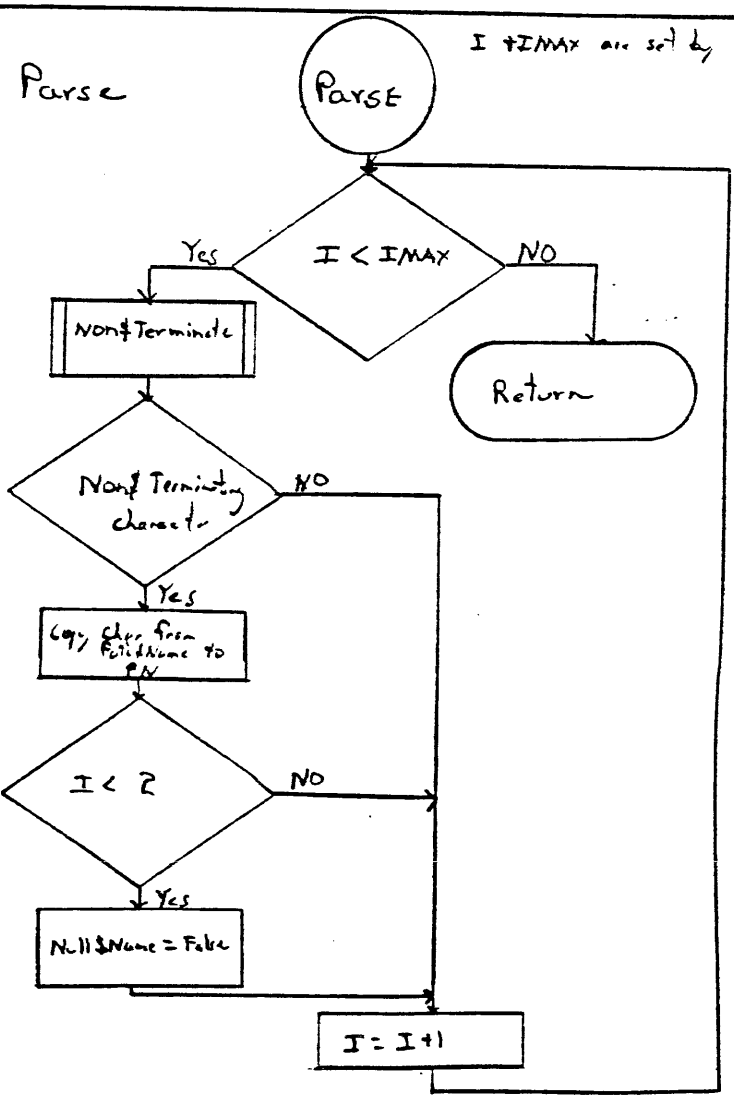


PATH Function

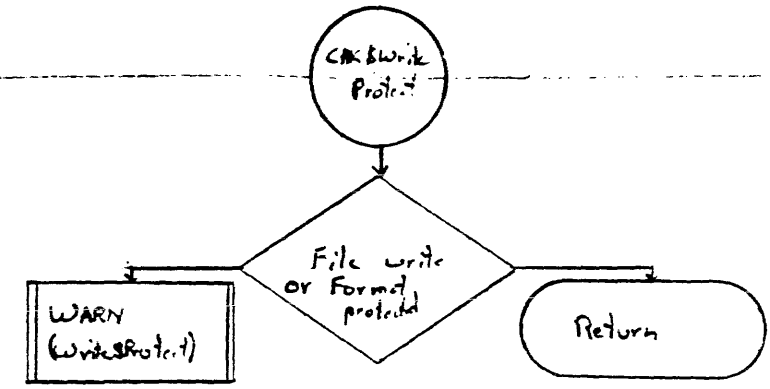



Next Char = '.'
Yes (Extension)
Skip '.'
IMAX = 10
PARSE
No
Extension = Null
No
Yes
Return Null$Extension
Device = DISK + Null$Name = True
No
Yes
Return Null$Filename
Non$Terminator OR '.' OR ':'
Yes
No
Return BAD$PATH
Return OK

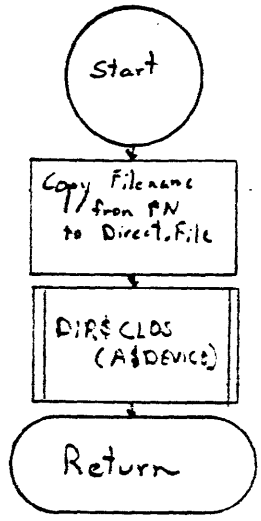

NON$ TERMINATOR
NON$ T
True False
Char = '0-9' OR 'A-Z' OR 'a-z' ?
True
Return True
False
Return False


Parse
I & IMAX are set by PAT.
PARSE
I < IMAX
Yes
NO
Non$Terminate
Return
Non$ Terminating character
NO
Yes
Copy char from FileName to FN
I < 2
NO
Yes
Null$Name = False
I = I+1

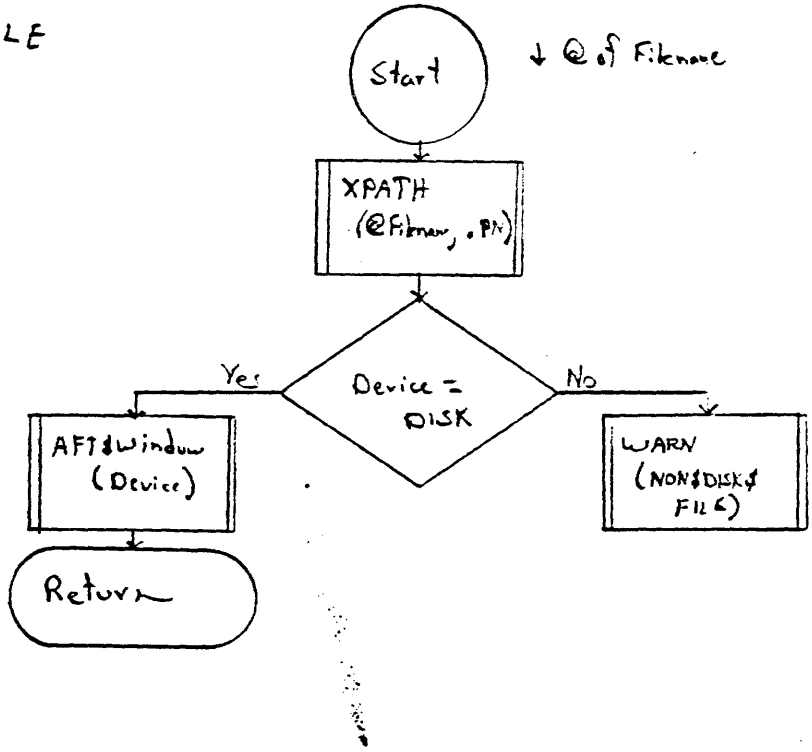
CHK$WRITE$PROTECT



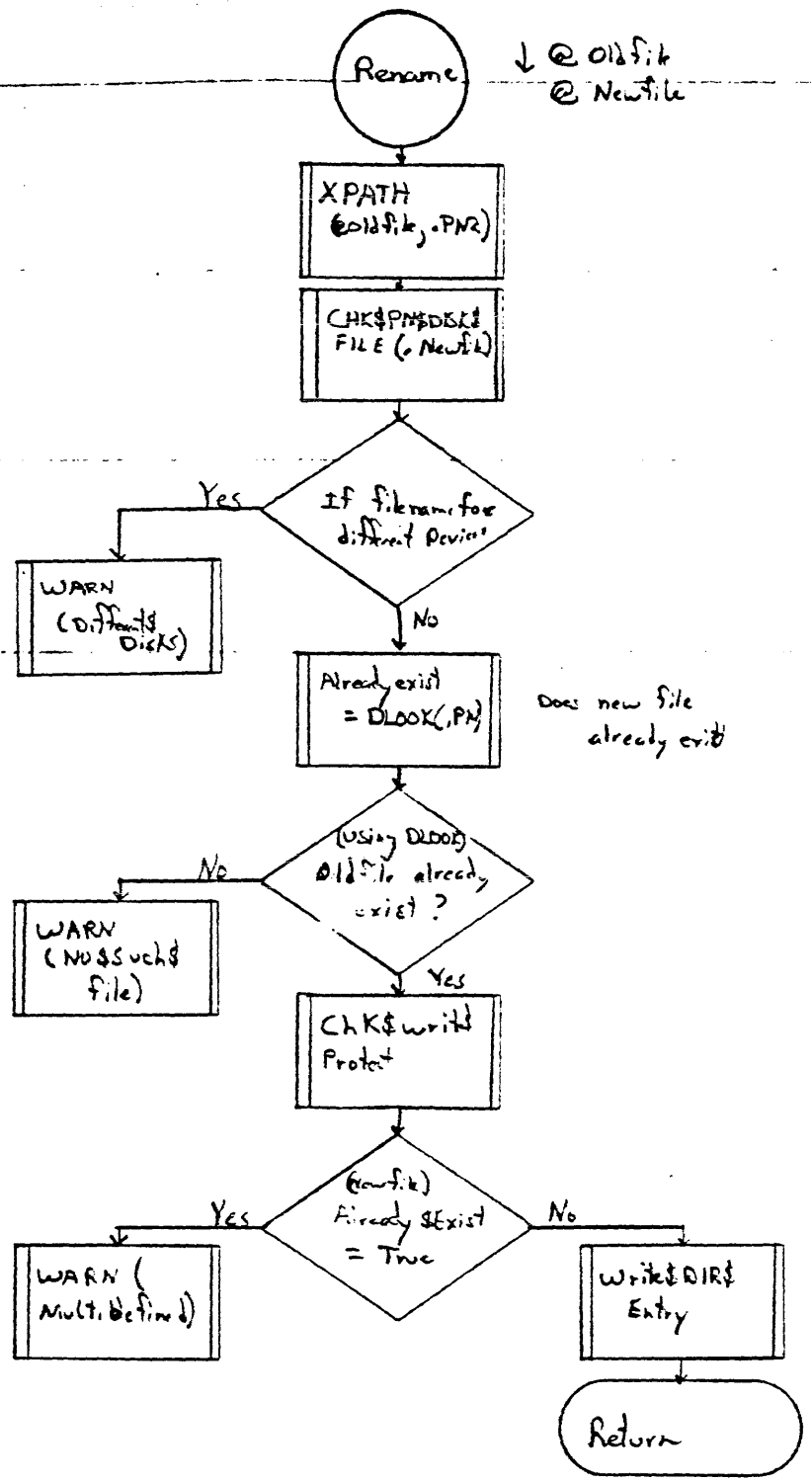
Write DIR Entry



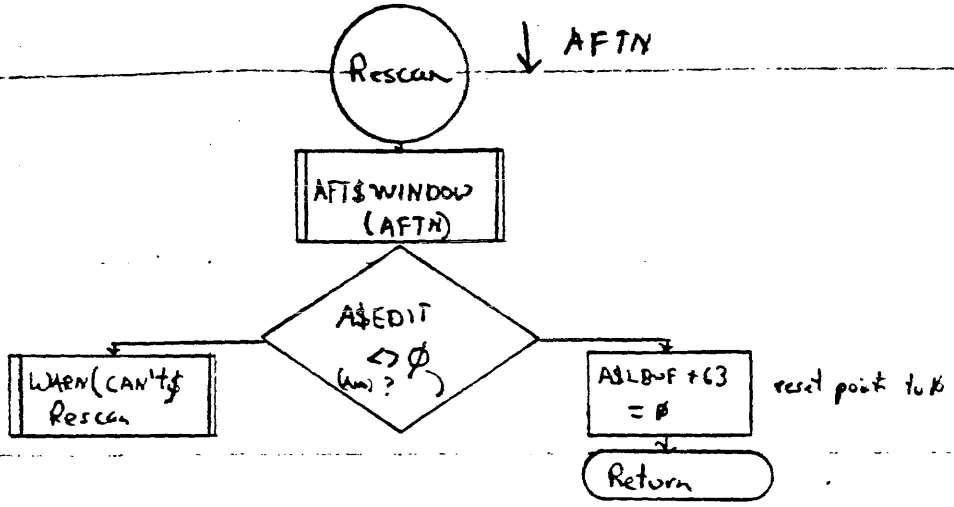
CHKPNDISK$FILE



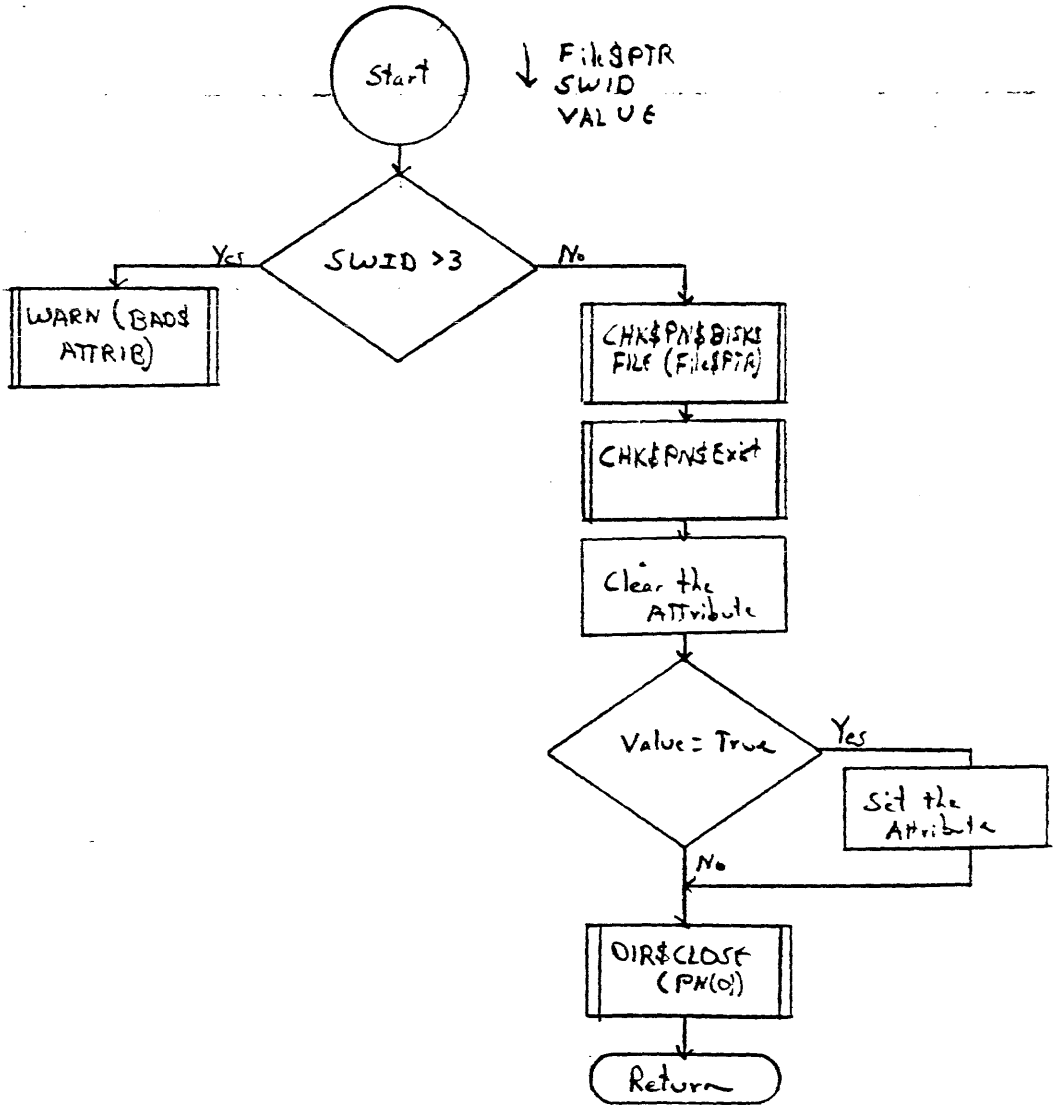
Rename

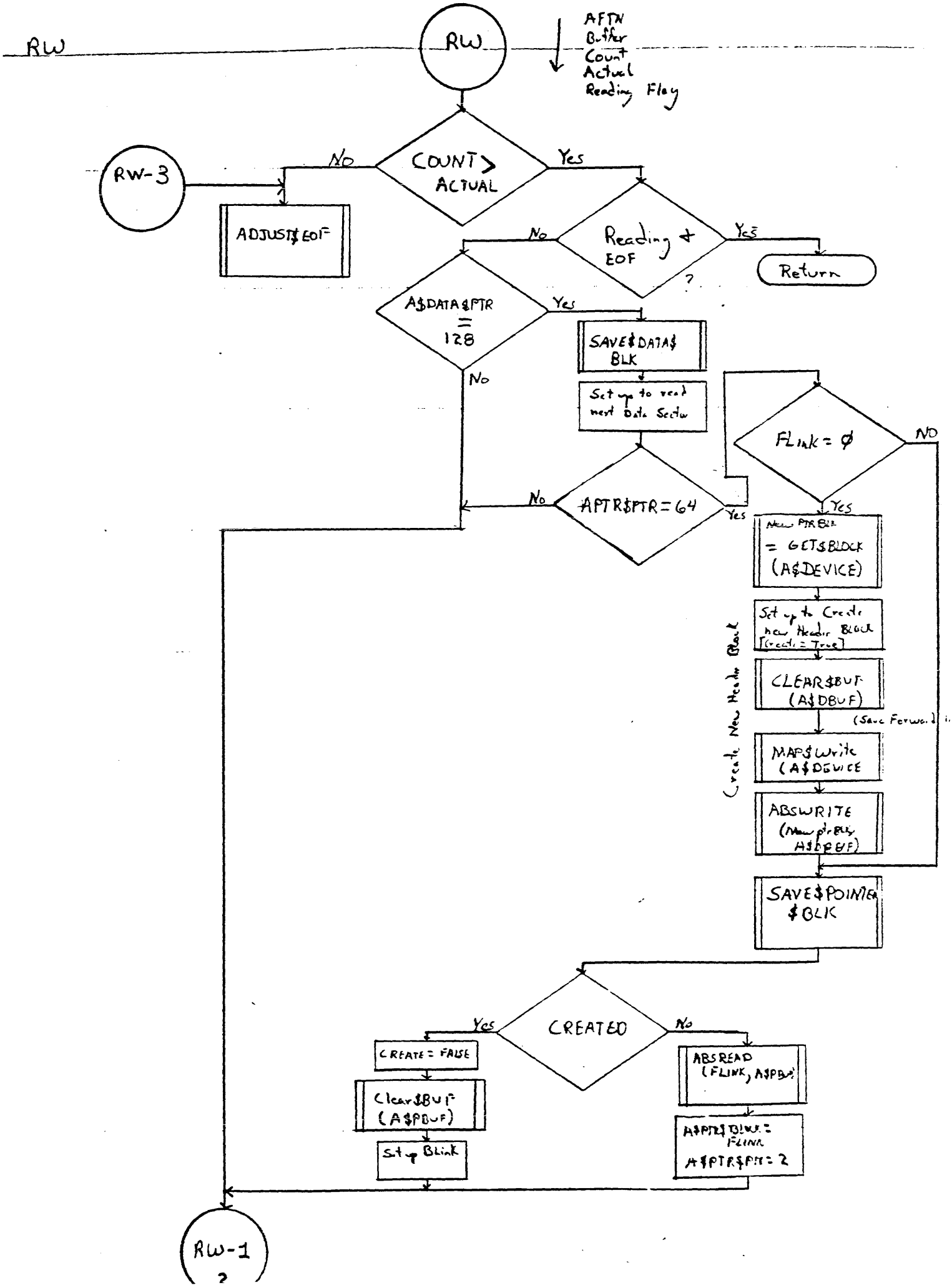

Rename
↓ @ OldFile
@ NewFile
XPATH (OldFile, .PNR)
CHKPNDSK$ FILE (, NewFile)
If filename for different Device
Yes
WARN (Different Disks)
No
Already exist = DLOOK(, PN)
Does new file already exist
(Using DLOOK) OldFile already exist ?
No
WARN (No$Such$ file)
Yes
Chk$write Protect
(NewFile) Already$Exist = True
Yes
WARN (Multi.Defined)
No
WriteDIR Entry
Return

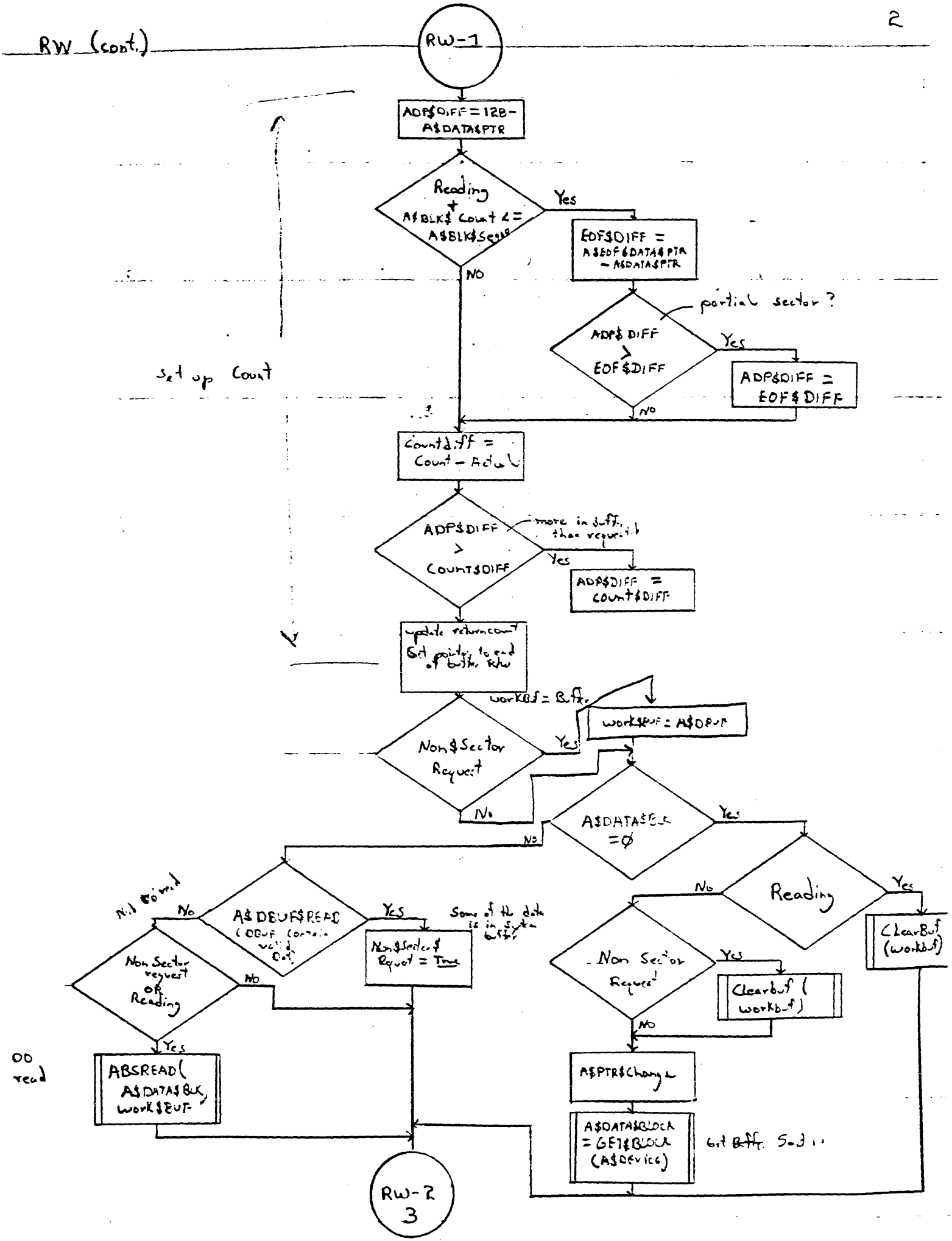
RESCAN

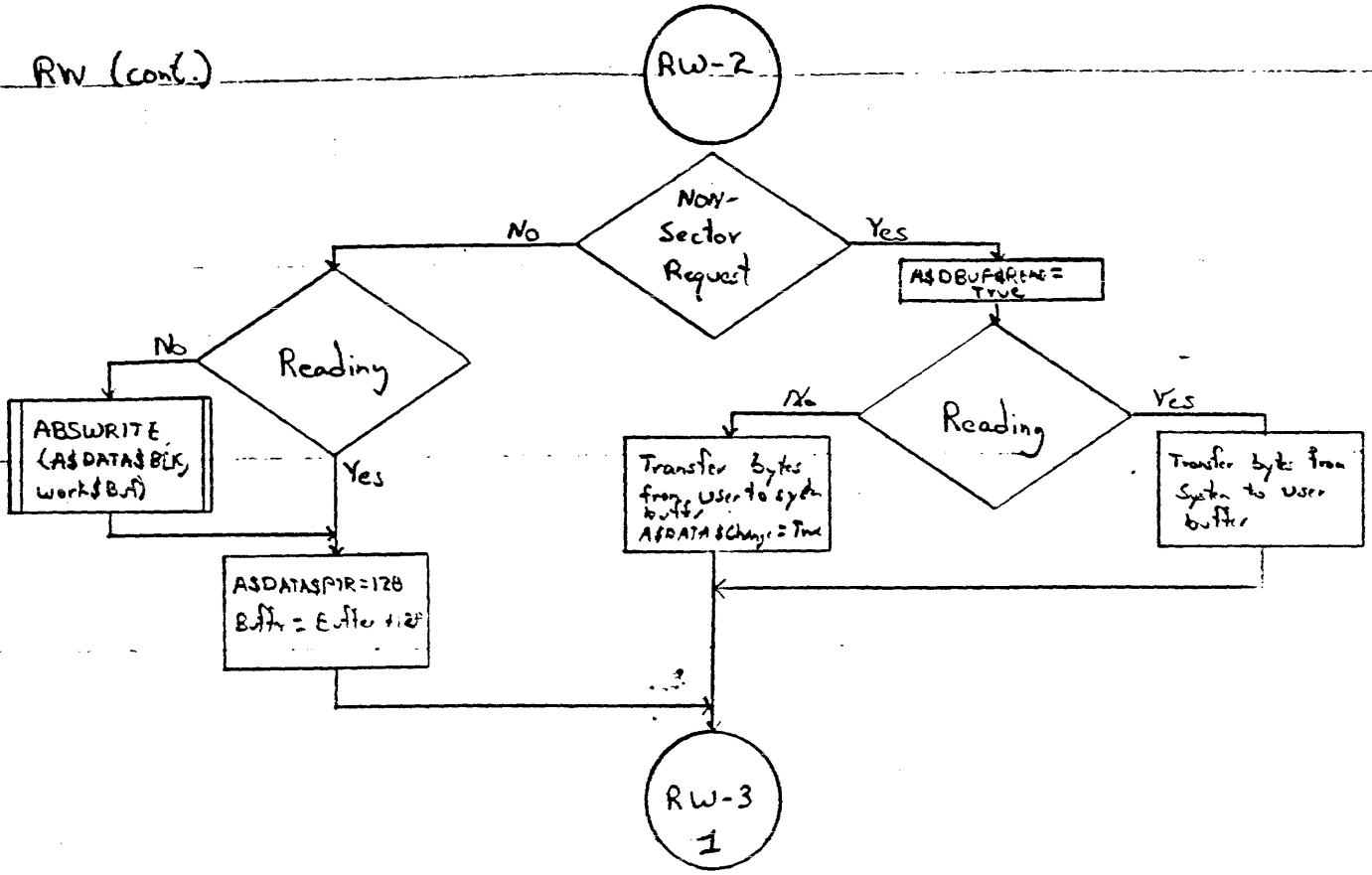


ATTRIB

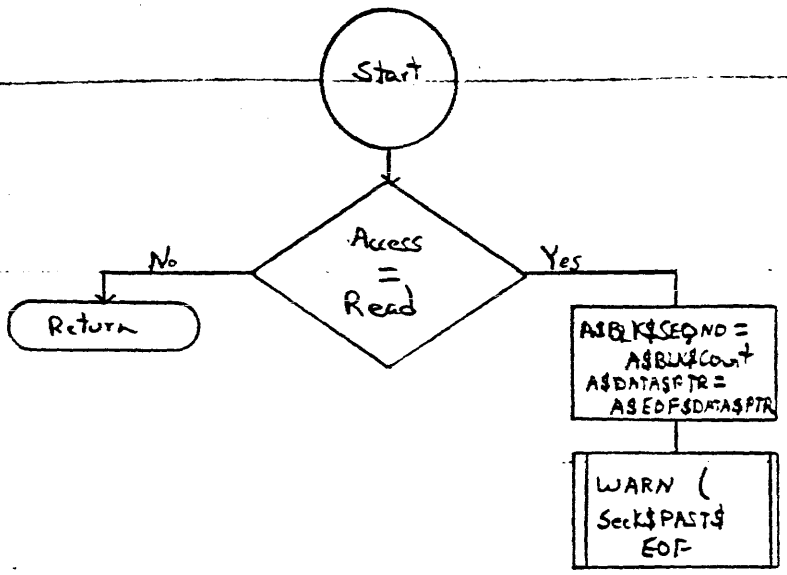



RW
AFTN
Buffer
Count
Actual
Reading Flag
RW-3
No
COUNT > ACTUAL
Yes
ADJUST$EOF
No
Reading + EOF ?
Yes
Return
A$DATA$PTR = 128
Yes
SAVE$DATA$ BLK
Set up to read next Data Sector
No
No
APTR$PTR = 64
Yes
FLink = ∅
NO
Yes
= GET$BLOCK (A$DEVICE)
Set up to Create new Header Block (Create = True)
CLEAR$BUF (A$DBUF)
(Save Forward
MAP$Write (A$DEVICE
ABSWRITE (
A$DBUF)
Create New Header Block
SAVE$POINTER $BLK
CREATED
Yes
No
CREATE = FALSE
Clear$BUF (A$PBUF)
Set BLink
ABSREAD (FLINK, A$PBUF)
FLINK
APTRPTR = 2
RW-1

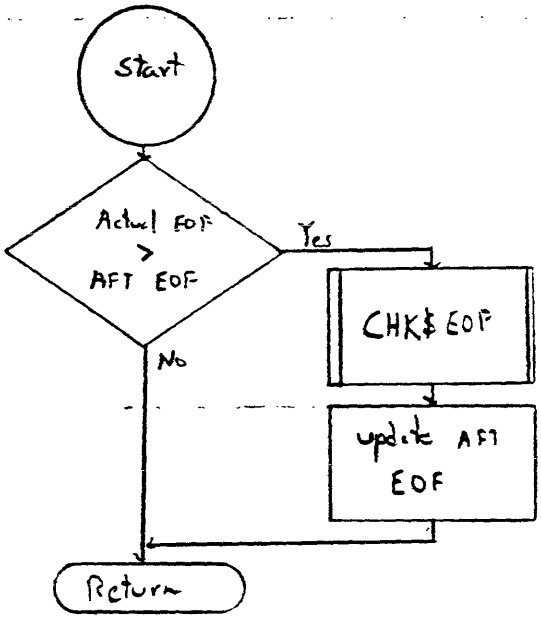

RW-1
ADP$DIFF = 128 - A$DATA$PTR
Reading + ABLK Count <= ABLKSecs
Yes
EOF$DIFF = A$EOF$DATA$ PTR - A$DATA$PTR
NO
partial sector?
ADP$ DIFF > EOF$DIFF
Yes
ADP$DIFF = EOF$ DIFF
NO
set up Count
Count$diff = Count - Adw
ADP$DIFF > COUNT$DIFF
more in buff. than requested
Yes
ADP$DIFF = COUNT$DIFF
update returncount
Set pointer to end of buffer Rdw
workBuf = Buffer
work$BUF = A$DBUF
Non$Sector Request
Yes
No
A$DATA$BLK = 0
Yes
No
Reading
No
Yes
ClearBuf (workbuf)
Non Sector Request
Yes
Clearbuf (workbuf)
No
APTRChange
A$DATA$BLOCK = GET$BLOCK (A$DEVICE)
Get Buffer Sub...
A$ DBUF$ READ (DBUF Contains valid Data)
No
Yes
Some of the data is in system buffer
Non$Sector$ Request = True
Non Sector request OR Reading
No
Yes
ABSREAD(A$DATA$BLK, work$BUF
RW-2
3


RW-2
Non-Sector Request
No
Yes
Reading
No
Yes
ABSWRITE (A$DATA$BLK, work$Buf)
A$DATASPTR=128
Buffer = Buffer + 128
A$DBUFFERENE = True
Reading
No
Yes
Transfer bytes from user to sys buffs, A$DATA$Changed = True
Transfer byte from Sys to user buffer
RW-3
1

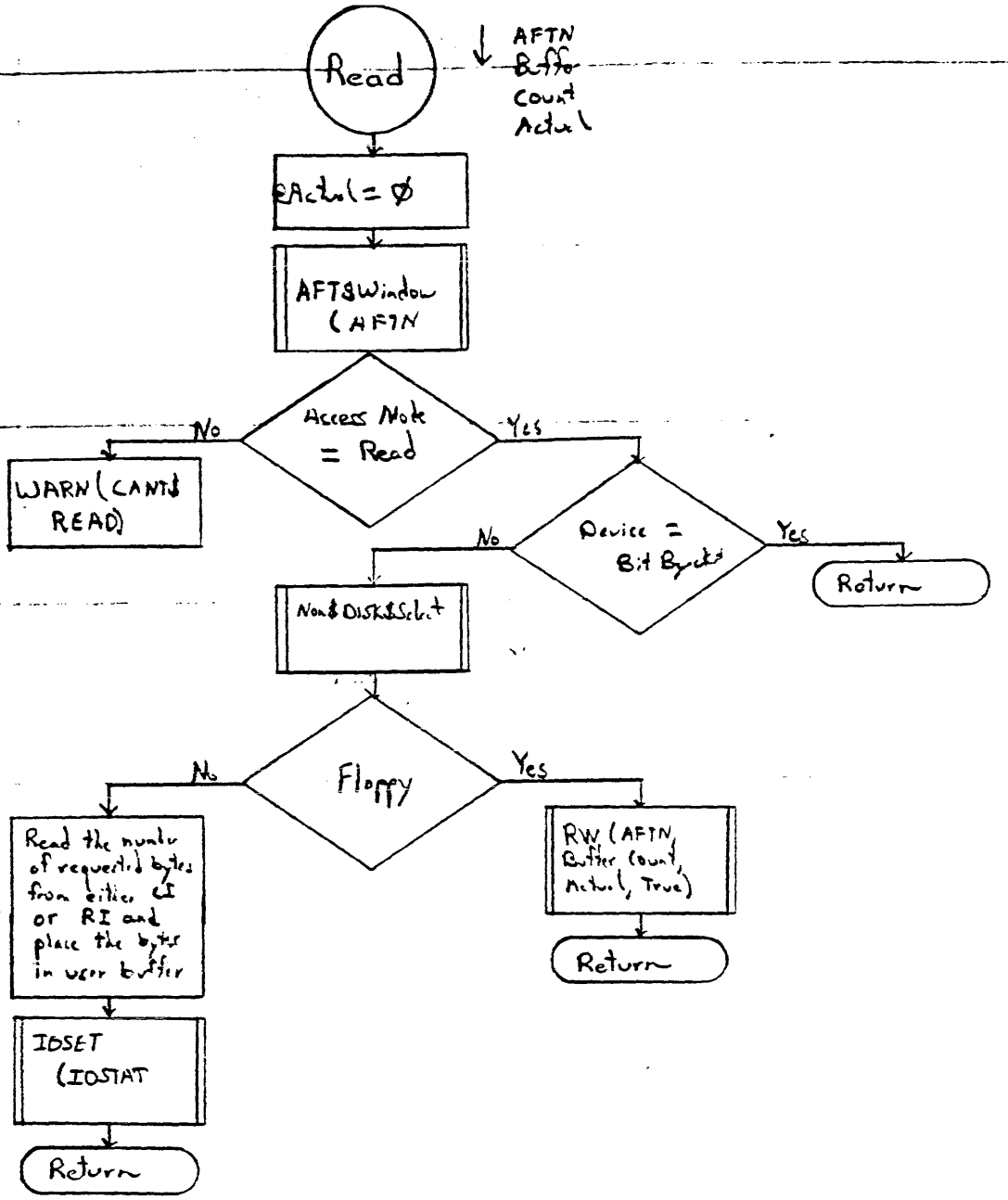
CHK$EOF



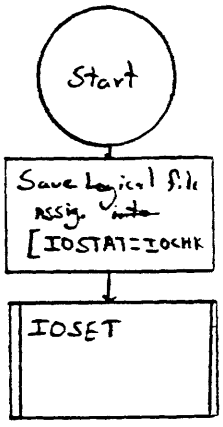
Adjust$EOF



Read

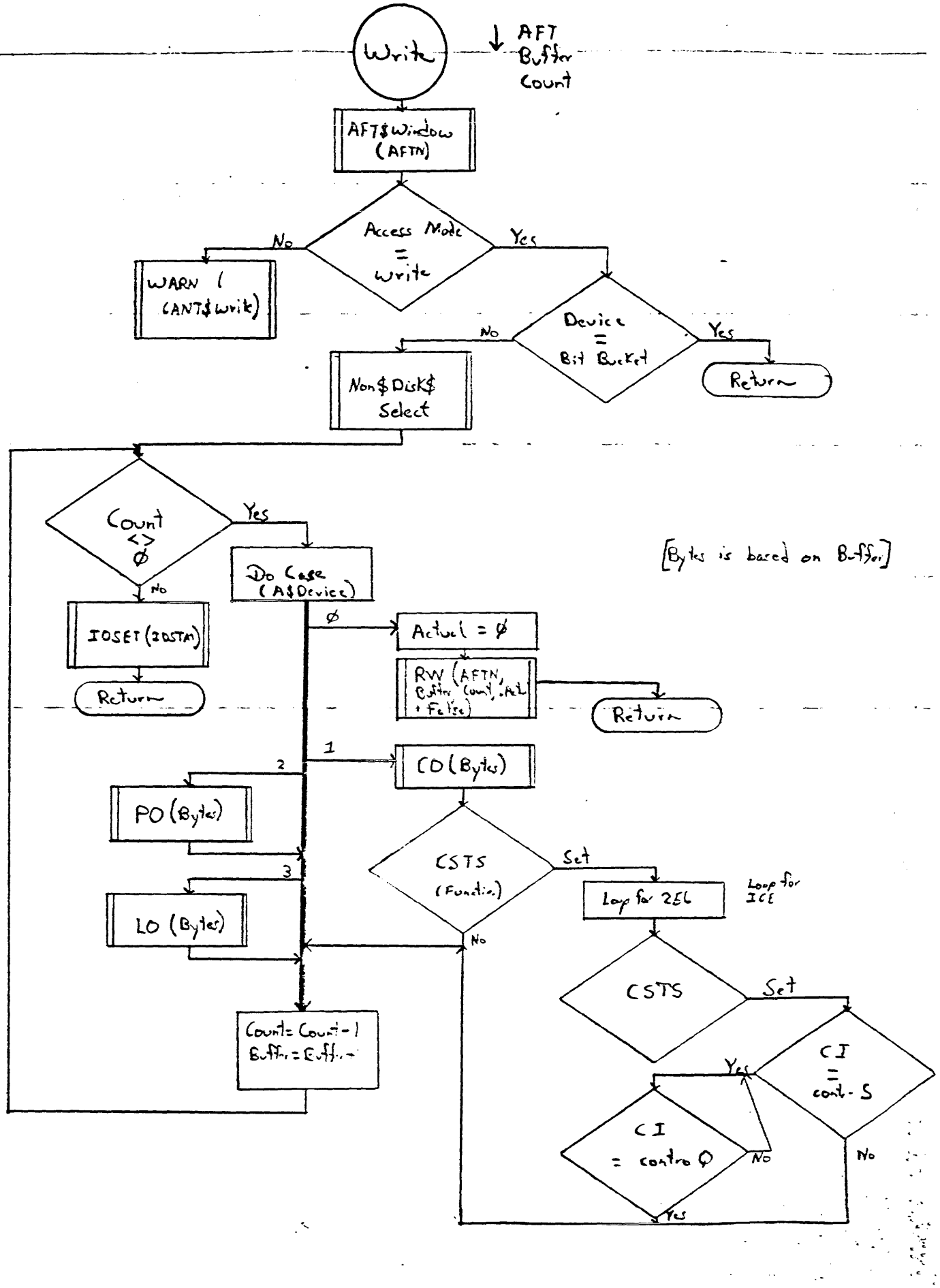


Non$Disk$Select

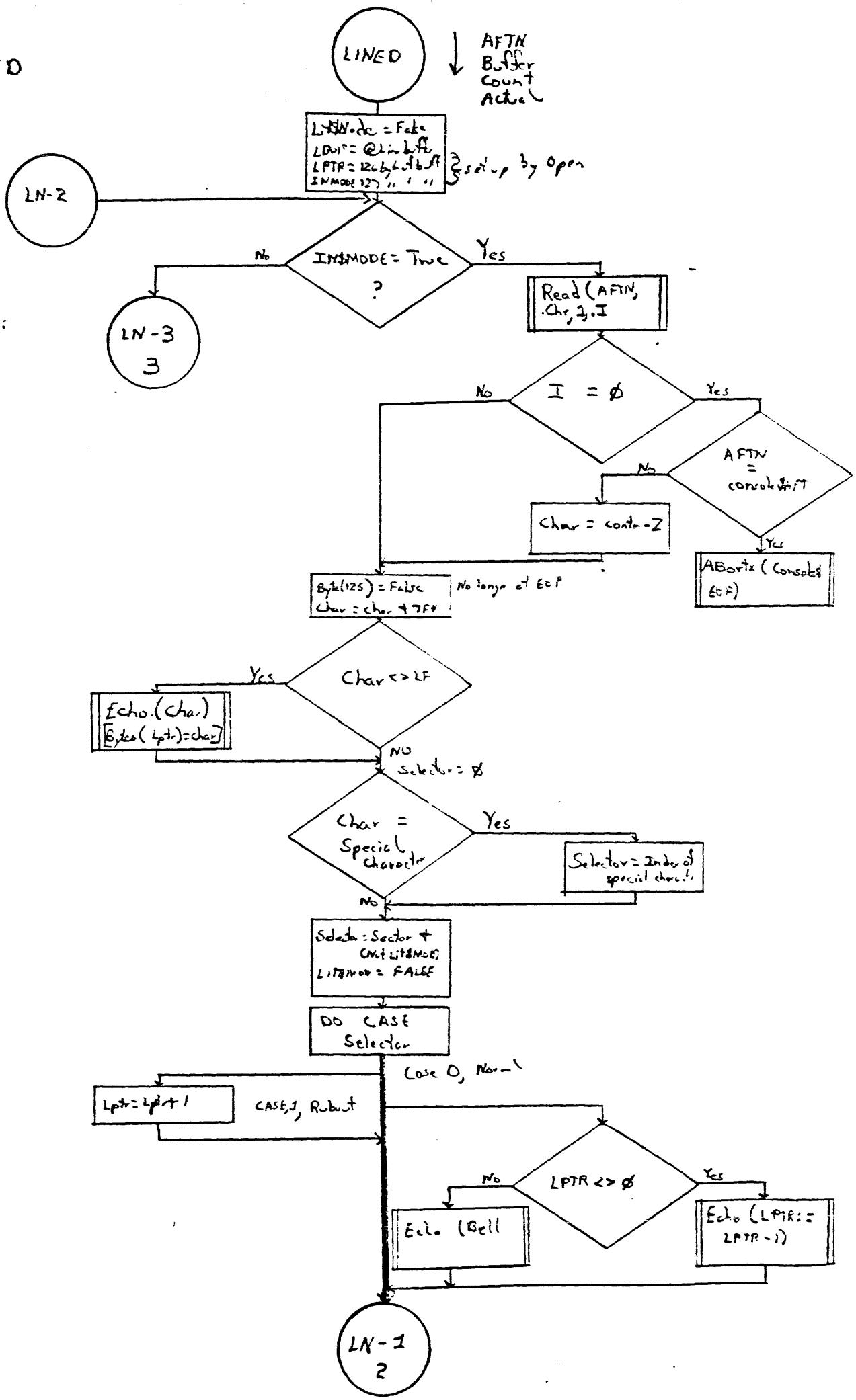


Set logical file assignments to use monitor routines for doing byte at a time I/O devices

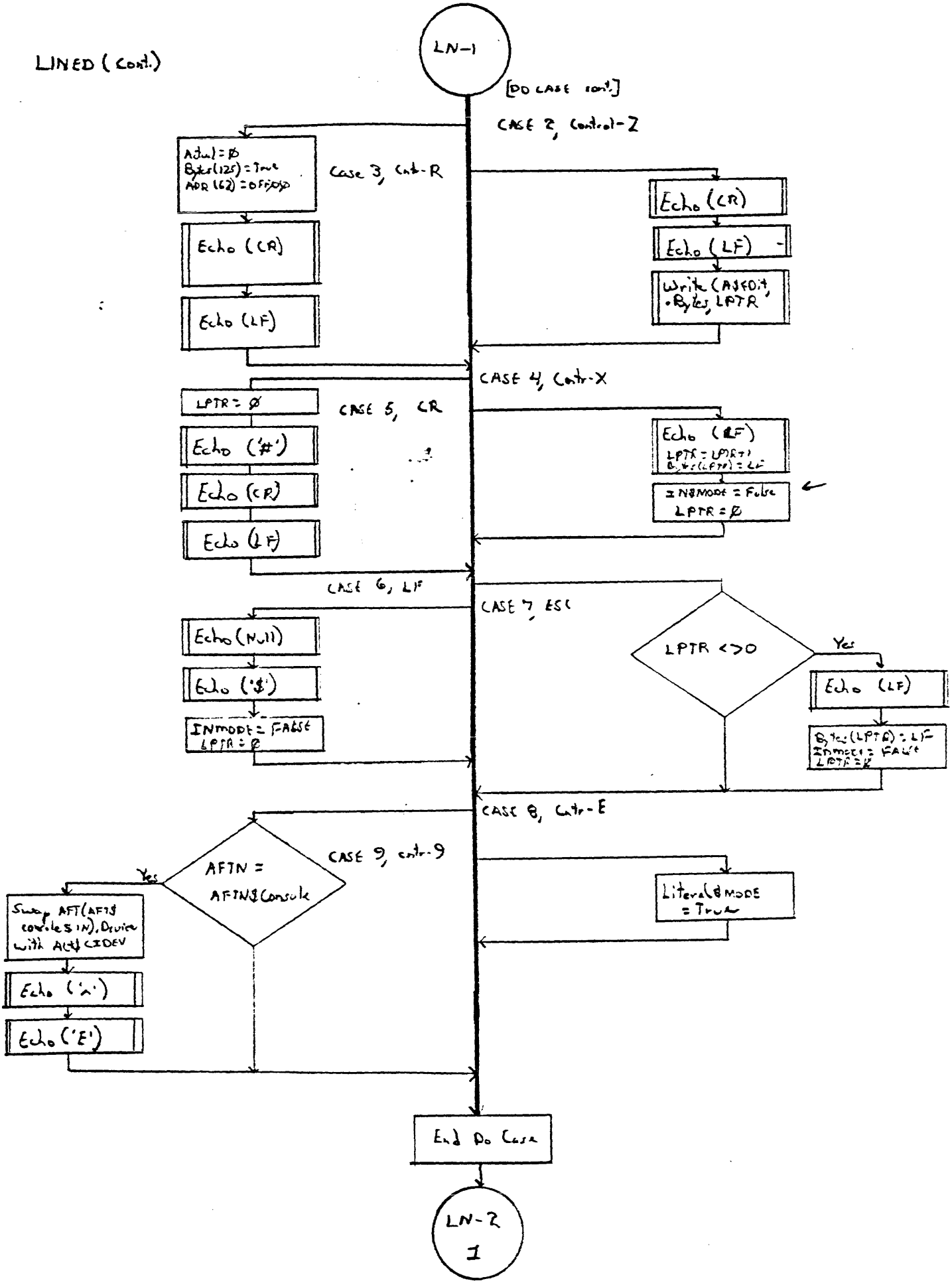
Write


Write
AFT
Buffer
Count
AFT$Window
(AFTN)
Access Mode
=
Write
No
Yes
WARN (
CANT$writ)
Device
=
Bit Bucket
No
Yes
Return
Non$Disk$
Select
Count
<>
Ø
Yes
No
Do Case
(A$Device)
IOSET (2ØSTN)
Return
[Bytes is based on Buffer]
Ø
Actual = Ø
RW (AFTN,
Buffer, Count, Actl
+ False)
Return
1
CO (Bytes)
2
PO (Bytes)
3
LO (Bytes)
CSTS
(Functio.)
Set
No
Loop for 256
Loop for
ICE
CSTS
Set
CI
=
cont-S
Yes
CI
= control Q
No
Yes
No
Count = Count - 1
Buffer = Buffer

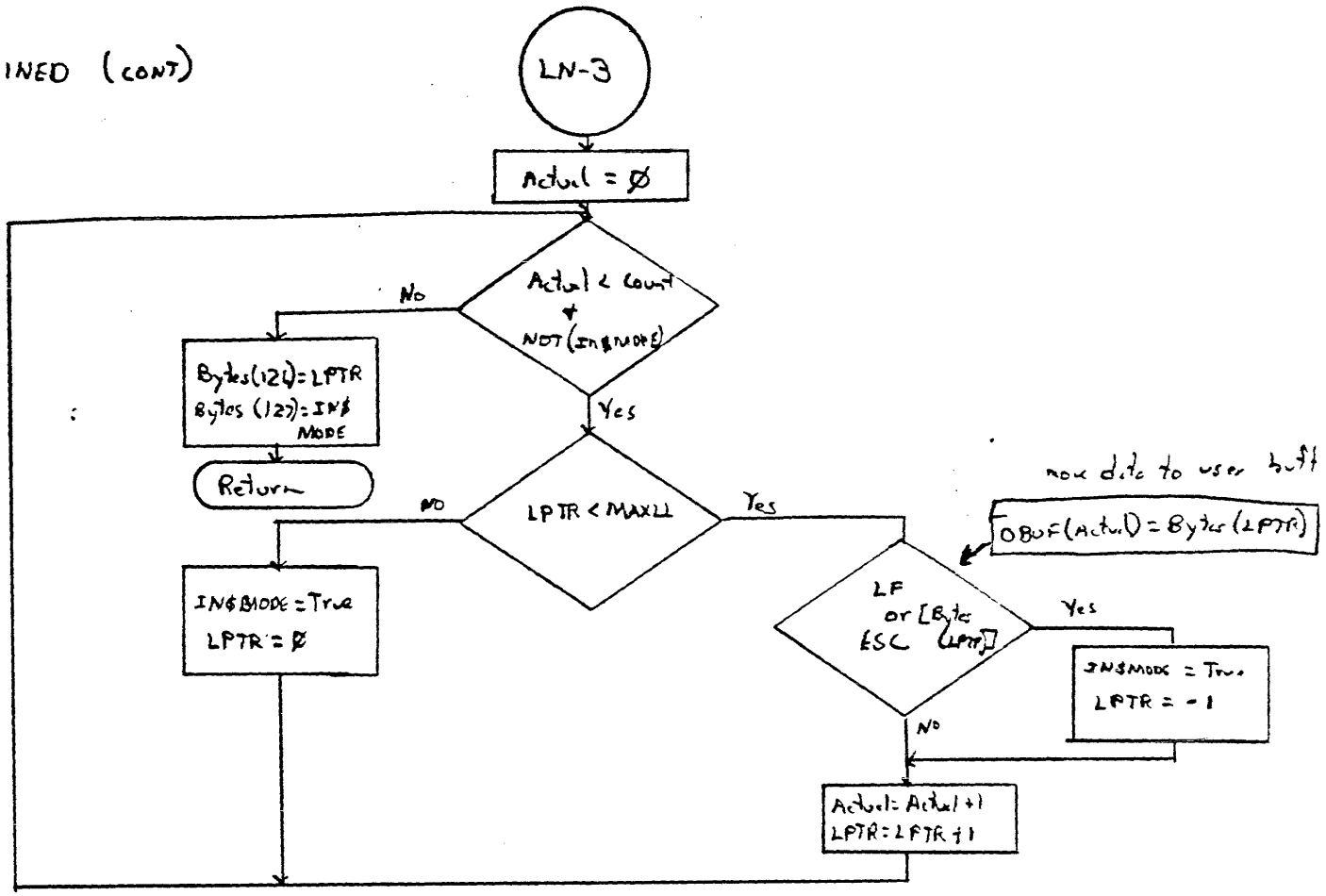
LINED


LINED
AFTN
Buffer
Count
Active
LitNMode = False
LBuff = @LineBuff
LPTR = @LineBuff
INMODE (125) := TRUE
set up by Open
LN-2
IN$MODE = True ?
No
Yes
LN-3
3
Read (AFTN, Char, 1, I
I = 0
No
Yes
AFTN = control-SHIFT
No
Yes
Char = control-Z
ABortx (Console EOF)
Byte(125) = False
Char = char & 7F#
No longer at EOF
Char <> LF
Yes
Echo (char)
Byte(Lptr) = char
NO
Selector = 0
Char = Special Character
Yes
Selector = Index of special char.
No
Selector = Sector + CMd LitNMode
LitNMode = FALSE
DO CASE Selector
Case 0, Non-1
Lptr = Lptr + 1
CASE 1, Robust
LPTR <> 0
No
Yes
Echo (Bell
Echo (LPTR := LPTR - 1)
LN-1
2

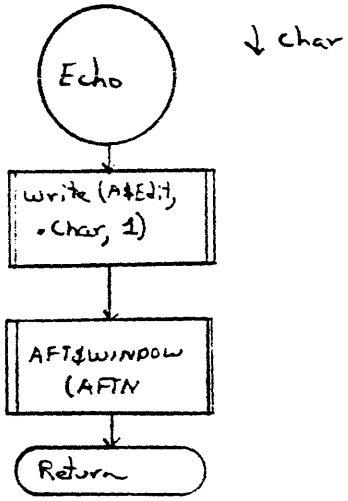
LINED (cont.)


LN-1
[DO CASE cont.]
CASE 2, Control-Z
Echo (CR)
Echo (LF)
Write (ASFDit, .Bytes, LPTR
Actual = Ø
Bytes(125) = True
ADR (62) = OFFSET
Case 3, Cntr-R
Echo (CR)
Echo (LF)
CASE 4, Cntr-X
LPTR = Ø
Echo ('#')
Echo (CR)
Echo (LF)
CASE 5, CR
Echo (LF)
LPTR = LPTR+1
Bytes(LPTR) = LF
INMODE = False
LPTR = Ø
CASE 6, LF
Echo (NULL)
Echo ('$')
INMODE = FALSE
LPTR = Ø
CASE 7, ESC
LPTR <>0
Yes
Echo (LF)
Bytes(LPTR) = LF
INMODE = FALSE
LPTR = Ø
CASE 8, Cntr-E
AFTN = AFTN$Console
CASE 9, cntr-9
Yes
Swap AFT(AFTN$Console$IN), Device with A(4) CIDEV
Echo ('^')
Echo ('E')
Literal$MODE = True
End Do Case
LN-2
1

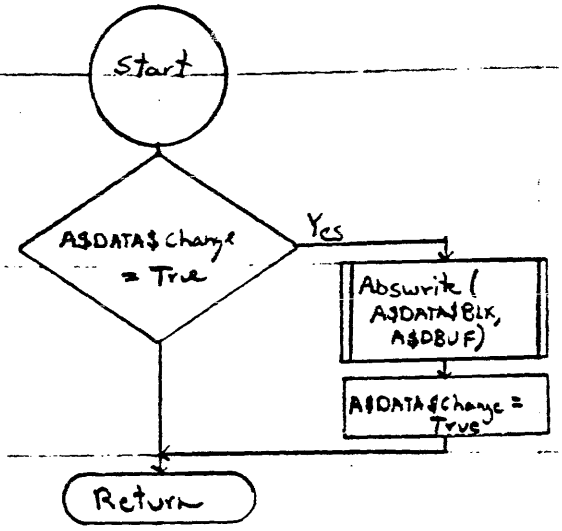
LINED (CONT)



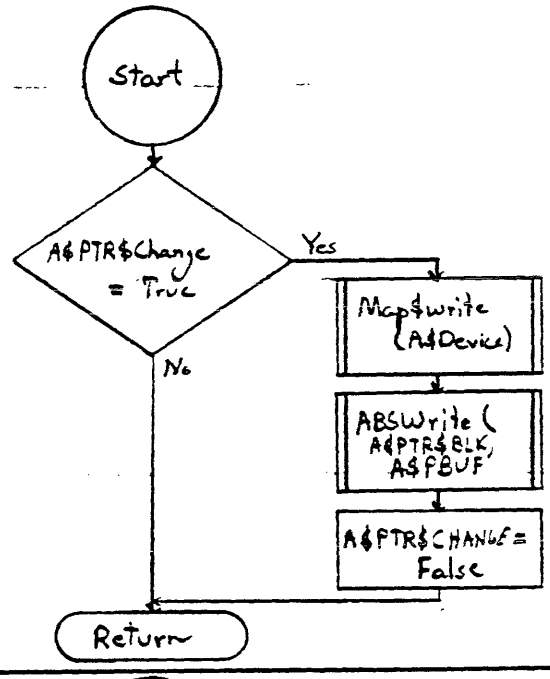
Echo



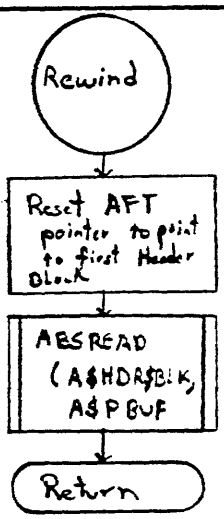
SAVE$DATA$BLK



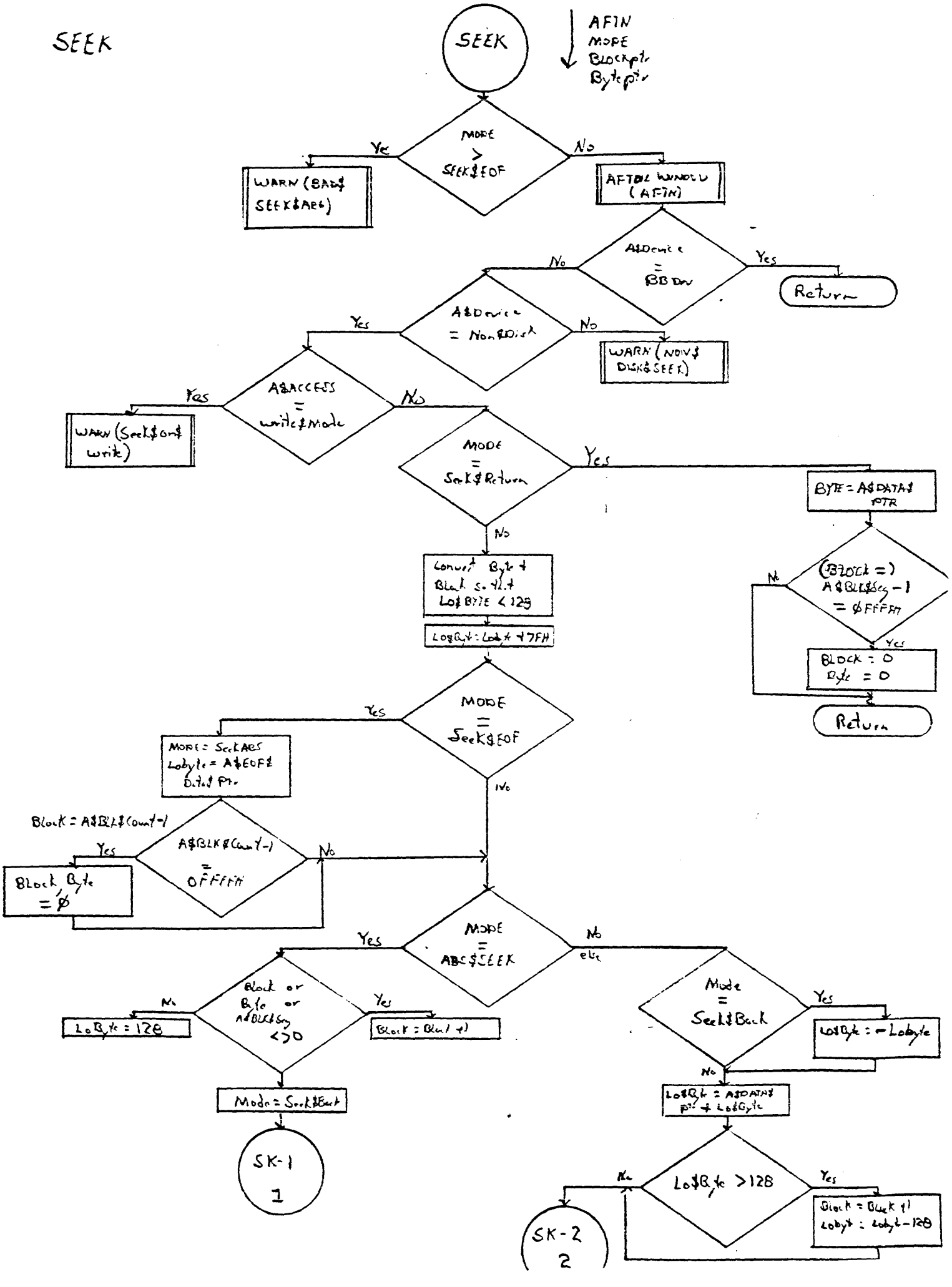
SAVE$POINTER



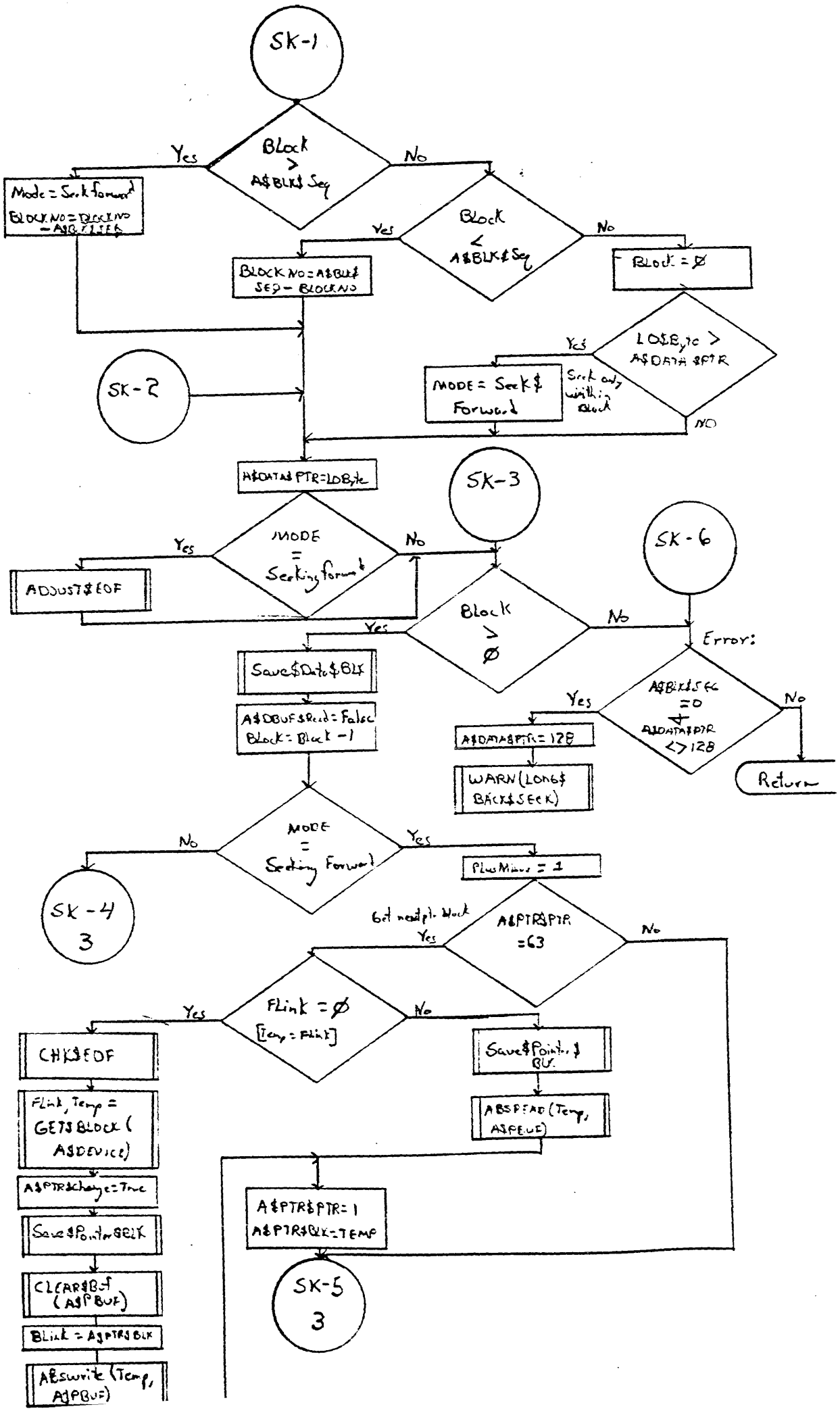
REWIND



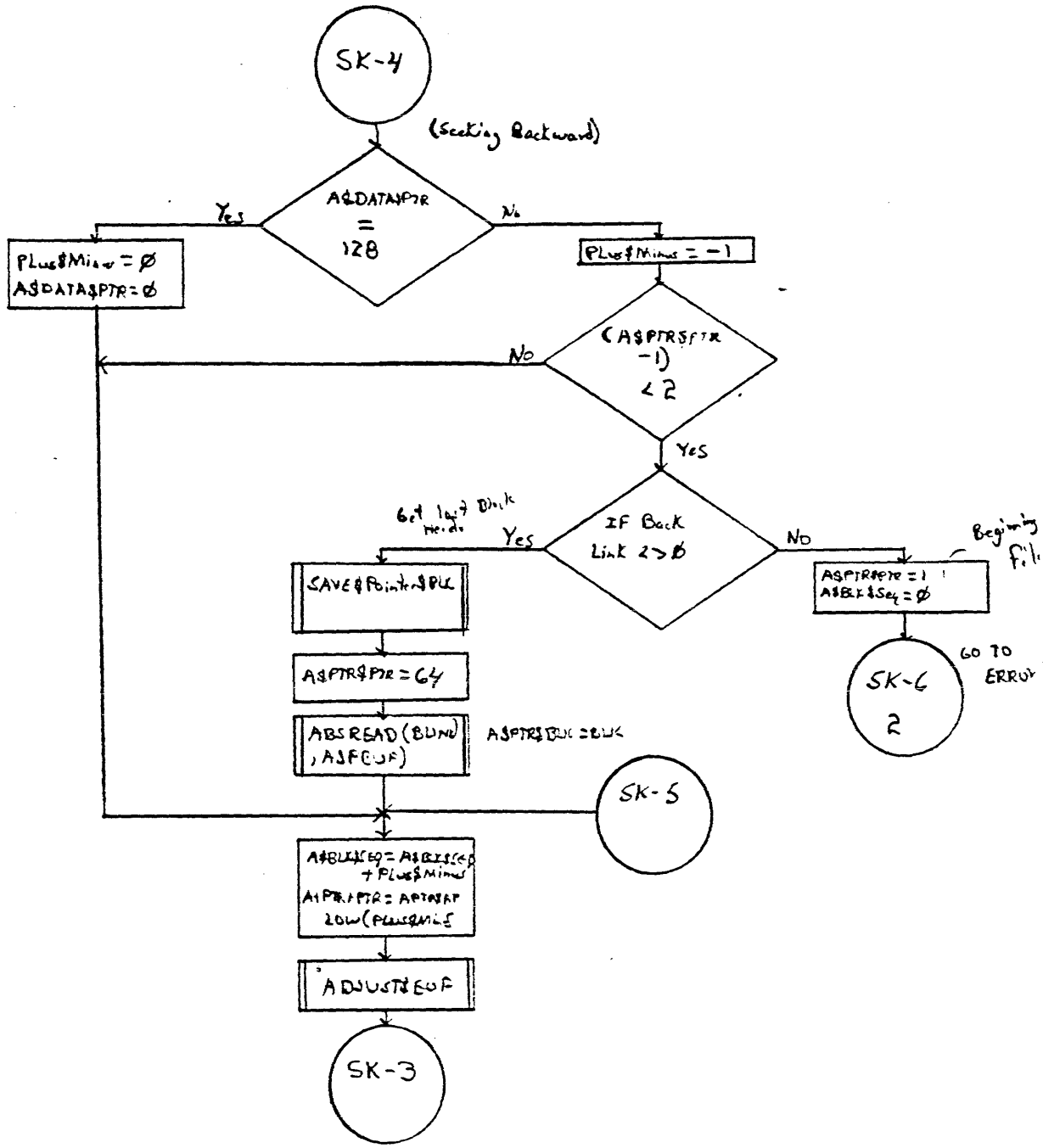
SEEK


SEEK
AFIN
MODE
Blockptr
Byteptr
MODE > SEEK$EOF
Yes
No
WARN (BAD$ SEEK$AREA)
AFTBL WINDOW (AFIN)
A$Device = BB DEV
No
Yes
Return
A$Device = Non$Disk
Yes
No
WARN (NOIV$ DISK$SEEK)
A$ACCESS = write$Mode
Yes
No
WARN (Seekon writ)
MODE = Seek$Return
Yes
No
BYTE = A$DATA$ PTR
(BBLOCK =) ABLKCnt - 1 = ØFFFFH
No
Yes
BLOCK = 0
Byte = 0
Return
Convert Byte & Block to 4 Lo$BYTE < 128
LogByt: Lobyt + 7FH
MODE = Seek$EOF
Yes
No
MODE = SeekABS
Lobyte = AEOF Data$ Ptr.
Block = ABLKCnt-1
ABLKCnt-1 = OFFFFH
Yes
No
Block Byte = Ø
MODE = ABC$SEEK
Yes
No
else
Block or Byte or ABLKCnt <>0
No
Yes
LoByte = 128
Block = Block + 1
Mode = Seek$Back
SK-1
1
Mode = Seek$Back
Yes
LogByt = -Lobyte
No
Lo$Byt = A$DATA$ Ptr + Lo$Byte
Lo$Byte > 128
No
Yes
Block = Block + 1
Lobyt = Lobyt - 128
SK-2
2

Seek (cont.)


SK-1
Block > ABLK Seq
Yes
No
Mode = Seek forward
BLOCK NO = BLOCK NO - ABLK SEQ
Block < ABLK Seq
Yes
No
BLOCK NO = ABLK SEQ - BLOCK NO
Block = Ø
LOB$ptr > A$DATA $PTR
Yes
No
MODE = Seek$ Forward
Seek only within Block
SK-2
A$DATA$PTR=LOB$ptr
SK-3
MODE = Seeking forward
Yes
No
ADJUST$EOF
SK-6
Block > Ø
Yes
No
Error:
ABLKSEQ =0 & A$DATA$PTR <> 128
Yes
No
A$DATA$PTR = 128
WARN(LONG$ BACK$SEEK)
Return
Save$Data$BLK
A$DBUF$Read = False
Block = Block - 1
MODE = Seeking Forward
No
Yes
SK-4 3
PlusMinus = 1
Get next/prev Block
APTRPTR =63
Yes
No
FLink = Ø [Temp = FLink]
Yes
No
CHK$EOF
FLink, Temp = GET$BLOCK (A$DEVICE)
APTRchange = True
Save$Pointer$BLK
CLEAR$Buf (A$PBUF)
BLink = APTRBLK
ABswrite (Temp, A$PBUF)
Save$Pointer$ BLK
ABSREAD (Temp, A$PBUF)
APTRPTR = 1
APTRBLK = TEMP
SK-5 3

Seek (cont.)


SK-4
(Seeking Backward)
A$DATA$PTR = 128
Yes
No
Plus$Minus = 0
A$DATA$PTR = 0
Plus$Minus = -1
(APTRPTR - 1) < 2
No
Yes
Got last Block needed
IF Back Link > 0
Yes
No
Beginning file
SAVE$Pointer$PUL
APTRPTR = 64
ABSREAD (BUFF, A$FCB)
APTRBUF = BUF
APTRPTR = 1
ABLKSeq = 0
SK-6
2
GO TO ERROR
SK-5
ABLKEQ = ABLKEQ + Plus$Minus
APTRPTR = APTRPTR + 2 * (Plus$Minus)
ADJUST$EOF
SK-3

```
LIST :F1:TOBOOT.MOD
LIST :F1:BUFFER.MOD
LIST :F1:JUMP  .M80
LIST :F1:TRAP  .M80
LIST :F1:JMPD  .M80
LIST :F1:AFT   .MOD
LIST :F1:ALLOC .MOD
LIST :F1:CLOSE .MOD
LIST :F1:ATTRIB.MOD
LIST :F1:ISIS  .MOD
LIST :F1:EXIT  .MOD
LIST :F1:PATH  .MOD
LIST :F1:ABORT .MOD
LIST :F1:OPEN  .MOD
LIST :F1:CONSOL.MOD
LIST :F1:ERROR .MOD
LIST :F1:DELETE.MOD
LIST :F1:RENAME.MOD
LIST :F1:RESCAN.MOD
LIST :F1:DISK  .MOD
LIST :F1:LOC62 .M80
LIST :F1:LOAD  .MOD
LIST :F1:DISK1 .M80
LIST :F1:DIRECT.MOD
LIST :F1:DISK2 .M80
LIST :F1:RW    .MOD
LIST :F1:SEEK  .MOD
END JOB
```

/*

THE PURPOSE OF THIS DOCUMENT IS TO BRIEFLY DESCRIBE THE OPERATION OF THE ISIS-II OPERATING SYSTEM, VERSION 3.

IT IS INTENDED THAT THIS DOCUMENT BE THE FIRST ONE ON THE LISTINGS OF THE RESIDENT ISIS SOURCE MODULES TRANSMITTED TO PRODUCT ENGINEERING. SIMILAR DOCUMENTS COVERING CUSP SOURCE MODULES AND CSPLIB SOURCE MODULES SHOULD APPEAR AS THE FIRST LISTINGS ON THEIR RESPECTIVE LISTINGS.

PLEASE REFER TO THE ISIS-II VERSION 3 ERS (EXTERNAL REFERENCE SPECIFICATION) AN TO THE ACTUAL SOURCE MODULES THEMSELVES FOR MORE DETAILED DESCRIPTION OF THE SYSTEM.

****

1. THE ISIS-II OPERATING SYSTEM MAY BE VIEWED AS A COLLECTION OF ROUTINES, SOME OF WHICH ARE ALWAYS PRESENT IN MEMORY AND THE REST OF WHICH ARE LOCATED ON THE SYSTEM DISKETTE (IN DRIVE 0) AND ARE CALLED IN AS NEEDED.
   A. THE RESIDENT PORTION OF ISIS, REFERRED TO AS THE "COLONEL", IS LOCATED IN THE FIRST 12K OF MDS/EMDS MEMORY (I.E. 0 TO 2FFFH). IT IS COMPOSED OF 20 PLM80 MODULES (ABORT, AFT, ALLOC, ATTRIB, BUFFER, CLOSE, CONSOL, DELETE, DIRECT, DISK, ERROR, EXIT, ISIS, LOAD, OPEN, PATH, RENAME, RESCAN, RW, SEEK) AND 5 ASM80 MODULES (DISK1, DISK2, JUMP, LOC62, TRAP). SPATH WHOCON
   B. THE NONRESIDENT PORTION OF ISIS, REFERRED TO AS THE "CUSPS", IS LOCATED ON THE SYSTEM DISKETTE. WHEN LOADED INTO MDS/EMDS MEMORY, A CUSP WILL OCCUPY MEMORY ABOVE THE ISIS BUFFER SPACE (THE ISIS BUFFER SPACE IS A CONTIGUOUS SECTION OF MEMORY BEGINNING AT 3000H AND EXTENDING UPWARDS IN 128 BYTE INCREMENTS). THE CUSPS CONSIST OF 13 MUTUALLY INDEPENDENT PLM80 MODULES (ATTRIB, BINOBJ, CLI, COPY, DELETE, DIR, EDIT, FORMAT, HEXOBJ, IDISK, OBJHEX, RENAME, SUBMIT). WITH THE EXCEPTION OF CLI, EACH CUSP CORRESPONDS TO AN ISIS SYSTEM COMMAND KNOWN TO THE USER; TYPING IN THE COMMAND ON THE CONSOLE DEVICE WILL CAUSE THE LOADING OF THE CORRESPONDING CUSP. CLI, THE COMMAND LANGUAGE INTERPRETER, IS RESPONSIBLE FOR PARSING AND ACTING ON INPUT TYPED BY THE USER ON THE CONSOLE; IT IS LOADED INTO THE MEMORY AS A RESULT OF ACTION TAKEN BY THE EXIT ROUTINE IN RESIDENT ISIS.
   C. IN ADDITION TO THE ABOVE, THE ISIS SYSTEM IS COMPOSED OF THE BOOTSTRAP LOADER. THE BOOTSTRAP IS COMPRISED OF A PLM80 MODULE CALLED T0BOOT (FOR TRACK 0 BOOT, THE TRACK NUMBER ON THE SYSTEM DISKETTE UPON WHICH THE FILE RESIDES) AND AN ASM80 MODULE CALLED JMPD. THE JMPD MODULE PLACES INTO LOCATION 3000H A JUMP INSTRUCTION TO T0BOOT (WHICH THE LOCATE PROGRAM PLACES AT 3200H). THE T0BOOT MODULE DOES THE ACTUAL BOOTING; ITS OPERATION IS DESCRIBED IN SECTION 2.C BELOW.

* circled items avail to user through jump to loc 90 where param. are picked up

2. THE ISIS-II OPERATING SYSTEM IS LOADED INTO MEMORY AS FOLLOWS:
   A. WHEN THE MDS/EMDS MONITOR/BOOTSTRAP OPERATION IS INITIATED, THE FIRST 26 SECTORS OF TRACK Ø ON THE SYSTEM DISKETTE ARE READ IN AND LOADED INTO MEMORY AT ADDRESS 3ØØØH AND UPWARDS.
   B. THE MDS/EMDS BOOT THEN TRANSFERS CONTROL TO THE ISIS TØBOOT (LOCATED AT 3ØØØH). THE ISIS TØBOOT IN TURN LOADS THE RESIDENT PORTION OF ISIS, CALLED ISIS.BIN (BECAUSE IT IS IN BINARY OBJECT FORMAT). TØBOOT ALSO DETERMINES THE EXACT DISK CONFIGURATION OF THE SYSTEM (I.E. NUMBER OF DRIVES, TYPES OF CONTROLLERS) AND PASSES THIS INFORMATION ON TO RESIDENT ISIS IN THE FORM OF THE 6-BYTE ARRAY DKCFTB (DISK CONFIGURATION TABLE).
   C. TØBOOT THEN TRANSFERS CONTROL TO ISIS VIA A CALL TO EXIT, A CALL FROM WHICH IT NEVER RETURNS. THE EXIT SYSTEM CALL IN TURN CAUSES THE LOADING AND EXECUTION OF THE COMMAND LANGUAGE INTERPRETER (CLI). CLI WILL SIT PATIENTLY WAITING FOR USER INPUT FROM THE CONSOLE EITHER IN THE FORM OF A SYSTEM COMMAND OR THE NAME OF A USER OBJECT FILE (WHICH IT WILL CAUSE TO BE LOADED AND EXECUTED).

3. RESIDENT ISIS CONSISTS OF MODULES WHICH IMPLEMENT THE ISIS SYSTEM CALLS (THE ISIS SYSTEM CALLS ARE: OPEN,READ,WRITE,SEEK,RESCAN,CLOSE, SPATH,DELETE,RENAME,ATTRIB,CONSOL,WHOCON,ERROR,LOAD,EXIT) AND MODULES WHICH ARE REQUIRED FOR THE CORRECT OPERATION OF THE SYSTEM (SUCH AS DIRECTORY MAINTENANCE, DISK STORAGE ALLOCATION AND DEALLOCATION, BUFFER ALLOCATION, AND ERROR RECOVERY).
   A. THE ISIS SYSTEM CALLS ARE CALLS TO LOCATION 4ØH. AT LOCATION 4ØH IS A JUMP TO THE MODULE CALLED ISIS. THIS MODULE COPIES THE PARAMETERS USED IN THE SYSTEM CALL, EXCHANGES STACK POINTERS (ISIS MAINTAINS A STACK POINTER), AND THEN CALLS THE MODULE RESPONSIBLE FOR CARRYING OUT THAT PARTICULAR SYSTEM CALL. UPON RETURN FROM THAT MODULE, THE ISIS STACK POINTER AND USER STACK POINTER ARE AGAIN EXCHANGED, AND CONTROL IS RETURNED TO THE USER PROGRAM (WHICH MAY BE A CUSP).
   B. WITHIN RESIDENT ISIS, THE MODULES CAN AND DO CALL EACH OTHER.

*/

5 ISIS system calls

```
ISIS-II PL/M-80 V3.1 COMPILATION OF MODULE PATH
OBJECT MODULE PLACED IN :F1:PATH.OBJ
COMPILER INVOKED BY:  PLM80 :F1:PATH.MOD PRINT(:F3:PATH.LST) DEBUG DATE(03-02-78)  IXREF(:F3:PATH.IXI)


  1         PATH:
            DO;
            /*
            ***************************************************************
            ***************************************************************

                              MODULE NAME PATH
                              ====== ==== ====

            ABSTRACT
            ========

               THIS IS THE MODULE THAT UNDERSTANDS THE SYNTAX OF
            PATHNAMES, AND PARSES THEM.

            MODULE ORGANIZATION
            ====== ============

               THE MODULE CONTAINS THE FOLLOWING COMPONENTS:

            1.  2 GLOBAL ARRAYS, 'PN' AND 'PN2', EITHER OF WHICH IS
                SUFFICIENT TO CONTAIN THE "INTERNAL FORM" OF A PATHNAME.
                THESE ARE PROVIDED HERE FOR THE CONVENIENT TRANSITORY
                USE BY SUCH SUBROUTINES AS OPEN, RENAME, DELETE, ETC.
                THEY ARE NOT DIRECTLY USED BY 'PATH'.
            2.  THE BYTE PROCEDURE 'PATH', WHICH CONVERTS A LEGAL PATHNAME
                INTO INTERNAL FORM, RETURNING AN ERROR NUMBER FOR THE
                TYPE OF PATHNAME FOUND.  (0 MEANS LEGAL PATHNAME, OTHER
                NUMBERS INDICATES SYNTAX ERRORS IN PATHNAME.)
            3.  THE PROCEDURE 'XPATH', WHOSE ONLY REASON FOR EXISTENCE IS
                TO REDUCE THE SPACE USED FOR CALLS TO 'PATH' THROUGHOUT
                THE PROGRAM; IT REDUCES CODE TO TEST THE RESULT FROM
                PATH AND OPTIONALLY CALL ERR, TO A SINGLE POINT IN THE
                PROGRAM.

            CALLING GRAPH
            ======= =====

            >>PATH
            >>XPATH
                'PATH'
                ERR  (ERROR)

            GLOBAL VARIABLES ACCESSED
            ====== ========= ========

             -> PN
             -> PN2

            GLOBAL VARIABLES MODIFIED
            ====== ========= ========
```

```
ATTRIB:
DO;
/*$NOLIST*/
$INCLUDE (:F2:COMMON.LIT)
$INCLUDE (:F2:ATTRIB.LIT)
$INCLUDE (:F2:AFT.LIT)
$INCLUDE (:F2:ERROR.MEX)
$INCLUDE (:F2:PATH.MEX)
$INCLUDE (:F2:DEVICE.LIT)
$INCLUDE (:F2:ERROR.LIT)
$INCLUDE (:F2:AFT.MEX)
$INCLUDE (:F2:DIRECT.MEX)
$INCLUDE (:F2:RW.MEX)
$LIST

ATTRIB:
  PROCEDURE(FILE$PTR,SWID,VALUE) PUBLIC;
    DECLARE (FILE$PTR,SWID) ADDRESS;
    DECLARE VALUE BOOLEAN;
    DECLARE MASK(*) BYTE DATA (INVISIBLE$ATTRIBUTE,
                               SYSTEM$ATTRIBUTE,
                               WRITEP$ATTRIBUTE,
                               FORMAT$ATTRIBUTE);

    IF SWID > 4 THEN CALL WARN(BAD$ATTRIB);
    CALL XPATH(FILE$PTR,.PN);
    IF PN(0) > F5DEV THEN CALL WARN(NO$DISK$FILE);  /* DO */
    CALL AFT$WINDOW(PN(0));
    IF NOT DLOOK(.PN) THEN CALL WARN(NO$SUCH$FILE);
    DIRECT.ATTRIB = DIRECT.ATTRIB OR MASK(LOW(SWID));
    IF NOT VALUE THEN
      DIRECT.ATTRIB = DIRECT.ATTRIB AND (NOT MASK(LOW(SWID)));
    CALL DIR$CLOS(PN(0));
  END ATTRIB;
END;

EOF
```

```
BUFFER:
DO;
 /* ***************************************************************
    ***************************************************************

                        MODULE NAME BUFFER
                        ====== ==== ======

    ABSTRACT
    ========

        THIS MODULE CONTAINS ROUTINES FOR OBTAINING, CLEARING,
    RELEASING, AND PACKING BUFFERS OF 128 BYTES EACH.

    MODULE ORGANIZATION
    ====== ============

        THIS MODULE CONTAINS AN ALLOCATION TABLE ('BUFFER$TABLE'), AND
    4 SUBROUTINES.
        BUFFERS ARE ALOCATED IN RAM, STARTING AT ".MEMORY", AND
    WORKING UPWARDS (TO THE USER'S PROGRAM ORIGIN POINT).
        'BUFFER$TABLE' CONTAINS 19 ENTRIES, ALLOWING MAXIMUM BUFFER
    USAGE FOR 6 OPEN FILES (ALL LINED INPUT FILES, 3 BUFFERS EACH),
    PLUS 1 BUFFER FOR THE CONSOLE.

    CALLING GRAPH
    ======= =====

    >>CLEAR$BUF
    >>RETURN$BUF
    >>GET$BUF
        ERR (ERROR)
    >>PACK$AFT$BUF
        GET$BUF
        RETURN$BUF

    GLOBAL VARIABLES ACCESSED
    ====== ========= ========

        MEMORY  (PL/M PRE-DECLARED VARIABLE)

    INVARIANTS
    ==========

        EACH ENTRY IN 'BUFFER$TABLE' IS INITIALIZED TO '0'; LEGAL
    VALUES ARE '0', '1' AND '2', WITH THE FOLLOWING MEANINGS:
        0 - CORRESPONDING BUFFER IS AVAILABLE, BUT NOT ALLOCATED.
        1 - CORRESPONDING BUFFER IS NOT AVAILABLE, BECAUSE IT
            IS IN THE CURRENT USER RAM AREA.
        2 - CORRESPONDING BUFFER IS IN USE.

        ENTRIES CHANGE WHEN BUFFERS ARE OBTAINED OR RELEASED (VIA
    GET$BUF AND RETURN$BUF), OR WHEN THE THE USER'S PROGRAM ORIGIN
    POINT IS CHANGED (BY MAKING A CALL TO THE 'LOAD' SUBROUTINE).
 */
/*$NOLIST*/
$INCLUDE (:F2:COMMON.LIT)
$INCLUDE (:F2:AFT.LIT)
$INCLUDE (:F2:ERROR.LIT)
$INCLUDE (:F2:ERROR.MEX)
$INCLUDE (:F2:AFT.MEX)
$LIST
```

```
DECLARE BUFFER$TABLE(19) BYTE PUBLIC INITIAL
     (0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0);

CLEAR$BUF:
  PROCEDURE (BUF$ADDRESS) PUBLIC;
    DECLARE BUF$ADDRESS ADDRESS;
    DECLARE BUFFER BASED BUF$ADDRESS (128) BYTE;
    DECLARE I BYTE;

    DO I=0 TO 127;
      BUFFER(I) = 0;
    END;
  END CLEAR$BUF;

RETURN$BUF:
  PROCEDURE (BUF$ADDRESS) PUBLIC;
    DECLARE BUF$ADDRESS ADDRESS;

    BUFFER$TABLE(SHR(BUF$ADDRESS - .MEMORY,7)) = 0;
  END RETURN$BUF;

GET$BUF:
  PROCEDURE ADDRESS PUBLIC;
    DECLARE I BYTE;

    DO I = 0 TO LAST(BUFFER$TABLE);
      IF BUFFER$TABLE(I) = 0 THEN
      DO;
        BUFFER$TABLE(I) = 2;
        RETURN .MEMORY + SHL(DOUBLE(I),7);
      END;
    END;
    CALL ERR(ABORT,MD$FREE$BUFFER);
  END GET$BUF;

 /* ***********************************************************************

                        PROCEDURE PACK$AFT$BUF
                        --------- ------------

    ABSTRACT
    --------

        THIS ROUTINE REASSIGNS BUFFER SPACE, AND MOVES BUFFER
    CONTENTS AS NECESSARY, TO ENSURE THAT THERE IS NO UNUSED
    BUFFER SPACE BELOW THE AREA OCCUPIED BY THE HIGHEST BUFFER
    IN USE.

    PARAMETERS
    ----------

    NONE

    VALUE RETURNED
    ----- --------

        THE ADDRESS OF THE FIRST (LOWEST) BYTE IN THE FIRST
    (LOWEST) UNALLOCATED BUFFER.  THIS INFORMATION IS THE "TOP OF
    BUFFER AREA" DESCRIBED IN THE MDS-DOS OPERATOR'S MANUAL.

    GLOBAL VARIABLES ACCESSED
    ------ --------- --------
```

```
    AFT.LBUF
    AFT.DBUF
    AFT.PBUF

    GLOBAL VARIABLES MODIFIED
    ------ --------- --------

    AFT.LBUF
    AFT.PBUF
    AFT.DBUF

    DESCRIPTION
    -----------

        FOR EVERY AFT ENTRY WHICH IS A CURRENTLY ASSIGNED
    BUFFER ADDRESS, THE BUFFER IS RETURNED (USING 'RETURN$BUF'),
    AND IS REPLACED BY A NEW BUFFER (USING 'GET$BUF').  IF THE NEW
    BUFFER IS A DIFFERENT ONE THAT THE ONE RETURNED, THEN THE
    DATA FROM THE OLD IS MOVED INTO THE NEW.
 */

PACK$AFT$BUF:
  PROCEDURE ADDRESS PUBLIC;
    DECLARE (I,J,INDEX) BYTE;
    DECLARE BUFFER$TOP ADDRESS;
    DECLARE BPTR ADDRESS, BUFADR BASED BPTR ADDRESS;
    DECLARE NEW$BADR ADDRESS, NEW$DATUM BASED NEW$BADR(128) BYTE,
            OLD$BADR ADDRESS, OLD$DATUM BASED OLD$BADR(128) BYTE;

    BUFFER$TOP = .MEMORY;
    DO I = AFT$BOTTOM TO AFT$LAST;
      IF NOT AFT(I).EMPTY THEN
      DO INDEX = 0 TO 2;
        BPTR = .AFT(I).PBUF - 2*INDEX;
        J = SHR(BUFADR - .MEMORY,7);
        IF J <= LAST(BUFFER$TABLE) AND BUFFER$TABLE(J) = 2 THEN
             DO;
             OLD$BADR = BUFADR;
             CALL RETURN$BUF(OLD$BADR);
             NEW$BADR = GET$BUF;
             IF BUFFER$TOP < NEW$BADR THEN BUFFER$TOP = NEW$BADR;
             IF NEW$BADR <> OLD$BADR THEN
               DO;
                 DO J = 0 TO 127;
                    NEW$DATUM(J) = OLD$DATUM(J);
                 END;
                 BUFADR = NEW$BADR;
               END;
             END;
      END; /* OF LOOP TO HANDLE EACH BUFFER P,D,L */
    END; /* OF LOOP TO TRAVERSE AFT */
    RETURN BUFFER$TOP + 128;
  END PACK$AFT$BUF;

END;

EOF
```

```
CONSOL:
DO;
/*$NOLIST*/
$INCLUDE (:F2:COMMON.LIT)
$INCLUDE (:F2:ERROR.LIT)
$INCLUDE (:F2:AFT.LIT)
$INCLUDE (:F2:OPEN.LIT)
$INCLUDE (:F2:DEVICE.LIT)
$INCLUDE (:F2:ERROR.MEX)
$INCLUDE (:F2:AFT.MEX)
$INCLUDE (:F2:PATH.MEX)
$INCLUDE (:F2:UNPATH.PEX)
$INCLUDE (:F2:OPEN.MEX)
$INCLUDE (:F2:CLOSE.MEX)
$LIST

DECLARE CUR$CONSOL$IN  (12) BYTE PUBLIC,
        CUR$CONSOL$OUT (12) BYTE PUBLIC;

DECLARE COLD$CONSOL$IN(5)  BYTE INITIAL (':XI: '),
        COLD$CONSOL$OUT(5) BYTE INITIAL (':XO: ');

DECLARE COLD$START$FLAG BOOLEAN INITIAL (TRUE);

DECLARE (COLD$CIDEV,ALT$CIDEV) BYTE PUBLIC;

CONSOLE:
  PROCEDURE (INFILE,OUTFILE) PUBLIC;
    DECLARE (INFILE,OUTFILE) ADDRESS;
    DECLARE INSTRING BASED INFILE BYTE,
            OUTSTRING BASED OUTFILE BYTE;
    DECLARE TEMP BYTE;
    DECLARE TORV(2) BYTE
      DATA ('TV'); /* T OR V IS 1ST LETTER OF DEVICE NAME */
    DECLARE INITIO BYTE AT (6);

    IF COLD$START$FLAG THEN
    DO;
      COLD$CONSOL$IN(1),COLD$CONSOL$OUT(1) = TORV(INITIO AND 1);
      INFILE = .COLD$CONSOL$IN;
      OUTFILE= .COLD$CONSOL$OUT;
    END;
    GLOBAL$SEVERITY = ABORT;
    CALL XPATH(OUTFILE,.PN);
    IF PN(0) <> CODEV THEN
    DO;
      CALL CLOSE(AFTN$CONSOL$OUT);
      TEMP = OPEN(OUTFILE,WRITE$MODE,FALSE);
      CALL XPATH(OUTFILE,.CUR$CONSOL$OUT);
    END;
    CALL XPATH(INFILE,.PN);
    IF PN(0) <> CIDEV THEN
    DO;
      CALL CLOSE(AFTN$CONSOL$IN);
      TEMP = OPEN(INFILE,READ$MODE,100H);
      IF COLD$START$FLAG THEN
        COLD$CIDEV = AFT(AFTN$CONSOL$IN).DEVICE;
      ALT$CIDEV = COLD$CIDEV;
      CALL XPATH(INFILE,.CUR$CONSOL$IN);
    END;
    COLD$START$FLAG = FALSE;
  END CONSOLE;
```

```
UNDCON:
  PROCEDURE(AFTN,BUFFERLOC) PUBLIC;
    DECLARE AFTN BYTE;
    DECLARE BUFFERLOC ADDRESS;
    DECLARE NAMELOC ADDRESS;

    NAMELOC = .CUR$CONSOL$OUT;
    IF AFTN THEN NAMELOC = .CUR$CONSOL$IN;
    CALL UNPATH(NAMELOC,BUFFERLOC);
  END UNDCON;
END;

EOF
```

```
DELETE:
DO;
 /* ***********************************************************
    ***********************************************************

                        MODULE NAME DELETE
                        ====== ==== ======

    ABSTRACT
    ========

    THIS MODULE CONSISTS OF PROCEDURES 'DEL' AND 'DELETE'.

    CALLING GRAPH
    ======= =====

    >>DELETE
        'XPATH' (PATH)
        'ERR' (ERROR)
        'AFT$WINDOW' (AFT)
        'DLOOK' (DIRECT)
    >>DEL
        'AFT$WINDOW' (AFT)
        'WRITE' (RW)
        'ABSID' (DISK)
        'SEEK' (SEEK)
        'FREE$BLOCK' (ALLOC)
        'MAPRW' (ALLOC)

    GLOBAL VARIABLES ACCESSED
    ====== ========= ========

    AFTN$DIRECT (LIT) -- DEL
    A$DATA$BLK (AFT) -- DEL
    A$DBUF (AFT) -- DEL

    GLOBAL VARIABLES MODIFIED
    ====== ========= ========

    $$$$
 */
/*$NOLIST*/
$INCLUDE (:F2:COMMON.LIT)
$INCLUDE (:F2:AFT.LIT)
$INCLUDE (:F2:DISK.LIT)
$INCLUDE (:F2:SEEK.LIT)
$INCLUDE (:F2:DEVICE.LIT)
$INCLUDE (:F2:ERROR.LIT)
$INCLUDE (:F2:ATTRIB.LIT)
$INCLUDE (:F2:DIRECT.MEX)
$INCLUDE (:F2:AFT.MEX)
$INCLUDE (:F2:PATH.MEX)
$INCLUDE (:F2:DISK.MEX)
$INCLUDE (:F2:ALLOC.MEX)
$INCLUDE (:F2:ERROR.MEX)
$INCLUDE (:F2:RW.MEX)
$INCLUDE (:F2:SEEK.MEX)
$LIST

DEL:
  PROCEDURE(DISKNUM) PUBLIC;
    DECLARE DISKNUM BYTE;
    DECLARE (AFTN,I) BYTE;
```

```
    DECLARE BLOCKNO ADDRESS DATA (0);
    DECLARE BYTENO ADDRESS DATA(16 /* SIZE(DIRECT) */);
    DECLARE TEMP ADDRESS;
    DECLARE BUF BASED TEMP (64) ADDRESS;
    DECLARE FLINK LITERALLY '1';

    /* INTERNAL DELETE ROUTINE
        DISKNUM = F0DEV, F1DEV, F2DEV, F3DEV, F4DEV, OR F5DEV
        DIRECT MUST CONTAIN THE DIRECTORY ENTRY FOR AN EXISTING
      FILE ON DISK.
        AFT SLOT FOR THE APPROPRIATE DIRECTORY MUST HAVE BUFFERS;
       AND THESE BUFFERS MUST BE SETUP FOR DIRECTORY READING AND
      WRITING. THESE BUFFERS WILL BE CLOBBERED.
    */

    DIRECT.EMPTY = TRUE;
    CALL AFTSWINDOW(AFTN:=AFTNSDIRECT+DISKNUM);
    CALL WRITE(AFTN,.DIRECT,SIZE(DIRECT));
      CALL ABSIO(WRITESCOMMAND,DISKNUM,ASDATASBLK,ASDBUF);
    CALL SEEK(AFTN,SEEKSBACK,.BLOCKNO,.BYTENO);
    TEMP = ASDBUF; /* TEMP -> A BUFFER WE KNOW EXISTS */
    DO WHILE DIRECT.HDRSBLK <> 0;
      CALL FREESBLOCK(DISKNUM,DIRECT.HDRSBLK);
      CALL ABSIO(READSCOMMAND,DISKNUM,DIRECT.HDRSBLK,TEMP);
      DIRECT.HDRSBLK = BUF(FLINK);
      DO I=2 TO 63;
        IF BUF(I)<>0 THEN CALL FREESBLOCK(DISKNUM,BUF(I));
      END;
    END;
    CALL ABSIO(READSCOMMAND,DISKNUM,ASDATASBLK,ASDBUF);
    CALL MAPSWRITE(DISKNUM);
  END DEL;

DELETE:
  PROCEDURE (PATHNAME) PUBLIC;
    DECLARE PATHNAME ADDRESS;
    DECLARE I BYTE;

    CALL XPATH(PATHNAME,.PN);
    IF PN(0) > F5DEV THEN CALL WARN(NONSDISKSFILE);  /* DO */
    CALL AFTSWINDOW(PN(0));
    IF NOT DLOOK(.PN) THEN CALL WARN(NOSSUCHSFILE);
      IF (DIRECT.ATTRIB AND (WRITEPSATTRIBUTE OR FORMATSATTRIBUTE)) <> 0
      THEN CALL WARN(WRITESPROTECT);
        DO I = AFTSBOTTOM TO AFTSLAST;
          CALL AFTSWINDOW(I);
          IF  (NOT ASEMPTY)
          AND (ASDEVICE = PN(0))
          AND (ASISNO = DIRECTSISNO)
          THEN CALL WARN(CANTSDELETE);
        END;
        CALL DEL(PN(0));
  END DELETE;
END;

EOF
```

```
DIRECT:
DO;
 /* ***************************************************************
    ***************************************************************

                        MODULE NAME DIRECTORY
                        ====== ==== =========

    ABSTRACT
    ========

        ALL KNOWLEDGE ABOUT THE FORMAT AND LOCATION OF A DISKETTE
    DIRECTORY IS CONTAINED BY THIS MODULE.

    MODULE ORGANIZATION
    ====== ============

        THE MODULE CONTAINS A DATA AREA, WHICH NORMALLY CONTAINS
    THE INFORMATION FROM A SINGLE ENTRY IN A DISKETTE DIRECTORY,
    AND THE PROCEDURES 'DIR$CLOSE' AND 'DLOOK'.

    CALLING GRAPH
    ======= =====

    >>DIR$CLOSE
        ABSWRITE (DISK)
    >>DLOOK
        GET$BUF (BUFFER)
        AFT$WINDOW (AFT)
        REWIND (SEEK)
        READ (RW)
        SEEK (SEEK)
        RETURN$BUF (BUFFER)

    GLOBAL VARIABLES ACCESSED
    ====== ========= ========

    ASDATA$BLK (AFT) -- DIR$CLOSE
    ASDBUF (AFT) -- DIR$CLOSE
    ASDEVICE (AFT) -- DLOOK

    GLOBAL VARIABLES MODIFIED
    ====== ========= ========

    ASDBUF (AFT) -- DLOOK
    ASPBUF (AFT) -- DLOOK
    DIRECT (ALL 16 BYTES) -- DLOOK

    INVARIANTS
    ==========

        ON EVERY DISKETTE, THE DIRECTORY FILE IS PARTITIONED INTO
    "ENTRIES" CONTAINING 16 BYTES OF DATA EACH.
     THE VARIOUS BYTE- AND ADDRESS-VARIABLES WITHIN AN ENTRY ARE
     GIVEN DESCRIPTIVE NAMES (SEE THE STRUCTURE 'DIRECT' BELOW).
          AT ALL TIMES, THE FOLLOWING MUST BE TRUE:

    1.  DIRECT.EMPTY = FALSE => THE ENTRY CONTAINS MEANINGFUL DATA.
    2.  (DIRECT.EMPTY = TRUE(0FFH)) => THE DIRECTORY SLOT IS EMPTY.
    3.  (DIRECT.EMPTY = TRUE(07FH)) => THE DIRECTORY SLOT, AND ALL
    FURTHER DIRECTORY SLOTS, ARE EMPTY.
 */
/*$NOLIST*/
```

```
$INCLUDE (:F2:COMMON.LIT)
$INCLUDE (:F2:SEEK.LIT)
$INCLUDE (:F2:AFT.LIT)
$INCLUDE (:F2:AFT.MEX)
$INCLUDE (:F2:ALLOC.MEX)
$INCLUDE (:F2:DISK.MEX)
$INCLUDE (:F2:RW.MEX)
$INCLUDE (:F2:BUFFER.MEX)
$INCLUDE (:F2:SEEK.MEX)
$LIST

DECLARE DIRECT STRUCTURE (
        EMPTY     BOOLEAN, /* FLAG TO INDICATE WHETHER DIRECTORY
                              ENTRY IS USED */
        FILE(6)   BYTE,    /* FILE NAME */
        EXT(3)    BYTE,    /* EXTENSION */
        ATTRIB    BYTE,    /* FILE ATTRIBUTES */
        EOF$COUNT BYTE,    /* CHARACTER COUNT, LAST DATA BLOCK */
        BLK$COUNT ADDRESS, /* NUMBER OF BLOCKS IN FILE */
        HDR$BLK   ADDRESS) /* ADDRESS OF FIRST POINTER BLOCK */
        PUBLIC;

DECLARE DIRECT$ISND BYTE PUBLIC; /* DIRECTORY ENTRY POINTER */

DIR$CLOS:
  PROCEDURE(AFTN) PUBLIC;
  DECLARE AFTN BYTE;

    /*
      THIS PROCEDURE ASSUMES THAT AFTWINDOW HAS BEEN CALLED,
      CORRECTLY SETTING UP THE APPROPRIATE DIRECTORY SLOT,
      AND THAT THE DBUFFER AND PBUFFER HAVE CORRECT DATA,
      BUT NEED NOT BE RETURNED TO THE POOL
    */
    CALL WRITE(AFTN,.DIRECT,SIZE(DIRECT));
    IF ASDBUF$READ THEN CALL ABSWRITE(ASDATA$BLK,ASDBUF);
  END DIR$CLOS;

DLOOK:
  PROCEDURE(FN) BOOLEAN PUBLIC;
    DECLARE FN ADDRESS;
    DECLARE RCOUNT BYTE;
        /* THE LOCATION AT (RCOUNT + 1) IS MODIFIED BY THE PROCEDURE READ,
           HENCE BYTENO MUST ALWAYS FOLLOW THE DECLARATION OF RCOUNT.
        */
    DECLARE BLOCKNO ADDRESS DATA (0);
    DECLARE BYTENO ADDRESS;
    DECLARE ARRAY BASED FN (12) BYTE;
    DECLARE (I,AFTN) BYTE;
    DECLARE DNUM ADDRESS;
    DECLARE RETURN$VALUE BOOLEAN;

    /*
      THIS PROCEDURE IS USED TO LOOKUP A FILE NAME IN A
      DISK DIRECTORY.  THE DEVICE NUMBER OF THE DISK
      IS CONTAINED IN FN(0), THE SIX CHARACTERS OF THE
      FILE NAME ARE IN FN(1) THROUGH FN(6), AND THE
      THREE CHARACTERS OF THE FILE EXTENSION ARE IN
      FN(7) THROUGH FN(9).

        THE PROCEDURE RETURNS 'TRUE' IF THE FILE IS FOUND,
      WITH DIRECT$ISND POINTING TO THE ENTRY OF THE FILE IN THE
      DIRECTORY
```

```
      OTHERWISE, THE PROCEDURE RETURNS 'FALSE', AND DIRECT$I$NO POINTS
      AT A BLANK SLOT IN THE DIRECTORY.  THE DIRECTORY MARKER IS
      ADJUSTED SO THAT IT POINTS AT THE BEGINNING OF THE ENTRY
      POINTED TO BY DIRECT$I$NO.

        IF THE DIRECTORY IS FULL, DIRECT$I$NO = 0FFH.

    */

    /* SET AFTN TO SLOT NUMBER FOR DIRECTORY */
    AFT(AFTN := A$DEVICE+AFTN$DIRECT).DBUF = GET$BUF;
    AFT(AFTN).PBUF = GET$BUF;
    CALL AFT$WINDOW(AFTN);
    CALL REWIND;
    DIRECT$I$NO = 0FFH;
    RCOUNT = 1;  /* ANY NON-ZERO VALUE FOR RCOUNT HERE */
    DNUM,DIRECT.EMPTY,RETURN$VALUE = FALSE;  /* DNUM = 0 */
    DO WHILE RCOUNT <> 0 AND DIRECT.EMPTY <> 7FH;
      CALL READ(AFTN,.DIRECT,SIZE(DIRECT),.RCOUNT);
      IF DIRECT.EMPTY THEN
      DO;
        IF DIRECT$I$NO=0FFH THEN DIRECT$I$NO = DNUM;
      END;
      ELSE
      DO;
        DO I = 1 TO 4;
          IF ARRAY(I)<>DIRECT.FILE(I-1) THEN GO TO NOMATCH;
        END;
        DIRECT$I$NO = DNUM;
        RETURN$VALUE = TRUE;
        RCOUNT = 0;
      END;
NOMATCH:
      DNUM = DNUM+1;
    END;
    IF DIRECT$I$NO <> 0FFH THEN
    DO;
      BYTENO = DIRECT$I$NO*SIZE(DIRECT);
      CALL SEEK(AFTN,SEEK$ABS,.BLOCKNO,.BYTENO);
    END;
    CALL RETURN$BUF(A$DBUF);
    CALL RETURN$BUF(A$PBUF);
    RETURN RETURN$VALUE;
  END DLOOK;
END;

EOF
```

```
RENAME:
DO;
 /* ***********************************************************************
    ***********************************************************************

                        MODULE NAME RENAME
                        ====== ==== ======

    ABSTRACT
    ========

    THIS MODULE CONTAINS THE 'RENAME' SYSTEM CALL PROCEDURE ONLY.

    CALLING GRAPH
    ======= =====

    >>RENAME
        'XPATH' (PATH)
        'ERR' (ERROR)
        'AFTSWINDOW' (AFT)
        'DLOOK' (DIRECTORY)
        'WRITE' (RW)
        'DIR$CLOSE' (DIRECTORY)

 */
/*$NOLIST*/
$INCLUDE (:F2:DEVICE.LIT)
$INCLUDE (:F2:ERROR.LIT)
$INCLUDE (:F2:ATTRIB.LIT)
$INCLUDE (:F2:COMMON.LIT)
$INCLUDE (:F2:AFT.LIT)
$INCLUDE (:F2:PATH.MEX)
$INCLUDE (:F2:DIRECT.MEX)
$INCLUDE (:F2:AFT.MEX)
$INCLUDE (:F2:ERROR.MEX)
$INCLUDE (:F2:RW.MEX)
$LIST

RENAME:
  PROCEDURE (OLD$FILE,NEW$FILE) PUBLIC;
    DECLARE (OLD$FILE,NEW$FILE) ADDRESS;
    DECLARE I BYTE;
    DECLARE ALREADY$EXISTS BOOLEAN;
    /* 'OLD$FILE' IS RENAMED; THE NEW NAME IS 'NEW$FILE' */

    CALL XPATH(OLD$FILE,.PN2);
    CALL XPATH(NEW$FILE,.PN);
    IF PN(0) > F5DEV THEN CALL WARN(NON$DISK$FILE);  /* DD */
    IF PN(0) <> PN2(0) THEN CALL WARN(DIFFERENT$DISK);
    CALL AFTSWINDOW(PN(0));
    /* IF NEW$FILE ALREADY EXISTS AND OLD$FILE DOES NOT EXIST,
       THEN WE WANT TO GIVE THE OLD$FILE ERROR MESSAGE.
       HOWEVER, DLOOK(NEW$FILE) SHOULD PRECEDE DLOOK(OLD$FILE).
       THIS IS BECAUSE POINTER VARIABLES SHOULD BE LEFT POINTING TO
       THE OLD FILE UPON EXIT FROM THIS PROCEDURE. */
    ALREADY$EXISTS = DLOOK(.PN);
    IF NOT DLOOK(.PN2) THEN CALL WARN(NO$SUCH$FILE);
    IF (DIRECT.ATTRIB AND (WRITEP$ATTRIBUTE OR FORMAT$ATTRIBUTE)) <> 0
    THEN CALL WARN(WRITE$PROTECT);
    IF ALREADY$EXISTS THEN CALL WARN(MULTIDEFINED);
    DO I = 0 TO 8;
      DIRECT.FILE(I) = PN(I+1);
    END;
```

```
    CALL DIR$CLOS(A$DEVICE);
  END RENAME;
END;

EOF
```

```
RESCAN:
DO;
 /* ****************************************************************
    ****************************************************************

                        MODULE NAME RESCAN
                        ====== ==== ======

    ABSTRACT
    ========

    THIS MODULE CONTAINS ONLY THE RESCAN SYSTEM CALL PROCEDURE.

    CALLING GRAPH
    ======= =====

    >>RESCAN
        AFT$WINDOW (AFT)
        ERR (ERROR)

    GLOBAL VARIABLES ACCESSED
    ====== ========= ========

    A$LBUF (AFT) -- RESCAN
    A$EDIT (AFT) -- RESCAN

    GLOBAL VARIABLES MODIFIED
    ====== ========= ========

     NONE
 */
/*$NOLIST*/
$INCLUDE (:F2:COMMON.LIT)
$INCLUDE (:F2:ERROR.LIT)
$INCLUDE (:F2:AFT.LIT)
$INCLUDE (:F2:AFT.MEX)
$INCLUDE (:F2:ERROR.MEX)
$LIST

RESCAN:
  PROCEDURE(AFTN) PUBLIC;
    DECLARE AFTN BYTE;
    DECLARE TEMP ADDRESS, BYTES BASED TEMP(128) BYTE,
                          ADDR BASED TEMP(64) ADDRESS;

    CALL AFT$WINDOW(AFTN);
    TEMP = A$LBUF;
    IF A$EDIT <> 0 AND NOT BYTES(125) THEN
      ADDR(63) = 0;        /*  LPTR = 0; IN$MODE = FALSE */
    ELSE CALL WARN(CANT$RESCAN);
  END RESCAN;
END;

EOF
```

```
TB: DO;

DECLARE TOBOOT LABEL PUBLIC;

/* THIS VERSION OF TOBOOT HAS BEEN MODIFIED TO WORK ON BOTH
   SINGLE DENSITY AND DOUBLE DENSITY.  IT WILL WORK ON BOTH
   THE MDS AND THE INTELLEC.
*/

$INCLUDE(:F2:CPYRT5.NOT)

$INCLUDE(:F2:CPYRT5.DTA)

$INCLUDE(:F2:COMMON.LIT)

$INCLUDE(:F2:CHAR.LIT)

$INCLUDE(:F2:GEOG.LIT)



$INCLUDE(:F2:WRITE.PEX)

$INCLUDE(:F2:EXIT.PEX)

$INCLUDE(:F2:CONSOL.PEX)

$INCLUDE(:F2:NUMOUT.PEX)

$INCLUDE(:F2:ERROR.LIT)

$INCLUDE(:F2:DISK.LIT)
$INCLUDE(:F2:DEVICE.LIT)

$INCLUDE(:F2:CI.PEX)

$INCLUDE(:F2:RI.PEX)

$INCLUDE(:F2:CO.PEX)

$INCLUDE(:F2:PO.PEX)

$INCLUDE(:F2:LO.PEX)

$INCLUDE(:F2:IOCHK.PEX)

$INCLUDE(:F2:IOSET.PEX)


DECLARE (USER$STATUS,USER$STACKPTR,START$ADDR) ADDRESS;
DECLARE A$DEVICE BYTE;
DECLARE TEMP BYTE;

DECLARE RTC LITERALLY 'OFFH';  /* REAL TIME CLOCK */
DECLARE BOOT LITERALLY '2';
/* INPUT FROM RTC IS A BYTE. THE SECOND BIT FROM THE RIGHT
   CORRESPONDS TO THE BOOT SWITCH. IF THIS BIT IS 1 THE SWITCH IS
   ON, AND IF IT IS 0 THE SWITCH IS OFF.
*/

DECLARE ISIS$SIGNON(12) BYTE INITIAL (CR,LF,'ISIS-II, V'),
        SIGN$V(2) BYTE, /* VERSION NUMBER */
        SIGN$DOT(1) BYTE INITIAL ('.'),
```

```
        SIGN$E(2) BYTE, /* EDIT NUMBER */
        SIGN$CRLF(2) BYTE INITIAL (CR,LF);

/* VERSION$LEVEL AND EDIT$LEVEL ARE SET IN THE LOC62 MODULE */
DECLARE VERSION$LEVEL BYTE AT (62);
DECLARE EDIT$LEVEL BYTE AT (63);


/* THE FOLLOWING FOUR VARIABLES ARE USED IN THE ERR PROCEDURE */
DECLARE
  STATUS ADDRESS ;          /* ERROR NUMBERS ARE PLACED HERE */
DECLARE
  DEBUG$TOGGLE BOOLEAN ;    /* GOVERNS ACTION WHEN ERROR OCCURS */
DECLARE
  GLOBAL$SEVERITY BYTE ;    /* OVERRIDES NORMAL ERROR SEVERITY */
DECLARE
  FDCC$ERROR$TYPE ADDRESS ;/* HAS DATA ON DISK I/O ERRORS */


DECLARE MDSMON ADDRESS DATA (0);  /* AN ENTRY POINT FOR THE MDS MONITOR */

DECLARE BTSTRP ADDRESS DATA (8);  /* AN ENTRY POINT FOR ISIS */


DECLARE INITIO$BASE ADDRESS INITIAL (6), INITIO BASED INITIO$BASE BYTE;


DECLARE MSG1(8) BYTE INITIAL (CR,LF,'ERROR '),
        MSG2(3) BYTE, /* ERROR NUMBER GOES HERE */
        MSG3(9) BYTE INITIAL (' USER PC '),
        MSG4(4) BYTE, /* USER PC IN HEX GOES HERE */
        MSG5(2) BYTE INITIAL (CR,LF),
        MSG6(5) BYTE INITIAL ('FDCC='),
        MSG7(4) BYTE, /* FDCC ERROR DATA GOES HERE */
        MSG8(2) BYTE INITIAL (CR,LF);


DECLARE SYS$FLG BYTE AT (0FFFFH);   /* 1 FOR INTELLEC, 0 FOR MDS */
DECLARE HDR$BLK ADDRESS;  /* HEADER BLOCK USED FOR LOADING ISIS */
DECLARE I BYTE;
DECLARE CONFIG$TABLE (6) BYTE;  /* WORKING CONFIGURATION TABLE */
DECLARE DKCFTB (6) BYTE EXTERNAL;   /* DISK CONFIGURATION TABLE IN ISIS */
DECLARE STAT BYTE;

DECLARE CON$MASK LITERALLY '00001000B';  /*  DISK CONTROLLER PRESENT */
DECLARE DD$MASK LITERALLY '00010000B';  /* DD PRESENT */
DECLARE ISD$MASK LITERALLY '00010000B';  /* ISD FLOPPY PRESENT */
DECLARE READ$STATUS$COMMAND LITERALLY '01CH';  /* READ STATUS COMMAND OF ISD */

DECLARE ENABL LITERALLY '05H';  /* PSEUDO ENABLE OF INTERRUPT */
DECLARE DISABL LITERALLY '0DH';  /* PSEUDO DISABL OF INTERRUPT  */
DECLARE CPUC LITERALLY '0FFH';  /* CONTROLLER PORT */

DECLARE MESS88(*) BYTE INITIAL (CR,LF,'ILLEGAL DISK DEVICE AT PORT 88H',CR,LF);


/* ***********************************************************

PROCEDURE ERR
--------- ---

  ABSTRACT
  --------
```

```
  THE ERR PROCEDURE IS USED TO HANDLE ERROR CONDITIONS.
  IF THE SEVERITY OF THE ERROR IS MEDIUM ('MESSAGE') TO
  HIGH ('ABORT'), THEN AN ERROR MESSAGE IS ISSUED TO THE
  CONSOLE.  IF THE SEVERITY IS HIGH ('ABORT') AND IF THE
  VARIABLE 'DEBUG$TOGGLE' IS SET TRUE, THEN CONTROL PASSES
  TO THE MDS MONITOR.

  PARAMETERS
  ----------

  SEVERITY     THE SEVERITY OF THE ERROR, SUCH AS MEDIUM ('MESSAGE')
               OR HIGH ('ABORT').

  ERROR$TYPE   ERROR NUMBER WHICH IDENTIFIES THE PARTICULAR
               TYPE OF ERROR, SUCH AS DISK I/O ERROR.

*/

ERR:
  PROCEDURE (SEVERITY,ERROR$TYPE);
    DECLARE (SEVERITY,ERROR$TYPE,I,IMAX) BYTE;
    DECLARE PC BASED USER$STACK$PTR ADDRESS;

    IF (SEVERITY := SEVERITY OR GLOBAL$SEVERITY) <> WARNING THEN
    DO;
      CALL NUMOUT(ERROR$TYPE,10,0,.MSG2,3);
      CALL NUMOUT(PC,16,'0',.MSG4,4);
      CALL IOSET((IOCHK AND CMSK) OR (INITIO AND 3));
      IMAX = 25;  /* NUMBER OF CHARACTERS IN NORMAL ERROR MESSAGE */
      IF ERROR$TYPE = DISK$IO$ERROR THEN
      DO;
        IMAX = 36; /* NUMBER OF CHARACTERS IN ERROR MESSAGE */
        CALL NUMOUT(FDCC$ERROR$TYPE,16,'0',.MSG7,4);
      END;
      DO I = 0 TO IMAX;
        CALL CO(MSG1(I));
      END;
    END;
    ELSE
    DO;
      USER$STATUS = ERROR$TYPE;
      STACK$PTR = USER$STACK$PTR;
    END;
    IF SEVERITY >= ABORT THEN
    DO;
      IF DEBUG$TOGGLE THEN CALL MDSMON; /* EXIT VIA MDS MONITOR */
      CALL BTSTRP; /* EXIT VIA SOFTWARE BOOTSTRAP */
    END;
  END ERR;


/* ***********************************************************************

PROCEDURE CONFIG
--------- ------

  ABSTRACT
  --------

  CONFIG DETERMINES THE CONFIGURATION OF DISK DRIVES ON THE SYSTEM AND
  PUTS THE INFORMATION INTO AN ARRAY CALLED CONFIGSTABLE.
```

```
  DESCRIPTION
  -----------

  CONFIG READS INPUTS FROM THE PORTS ASSOCIATED WITH THE DISK CONTROLLERS.
  IT DETERMINES WHETHER EACH DRIVE IS DOUBLE DENSITY, SINGLE DENSITY, OR
  INTEGRATED SINGLE DENSITY.  THIS INFORMATION IS PUT INTO THE ARRAY
  CALLED CONFIG$TABLE, WITH 1 STANDING FOR DOUBLE DENSITY, 2 FOR
  SINGLE DENSITY, AND 3 FOR INTEGRATED SINGLE DENSITY.  0 MEANS THE DRIVE
  IS NOT BEING USED.  EACH BYTE OF THE ARRAY CORRESPONDS TO THE DISK DRIVE
  OF THE SAME NUMBER (CONFIG$TABLE(0) = :F0: ,ETC.)
*/

CONFIG:
  PROCEDURE;
    DECLARE I BYTE;

    /* INITIALIZE CONFIG$TABLE WITH ALL ZEROS */
    DO I = 0 TO 5;
      CONFIG$TABLE(I) = 0;
    END;

    /* READ STATUS OF DISK CONTROLLER AT 78H
       AND FILL IN CONFIG$TABLE ACCORDINGLY */
    STAT = INPUT(FDCC$STATUS$0);
    IF SHR((STAT AND CON$MASK),3) THEN  /* CONTROLLER PRESENT */
      DO;
        IF SHR((STAT AND DD$MASK),4) THEN  /* DD PRESENT*/
          DO I = 0 TO 3;
            CONFIG$TABLE(I) = 1;
          END;
          ELSE  /* SD */
          DO I = 0 TO 1;
            CONFIG$TABLE(I) = 2;
          END;
      END;

    /* READ STATUS OF DISK CONTROLLER AT 88H
       AND FILL IN CONFIG$TABLE ACCORDINGLY */
    STAT = INPUT(FDCC$STATUS$1);
    IF SHR((STAT AND CON$MASK),3) THEN  /* CONTROLLER PRESENT */
      DO;
        IF SHR((STAT AND DD$MASK),4) THEN  /* DD PRESENT */
          DO;
            DO I = 0 TO LENGTH(MESS88) - 1;
              CALL CO(MESS88(I));
            END;
            CALL MDSMON;  /* EXIT VIA MONITOR */
          END;
          ELSE  /* SD */
          DO;
            IF CONFIG$TABLE(2) = 0 AND CONFIG$TABLE(3) =0 THEN  /* NOT DD SYS */
              DO I = 2 TO 3;
                CONFIG$TABLE(I) = 2;
              END;
              ELSE  /* DD SYS */
              DO I = 4 TO 5;
                CONFIG$TABLE(I) = 2;
              END;
          END;
      END;

    /* DETERMINE IF SYSTEM IS AN MDS OR INTELLEC  IF IT IS AN
```

```
      INTELLEC, THEN READ STATUS OF THE INTEGRATED SINGLE
      DENSITY CONTROLLER AND FILL IN CONFIG$TABLE ACCORDINGLY.  */
    IF SYS$FLG = 1 THEN  /* SYSTEM IS AN INTELLEC */
      DO;
        OUTPUT(CPUC) = DISABL;  /* DISABLE INTERRUPTS */
        I = 25;  /* THIS TIMEOUT LOOP IS NECESSARY TO TAKE
                    CARE OF CASES WHERE THERE IS NO IOC.  */
        DO WHILE ((INPUT(0C1H) AND 07H) <> 0) AND ((I := I - 1) <> 0);
          ;  /* INPUT DBB STATUS. LOOP UNTIL F0 = IBF = OBF = 0. */
        END;
        IF I = 0 THEN
          DO;
            OUTPUT(CPUC) = ENABL;
            RETURN;
          END;
        OUTPUT(0C1H) = READ$STATUS$COMMAND;  /* ISSUE COMMAND */
        I = 250;     /* THIS TIMEOUT LOOP IS PLACED HERE BECAUSE AT THE
                        MOMENT,(NOV.'77), NOT ALL INTELLEC SYSTEMS HAVE
                        THE CURRENT IOC FIRMWARE. */
        DO WHILE ((INPUT(0C1H) AND 07H) <> 1) AND ((I := I - 1) <> 0);
          ;    /* INPUT DBB STATUS. LOOP UNTIL F0 = IBF = 0 AND OBF = 1. */
        END;

        IF I = 0 THEN
          DO;
            OUTPUT(CPUC) = ENABL;
            RETURN;
          END;
        STAT = INPUT(0C0H);  /* INPUT STATUS FROM ISD */
        OUTPUT(CPUC) = ENABL;  /* ENABLE INTERRUPTS */
        IF SHR((STAT AND ISD$MASK),3) THEN   /* ISD PRESENT */
          DO;
            IF CONFIG$TABLE(0) = 0 THEN    /* ISD IS SYSTEM DISK */
              CONFIG$TABLE(0) = 3;
            ELSE   /* ISD IS NOT SYSTEM DISK */
              CONFIG$TABLE(4) = 3;
          END;
      END;

END CONFIG;



/* ***********************************************************************

                        PROCEDURE DISKIO
                        --------- ------

    ABSTRACT
    --------

        THIS PROCEDURE PROVIDES ACCESS TO THE FDCC CONTROLLER(S),
        CONTROLLER 1 = BASE ADDRESS 78H (DRIVES 0,1,2,3 FOR DD,
                                         DRIVES 0,1 FOR SD),
        CONTROLLER 2 = BASE ADDRESS 88H (DRIVES 2,3 FOR SD,
                                        OR DRIVES 4,5 WHEN 0-3 IS DD),
        INTEGRATED SINGLE DENSITY CONTROLLER = PORT 0C1H.

    PARAMETERS
    ----------

    DRIVE   AN INTEGER 0 THROUGH 5, SPECIFYING THE DISK TO BE
            ACCESSED
```

```
    IOPB    THE ADDRESS OF A PARAMETER BLOCK TO BE SENT TO THE
            FDCC CONTROLLER.  THIS PARAMETER BLOCK MUST BE SET UP
            AS IF IT WERE FOR DRIVE 0; IF 'DRIVE' SELECTS
            ANOTHER DRIVE, 'DISKIO' WILL SET ALL THE NECESSARY
            ADDITIONAL BITS.

    VALUE RETURNED
    ----- --------

    NONE

    DESCRIPTION
    -----------

        THE CALLER PROVIDES A PARAMETER BLOCK SPECIFYING SOME
    VALID DISK OPERATION ON DRIVE 0, AND AN INTEGER DRIVE
    SELECT VALUE.
        THIS PROCEDURE WAITS FOR THE CONTROLLER TO GO UNBUSY, THEN
    PERFORMS THE DESIRED ACTION.  IN CASE OF A CONTROLLER ERROR,
    THE DISK DRIVE IS RECALIBRATED AND THE ACTION IS TRIED AGAIN.
    IF A SUCCESSFUL COMPLETION CANNOT BE OBTAINED AFTER "MAX$RETRIES"
    ATTEMPTS, A FATAL ERROR OCCURS; OTHERWISE A NORMAL RETURN IS
    MADE.
 */

DECLARE DRIVE$READY BYTE DATA(01H);

DISKIO:
  PROCEDURE (DRIVE,IOPB) PUBLIC;
  /* THIS PROCEDURE ISSUES THE IOPB TO THE DISK CONTROLLER. IN THE */
  /* CASE OF THE 8271 IT ALSO TRANSFERS THE DATA ON A BYTE BY BYTE BASIS */
    DECLARE DRIVE BYTE;        /* DRIVE NUMBER; ASSUMES VALUE 0,1,2,3,4 OR 5 */
    DECLARE TEMP ADDRESS, (TEMP2,TEMP1) BYTE AT (.TEMP);
    DECLARE IOPB ADDRESS;      /* POINTER TO THE PARAMETER BLOCK NAMED DCB */
    DECLARE DCB BASED IOPB STRUCTURE (
                    IOCW BYTE,          /* CHANNEL WORD */
                    IOINS BYTE,         /* INSTRUCTION */
                    NSEC BYTE,          /* NUMBER OF SECTORS */
                    TADR BYTE,          /* TRACK ADDRESS */
                    SADR BYTE,          /* SECTOR ADDRESS */
                    BUF ADDRESS);    /* BUFFER ADDRESS */
    DECLARE RECAL$PB STRUCTURE (
                    IOCW BYTE,
                    IOINS BYTE,
                    NSEC BYTE,
                    TADR BYTE,
                    SADR BYTE);
    DECLARE I BYTE;                 /* INDEX VARIABLE USED IN FOR STATEMENTS */
    DECLARE IVAL BYTE;                 /* INTERRUPT MASK VALUE */
    DECLARE WPBC$COMMAND       LITERALLY       '15H', /* ISD */
                    WPCC$COMMAND        LITERALLY       '16H', /* ISD */
                    WDBC$COMMAND        LITERALLY       '17H', /* ISD */
                    WDCC$COMMAND        LITERALLY       '18H', /* ISD */
                    RDBC$COMMAND        LITERALLY       '19H', /* ISD */
                    RDCC$COMMAND        LITERALLY       '1AH', /* ISD */
                    RRSTS$COMMAND       LITERALLY       '1BH', /* ISD */
                    RDSTS$COMMAND       LITERALLY       '1CH'; /* ISD */
     DECLARE (ISD$DRIVE, DD$DRIVE) BOOLEAN;
     /* INDICATES IF DRIVE NUMBER CORRESPONDS TO ISD OR DD */

    DECLARE BUFF$PTR ADDRESS;  /* WILL BE SAME AS DCB.BUF */
    DECLARE (BUFF BASED BUFF$PTR) (128) BYTE;  /* USED FOR DATA TRANSFER WITH ISD */
```

```
IOCDR1:                          /* ISD */
  PROCEDURE (COMMAND) BYTE;
  /* THIS PROCEDURE RETURNS EITHER THE FLOPPY DEVICE STATUS OR DATA */
  /* FROM THE 8271 ISD.  IT IS THE PLM80 EQUIVALENT TO THE MONITOR'S*/
  /* IOCDR1 ROUTINE.                                                */
    DECLARE (COMMAND, INPUT$DATA) BYTE;

      OUTPUT(CPUC) = DISABL;                    /* DISABLE INTERRUPTS */
      DO WHILE (INPUT(0C1H) AND 07H) <> 0;
        ;               /* INPUT DBB STATUS; LOOP UNTIL F0 = IBF = OBF = 0 */
      END;
      OUTPUT(0C1H) = COMMAND;           /* ISSUE THE COMMAND */
      DO WHILE (INPUT(0C1H) AND 07H) <> 1;
        ;               /* INPUT DBB STATUS; LOOP UNTIL F0 = IBF = 0 AND OBF = 1 */
      END;
     INPUT$DATA = INPUT(0C0H);          /*INPUT STATUS OR DATA FROM DBB */
     OUTPUT(CPUC) = ENABL;              /* ENABLE INTERRUPTS */
      RETURN (INPUT$DATA);       /* RETURN WITH THE DEVICE STATUS OR DISK DATA */
  END IOCDR1;

IOCDR2:                                 /* ISD */
  PROCEDURE (COMMAND, OUTPUT$DATA);
  /* THIS PROCEDURE OUTPUTS DATA TO THE 8271 ISD. IT IS THE PLM80    */
  /* TO THE MONITOR'S IOCDR2 ROUTINE.                                */

    DECLARE (COMMAND, OUTPUT$DATA) BYTE;

      OUTPUT(CPUC) = DISABL;     /* DISABLE INTERRUPTS */
      DO WHILE (INPUT(0C1H) AND 07H) <> 0;
        ;       /* INPUT DBB STATUS; LOOP UNTIL F0 = IBF = OBF = 0 */
      END;
      OUTPUT(0C1H) = COMMAND;    /* ISSUE THE COMMAND */
      DO WHILE (INPUT(0C1H) AND 07H) <> 0;
        ;       /* INPUT DBB STATUS; LOOP UNTIL F0 = IBF = OBF = 0 */
      END;
      OUTPUT(0C0H) = OUTPUT$DATA;       /* WRITE DATA TO THE ISD FLOPPY DISK */
      OUTPUT(CPUC) = ENABL;      /* ENABLE INTERRUPTS */
    END IOCDR2;

TRANSFER$IOPB$TO$ISD:
  PROCEDURE;
  /* THIS PROCEDURE SENDS THE IOPB TO THE 8271 ISD             */
      CALL IOCDR2(WPBC$COMMAND, DCB.IOCW);
      CALL IOCDR2(WPCC$COMMAND, DCB.IOINS);
      CALL IOCDR2(WPCC$COMMAND, DCB.NSEC);
      CALL IOCDR2(WPCC$COMMAND, DCB.TADR);
      CALL IOCDR2(WPCC$COMMAND, DCB.SADR);
  END TRANSFER$IOPB$TO$ISD;

DISK$STAT:
  PROCEDURE BYTE;
  /* THIS PROCEDURE RETURNS THE DISK DEVICE STATUS           */
    IF ISD$DRIVE                                    /* ISD */
    THEN RETURN IOCDR1(RDSTS$COMMAND);
    ELSE RETURN INPUT(FDCC$STATUS$0); /* DD ON F0,F1,F2,F3 */
                                      /* SD ON F0,F1       */
  END DISK$STAT;

RSTYPE:
  PROCEDURE BYTE;
  /* THIS PROCEDURE RETURNS WITH THE RESULT TYPE OF A DISK OPERATION  */
    IF ISD$DRIVE                                    /* ISD */
```

```
    THEN RETURN 00H;     /* THERE IS NO RESULT TYPE FOR ISD OPERATION */
    ELSE RETURN INPUT(RESULT$TYPE$0);   /* DD ON F0,F1,F2,F3 */
                                        /* SD ON F0,F1       */
  END R$TYPE;

R$BYTE:
  PROCEDURE BYTE;
  /* THIS PROCEDURE RETURNS WITH THE RESULT BYTE OF A DISK OPERATION   */
    IF ISD$DRIVE                                    /* ISD */
    THEN RETURN IOCDR1(RRSTS$COMMAND);
    ELSE RETURN INPUT(RESULT$BYTE$0);   /* DD ON F0,F1,F2,F3  */
                                        /* SD ON F0,F1        */
  END R$BYTE;

START$IO:
  PROCEDURE (IOPB);
  /* THIS PROCEDURE OUTPUTS THE ADDRESS OF THE IOPB TO THE DISK CONTROLLER   */
  /* AND IN THE CASE OF THE 8271 ISD INVOLVING A DATA TRANSFER OPERATION,   */
  /* IT ALSO TRANSFERS THE DATA ON A BYTE BY BYTE BASIS.                    */

  DECLARE IOPB ADDRESS;

  IF ISD$DRIVE                              /* ISD */
  THEN DO;
          CALL TRANSFER$IOPB$TO$ISD;    /* ISSUE THE IOPB */
          DO WHILE (DISK$STAT AND DISK$DONE) = 0;
            ;
          END;
          IF DCB.IOINS = READ$COMMAND THEN DO;
            BUFF$PTR = DCB.BUF;
            OUTPUT(CPUC) = DISABL;
            DO WHILE (INPUT(0C1H) AND 07H) <> 0;
              ;  /* INPUT DBB STATUS; LOOP UNTIL F0 = IBF = OBF = 0 */
            END;
            OUTPUT(0C1H) = RDBC$COMMAND;
            DO I = 0 TO 127;  /* GET 128 BYTES OF DATA */
              DO WHILE (INPUT(0C1H) AND 07H) <> 1;
                ;  /* INPUT DBB STATUS; LOOP UNTIL F0 = IBF = 0 AND OBF = 1 */
              END;
              BUFF(I) = INPUT(0C0H);  /* INPUT DATA FROM DBB */
            END;
            OUTPUT(CPUC) = ENABL;
          END;
      END;
  ELSE DO;    /* NOT ISD */
         OUTPUT(LOW$ADDRESS$0) = LOW(IOPB);
         OUTPUT(HIGH$ADDRESS$0) = HIGH(IOPB);
         DO WHILE (DISK$STAT AND DISK$DONE) = 0;
           ;
         END;
         END;
END START$IO;

    IVAL = INPUT(0FCH);
    OUTPUT(0FCH) = IVAL OR 2;
    /*  PREVENT INTERRUPT 1 FROM CAUSING REENTRANCY */
    IF CONFIG$TABLE(DRIVE) = 3 THEN ISD$DRIVE = TRUE;
    ELSE DO;
            ISD$DRIVE = FALSE;
            IF CONFIG$TABLE(DRIVE) = 2 THEN DD$DRIVE = TRUE;
            ELSE DO;
                    DD$DRIVE = FALSE;
                    /* MUST BE SD OR CONTROLLER NOT PRESENT */
```

```
                END;
        END;

    RECAL$PB.IOCW = 80H;
    RECAL$PB.IOINS = RECALIBRATE;
    RECAL$PB.SADR = 0;


    DO I = 0 TO MAX$RETRIES;
      DO WHILE (DISK$STAT AND DISK$DONE) <> 0;
        TEMP1 = R$TYPE;
        TEMP1 = R$BYTE;
      END;
      /* IF DISK DRIVE NOT READY, GIVE FATAL ERROR */

      IF (DISK$STAT AND DRIVE$READY) = 0
      THEN CALL ERR(ABORT,DRIVE$NOT$READY);
      CALL START$IO(IOPB);
      TEMP1 = R$TYPE;
      IF (TEMP2 := R$BYTE) = 0
      THEN DO;
                OUTPUT(0FCH) = IVAL;    /* RESTORE INTERRUPT 1 */
                RETURN;
           END;
      CALL START$IO(.RECAL$PB);
    END;

    FDCC$ERROR$TYPE = TEMP;
    CALL ERR(ABORT,DISKIO$ERROR);
  END DISKIO;

/* ***************************************************************

                        PROCEDURE ABSIO
                        --------- -----

    ABSTRACT
    --------

       ABSIO ACCOMPLISHES THE TRANSFER OF 128 BYTES OF
    DATA TO/FROM THE DISKETTE.

    PARAMETERS
    ----------
    COMMAND     MUST BE THE NUMERIC VALUE OF THE FDCC COMMAND
                DESIRED.  (LITERALS 'READ$COMMAND' AND 'WRITE$COMMAND'
                EXIST FOR THE COMMON OPERATIONS.)
    DISK        INTEGER 0 OR 1, SELECTS WHICH DRIVE.
    BLOCK       DISKETTE BLOCK NUMBER, A TRACK NUMBER (0-76)
                IN THE HIGH ORDER 8 BITS AND A SECTOR NUMBER (1-26)
                IN THE LOW ORDER 8 BITS.
    BUFFER$PTR  THE ADDRESS OF A 128 BYTE BUFFER IN RAM.

    VALUE RETURNED
    ----- --------

    NONE

    DESCRIPTION
    -----------

    AN I/O PARAMETER BLOCK ("DCB") IS SETUP ACCORDING TO THE
    PARAMETERS PROVIDED, AND "DISKIO" IS CALLED
```

```
*/

ABSIO:
  PROCEDURE (COMMAND,DISK,BLOCK,BUFFER$PTR) PUBLIC;
    DECLARE (COMMAND,DISK) BYTE;
    DECLARE (BLOCK,BUFFER$PTR) ADDRESS;
    /* VALUE OF 'DISK' MUST BE 0  - 3 */
    DECLARE DCB STRUCTURE (
            IOCW BYTE,
            IOINS BYTE,
            NSEC BYTE,
            TADR BYTE,
            SADR BYTE,
            BUF ADDRESS);

    DCB.IOCW = 80H;
    DCB.IOINS = COMMAND;
    DCB.NSEC = 1;  /* IF THIS VALUE CHANGES IN THE FUTURE,  */
                   /* THE 8271 DRIVER MUST BE CHANGED       */
    DCB.TADR = HIGH(BLOCK);
    DCB.SADR = LOW(BLOCK);
    DCB.BUF = BUFFER$PTR;
    CALL DISKIO(DISK,.DCB);
  END ABSIO;


/* ***********************************************************

PROCEDURE ALOADR
--------- ------

  ABSTRACT
  --------

  ALOADR LOADS THE ABSOLUTE ISIS FILE INTO MEMORY. ALOADR
  CALLS ABSIO, WHICH IN TURN CALLS DISKIO.

  PARAMETERS
  ----------

  HDRBLK    HEADER BLOCK OF ISIS FILE.

  VALUE RETURNED
  ----- --------

  LOADADR   ADDRESS WHERE FILE IS LOADED.

  DESCRIPTION
  -----------

  ALOADR LOADS ISIS INTO A 128 BYTE BUFFER.  THEN ISIS IS
  TRANSFERED FROM THE BUFFER TO MEMORY.

*/

ALOADR:
  PROCEDURE(HDRBLK) ADDRESS ;
    /* LOAD INTO MEMORY THE ABS LOAD FILE AT 'HDRBLK' */
    DECLARE HDRBLK ADDRESS;
    DECLARE BUF(128) BYTE, BUFPTR BYTE;
    DECLARE POINTERS(64) ADDRESS, PTRPTR BYTE;
    DECLARE LOADADR ADDRESS, TARGET BASED LOADADR BYTE;
```

```
    DECLARE LENGTH ADDRESS;

    L1B:
      /* LOAD 1 BYTE */
      PROCEDURE; /* LOADS 1 BYTE FROM BUF INTO MEMORY */
        TARGET = BUF(BUFPTR);
        BUFPTR = BUFPTR + 1;
        LENGTH = LENGTH - 1;
        LOADADR= LOADADR+ 1;
      END L1B;

    G128B:
      /* GET 128 BYTES INTO BUFFER AT BUFADR */
      PROCEDURE(BUFADR);
        DECLARE BUFADR ADDRESS;
        IF (PTRPTR := PTRPTR + 1) = 64 THEN
        DO;
          IF POINTERS(FLINK) = 0 THEN CALL MDSMON;
          CALL ABSIO(READ$COMMAND,0,POINTERS(FLINK),.POINTERS);
          PTRPTR = 2;
        END;
        IF POINTERS(PTRPTR) = 0 THEN CALL MDSMON;
        CALL ABSIO(READ$COMMAND,0,POINTERS(PTRPTR),BUFADR);
      END G128B;

    G1B:
      /* GET 1 BYTE */
      PROCEDURE BYTE;
        IF (BUFPTR AND 7FH) = 0 THEN
        DO;
          CALL G128B(.BUF);
          BUFPTR = 0;
        END;
        BUFPTR = BUFPTR + 1;
        RETURN BUF(BUFPTR-1);
      END G1B;

    G2B:
      /* GET 2 BYTES */
      PROCEDURE ADDRESS;
        RETURN G1B + (256 * G1B);
      END G2B;

    POINTERS(FLINK) = HDRBLK; /* INITILIZE G128B */
    PTRPTR = 63;              /*     ''      ''  */
    BUFPTR = 0;
STARTSLOADINGSNEXTSRECORD:
    DO WHILE TRUE;
      LENGTH = G2B;
      LOADADR= G2B;
      IF LENGTH = 0 THEN RETURN LOADADR;
      DO WHILE BUFPTR <> 128;
        IF LENGTH > 0 THEN CALL L1B;
        ELSE GO TO STARTSLOADINGSNEXTSRECORD;
      END;
      DO WHILE LENGTH >= 128;
        CALL G128B(LOADADR);
        LOADADR = LOADADR + 128;
        LENGTH  = LENGTH  - 128;
      END;
      BUFPTR = 0;
      IF LENGTH <> 0 THEN
      DO;
```

```
        CALL G128B(.BUF);
        DO WHILE LENGTH > 0;
          CALL L1B;
        END;
      END;
    END;
    RETURN LOADADR;
  END ALOADR;


/* ***********************************************************

              BEGINNING OF MAIN PROGRAM                    */


TOBOOT:

TEMP = INPUT(RESULTSTYPE$0);
TEMP = INPUT(RESULTSBYTE$0);
ENABLE;
IF SYSSFLG = 0 THEN    /* SYSTEM IS AN MDS. WAIT FOR BOOT SWITCH */
  DO WHILE (INPUT(RTC) AND BOOT) <> 0;
    ;
  END;
CALL CONFIG;  /* DETERMINE DISK CONFIGURATION */
GLOBAL$SEVERITY = WARNING;
DEBUGSTOGGLE = TRUE;  /* CONTROL RETURNS TO MONITOR AFTER AN ERROR */
IF CONFIG$TABLE(0) = 1 THEN HDR$BLK = ISIS$HDR$BLK;  /* DD DISK */
ELSE HDR$BLK = S$ISIS$HDR$BLK;  /* SD DISK */
START$ADDR = ALOADR(HDR$BLK);  /* LOAD ISIS */
DO I = 0 TO 5;  /* COPY DISK CONFIGURATION INFORMATION TO ISIS */
  DKCFTB(I) = CONFIG$TABLE(I);
END;
CALL CONSOL(.MEMORY,.MEMORY,.USER$STATUS); /* INITIALIZE CONSOLE */
CALL NUMOUT(VERSION$LEVEL,16,0,.SIGN$V,2);
CALL NUMOUT(EDIT$LEVEL,16,0,.SIGN$E,2);
CALL WRITE(0,.ISIS$SIGNON,19,.USER$STATUS);
CALL EXIT;


END;
```

```
ATTRIB:
DO;

DECLARE VERSION$LEVEL LITERALLY '03H',
        EDIT$LEVEL LITERALLY '00H';

DECLARE VERSION (*) BYTE DATA (VERSION$LEVEL,EDIT$LEVEL);

$INCLUDE (:F2:CPYRT5.NOT)
$INCLUDE (:F2:CPYRT5.DTA)
/*$NOLIST*/
$INCLUDE (:F2:COMMON.LIT)
$INCLUDE (:F2:CHAR.LIT)
$INCLUDE (:F2:ATTRIB.LIT)
$INCLUDE (:F2:OPEN.LIT)
$INCLUDE (:F2:ERROR.LIT)
$INCLUDE (:F2:READ.PEX)
$INCLUDE (:F2:WRITE.PEX)
$INCLUDE (:F2:ATTRIB.PEX)
$INCLUDE (:F2:EXIT.PEX)
$INCLUDE (:F2:OPEN.PEX)
$INCLUDE (:F2:UNPATH.PEX)
$INCLUDE (:F2:WPATH.PEX)
$INCLUDE (:F2:UCASE.PEX)
$INCLUDE (:F2:SEA.PEX)
$INCLUDE (:F2:WDELIM.PEX)
$INCLUDE (:F2:DMEM.PEX)
$INCLUDE (:F2:DLIMIT.PEX)
$INCLUDE (:F2:DBLANK.PEX)
$INCLUDE (:F2:FUPPER.PEX)
$INCLUDE (:F2:FERROR.PEX)
$LIST

DECLARE (AFTN,ACTUAL,STATUS) ADDRESS;
DECLARE BUFFER$PTR ADDRESS, CHAR BASED BUFFER$PTR(1) BYTE;
DECLARE BUFFER (128) BYTE;
DECLARE DIRSFIL(*) BYTE INITIAL (':FX:ISIS.DIR ');
DECLARE PATHNAME(15) BYTE;
DECLARE BUF16(16) BYTE;
DECLARE PN(10) BYTE;
DECLARE OLDSAT BYTE;  /* STORES ORIGINAL ATTRIBUTES */
DECLARE DIR$AFTN ADDRESS;
DECLARE PTR$SAVE ADDRESS;
DECLARE FOUND BYTE;
DECLARE FILE$NAME ADDRESS;
DECLARE ISNO BYTE;
DECLARE DISK BYTE;
DECLARE TEMP BYTE;
DECLARE I BYTE;
DECLARE VALUE BYTE;
DECLARE QUES(*) BYTE DATA (', MODIFY ATTRIBUTES? ');
DECLARE ATTRIB$MODIFIED(*) BYTE DATA (', ATTRIBUTES MODIFIED',CR,LF);
DECLARE AT$LIST(*) BYTE DATA(1,2,4,80H);
DECLARE OPTION(9) STRUCTURE(
        LEN BYTE,
        STR(2) BYTE,
        SW BOOLEAN,
        SWID BYTE,
        VAL BYTE
        ) INITIAL (
        1,'R ',FALSE,0,0,
        2,'WO',FALSE,WRITEP$SWID,0,
        2,'W1',FALSE,WRITEP$SWID,1,
```

```
        2,'I0',FALSE,INVISIBLE$SWID,0,
        2,'I1',FALSE,INVISIBLE$SWID,1,
        2,'F0',FALSE,FORMAT$SWID,0,
        2,'F1',FALSE,FORMAT$SWID,1,
        2,'S0',FALSE,SYSTEM$SWID,0,
        2,'S1',FALSE,SYSTEM$SWID,1
        );

/* *-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*
   *-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*

     BEGINNING OF MAIN PROGRAM.

   *-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*
   *-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-* */

CALL READ(1,.BUFFER,LENGTH(BUFFER),.ACTUAL,.STATUS);
BUFFER(ACTUAL) = CR;
CALL FORCE$UPPER(.BUFFER);
BUFFER$PTR = DEBLANK(.BUFFER);
FOUND = NO$SUCH$FILE;
FILE$NAME = BUFFER$PTR;
STATUS = UPATH(BUFFER$PTR,.PN);
CALL FILE$ERROR(STATUS,BUFFER$PTR,TRUE);
BUFFER$PTR = DEBLANK(UDELIMIT(BUFFER$PTR));
DIR$FIL(2) = (DISK:=PN(0)) + '0';
PN(0) = 0;

CALL WRITE(0,.('      FILE        CURRENT ATTRIBUTES',CR,LF),38,.STATUS);

CALL OPEN(.DIR$AFTN,.DIR$FIL,READ$MODE,0,.STATUS);
ISND = 0;
PTR$SAVE = BUFFER$PTR;
DO WHILE ISND < 200;
  DO I = 0 TO LAST(OPTION);
    OPTION(I).SW = FALSE;
  END;
  ISND = DNEW(DIR$AFTN,.PN,ISND,.BUF16);
  OLD$AT = BUF16(10);
  IF ISND <= 200 THEN
  DO;
    FOUND = OK;
    BUFFER$PTR = PTR$SAVE;
    BUF16(0) = DISK;
    CALL UNPATH(.BUF16,.PATHNAME);
    BUFFER$PTR = DEBLANK(BUFFER$PTR);
    STATUS = INVALID$SYNTAX;
    DO WHILE CHAR(0) <> CR;
      IF CHAR(0) = '$' THEN
        BUFFER$PTR = DEBLANK(BUFFER$PTR+1);
      STATUS = UNRECOG$SWITCH;
      DO I = 0 TO LAST(OPTION);
        IF SEQ(BUFFER$PTR,.OPTION(I).STR,OPTION(I).LEN)
          AND DELIMIT(BUFFER$PTR)-BUFFER$PTR = OPTION(I).LEN THEN
        DO;
          OPTION(I).SW = TRUE;
          STATUS = OK;
          IF I > 0 THEN
          DO;
            IF I THEN OPTION(I+1).SW = FALSE;
            ELSE OPTION(I-1).SW = FALSE;
          END;
        END;
```

```
      END;
      CALL FILE$ERROR(STATUS,BUFFER$PTR,TRUE);
      BUFFER$PTR = DEBLANK(DELIMIT(BUFFER$PTR));
    END;
    CALL FILE$ERROR(STATUS,BUFFER$PTR,TRUE);
    MEMORY(0) = 'Y';
    IF OPTION(0).SW THEN /* QUERY */
    DO;
      CALL WRITE(0,.(' '),1,.STATUS);
      CALL WRITE(0,.PATHNAME,DELIMIT(.PATHNAME)-.PATHNAME,.STATUS);
      CALL WRITE(0,.QUES,LENGTH(QUES),.STATUS);
      CALL READ(1,.MEMORY,128,.ACTUAL,.STATUS);
    END;
    IF UPPER$CASE(MEMORY(0)) = 'Y' THEN
    DO;
      DO I = 1 TO LAST(OPTION);
        IF OPTION(I).SW THEN
          DO;
          CALL ATTRIB(.PATHNAME,OPTION(I).SWID,OPTION(I).VAL,.STATUS);
          CALL FILE$ERROR(STATUS,.PATHNAME,TRUE);
          OLD$AT = OLD$AT OR AT$LIST(OPTION(I).SWID);
          IF OPTION(I).VAL = 0 THEN
            OLD$AT = OLD$AT XOR AT$LIST(OPTION(I).SWID);
        END;
      END;
/*      IF OLD$AT <> BUF16(10) THEN */
        DO;
      CALL WRITE(0,.(' '),1,.STATUS);
      TEMP = DELIMIT(.PATHNAME)-.PATHNAME;
      CALL WRITE(0,.PATHNAME,TEMP,.STATUS);
      DO I = 1 TO 25-TEMP;
        CALL WRITE(0,.(' '),1,.STATUS);
      END;
      IF (OLD$AT AND WRITEP$ATTRIBUTE) <> 0 THEN
        CALL WRITE(0,.('W'),1,.STATUS);
      IF (OLD$AT AND SYSTEM$ATTRIBUTE) <> 0 THEN
        CALL WRITE(0,.('S'),1,.STATUS);
      IF (OLD$AT AND INVISIBLE$ATTRIBUTE) <> 0 THEN
        CALL WRITE(0,.('I'),1,.STATUS);
      IF (OLD$AT AND FORMAT$ATTRIBUTE) <> 0 THEN
        CALL WRITE(0,.('F'),1,.STATUS);
      CALL WRITE(0,.(CR,LF),2,.STATUS);
      END;
    END;
  END;
  CALL FILE$ERROR(FOUND,FILE$NAME,FALSE);
END;
CALL EXIT;
END;

EOF
```

```
BINOBJ:
DO;

DECLARE VERSION$LEVEL LITERALLY '02H',
        EDIT$LEVEL LITERALLY '18H';

DECLARE VERSION (*) BYTE DATA (VERSION$LEVEL,EDIT$LEVEL);

$INCLUDE (:F2:CPYRT5.DTA)
$INCLUDE (:F2:CPYRT5.NOT)
/*$NOLIST*/
$INCLUDE (:F2:COMMON.LIT)
$INCLUDE (:F2:CHAR.LIT)
$INCLUDE (:F2:OPEN.LIT)
$INCLUDE (:F2:SEG.LIT)
$INCLUDE (:F2:RECTYP.LIT)
$INCLUDE (:F2:ERROR.LIT)
$INCLUDE (:F2:MEMCK.PEX)
$INCLUDE (:F2:READ.PEX)
$INCLUDE (:F2:WRITE.PEX)
$INCLUDE (:F2:EXIT.PEX)
$INCLUDE (:F2:OPEN.PEX)
$INCLUDE (:F2:CLOSE.PEX)
$INCLUDE (:F2:DBLANK.PEX)
$INCLUDE (:F2:DLIMIT.PEX)
$INCLUDE (:F2:FERROR.PEX)
$INCLUDE (:F2:FUPPER.PEX)
$INCLUDE (:F2:SEQ.PEX)
$INCLUDE (:F2:PATH.PEX)
$LIST

DECLARE BUFFER$SIZE ADDRESS;
DECLARE IBUF(3328) BYTE;
DECLARE IPTR ADDRESS;
DECLARE BIN$BASE ADDRESS;
DECLARE BIN$REC BASED BIN$BASE STRUCTURE(
                        LENGTH ADDRESS,
                        ADDR ADDRESS);
DECLARE BUFFER(128) BYTE;
DECLARE BUFFER$PTR ADDRESS, CHAR BASED BUFFER$PTR BYTE;
DECLARE (OUTPUT$PTR,INPUT$PTR) ADDRESS;
DECLARE ACTUAL ADDRESS;
DECLARE STATUS ADDRESS;
DECLARE (START,ENDFILE) BOOLEAN;
DECLARE (AFT$OUT,AFT$IN) ADDRESS;
DECLARE START$VALUE ADDRESS;
DECLARE RECORD$PTR ADDRESS;
DECLARE MEMORY$PTR ADDRESS, MEM BASED MEMORY$PTR BYTE;
/*                                  */
/*     CONTENT RECORD DEFINITION    */
/*                                    */
DECLARE CONTENT STRUCTURE(
                TYPE BYTE,
                LENGTH ADDRESS,
                SEG$ID BYTE,
                ADDR ADDRESS,
                DAT(1) BYTE
                ) AT (.MEMORY);
DECLARE JUNK BYTE;
DECLARE RECORD$ADDRESS ADDRESS;
DECLARE RLEN ADDRESS;
DECLARE TYPE BYTE;
DECLARE IN ADDRESS;
```

```
DECLARE OUT  ADDRESS;
DECLARE LEN$BIN ADDRESS;
DECLARE END$REC  BYTE;
DECLARE CHECKSUM BYTE;
/*                                */
/*     MODULE HEADER RECORD DEFINITION */
/*                                */
DECLARE MODHDR STRUCTURE(
               TYPE(1)  BYTE,
               LENGTH   ADDRESS,
               NAME$LEN  BYTE,
               NAME(31) BYTE,
               TRN$ID   BYTE,
               TRN$VN   BYTE,
               CHKSUM   BYTE);
/*                                */
/*     MODULE END RECORD DEFINITION */
/*                                */
DECLARE MODEND STRUCTURE(
               REC$TYPE  BYTE,
               LENGTH    ADDRESS,
               TYPE      BYTE,
               SEG$ID    BYTE,
               OFFSET    ADDRESS,
               CHKSUM    BYTE);
/*                                */
/*     MODULE END OF FILE RECORD    */
/*     DEFINITION                 */
/*                                */
DECLARE MODEOF STRUCTURE(
               TYPE    BYTE,
               LENGTH  ADDRESS,
               CHKSUM  BYTE);
/*                                */

OUT$RECORD:
  PROCEDURE (PTR);
    DECLARE PTR ADDRESS, CHAR BASED PTR(1) BYTE;
    DECLARE P1 ADDRESS, ADDR BASED P1 ADDRESS;
    DECLARE (I,STATUS) ADDRESS;
    DECLARE CHECKSUM BYTE;

    P1 = PTR + 1;
    CHECKSUM = 0;
    DO I = 0 TO ADDR + 1;
      CHECKSUM = CHECKSUM + CHAR(I);
    END;
    CHAR(ADDR+2) = 0-CHECKSUM;
    CALL WRITE(AFT$OUT,PTR,ADDR+3,.STATUS);
    CALL FILE$ERROR(STATUS,OUTPUT$PTR,TRUE);
  END OUT$RECORD;

GET$NEXT$BIN$RCD:
  PROCEDURE ;
      CALL READ(AFT$IN,.IBUF,LENGTH(IBUF),.ACTUAL,.STATUS);
      CALL FILE$ERROR(STATUS,INPUT$PTR,TRUE);
      IF ACTUAL = 0 THEN
      DO;
        CALL FILE$ERROR(EARLY$EOF,INPUT$PTR,TRUE);
      END;
  END GET$NEXT$BIN$RCD;
/*                                                    */
/*                                                    */
```

```
GET$NEXT$BIN$BYTE:
  PROCEDURE  BYTE;
    DECLARE TEMP BYTE;
/*                                  */
      IF IPTR = 0 THEN
        CALL GET$NEXT$BIN$RCD;
      TEMP = IBUF(IPTR);
      IF(IPTR:=IPTR + 1) = ACTUAL  THEN
      IPTR = 0;
      RETURN TEMP;
/*                                  */
END GET$NEXT$BIN$BYTE;


START = FALSE;
ENDFILE = FALSE;

/* *-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*
   *-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*

     BEGINNING OF MAIN PROGRAM.

   *-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*
   *-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-* */

                   /* INITIALIZE MODULE HEADER RECORD AREA */
MODHDR.TYPE(0) = MODHDR$TYPE;
DO IN = 1 TO SIZE(MODHDR) - 1;
  MODHDR.TYPE(IN) = 0;
END;
/*                                  */
INPUT$PTR = .(':CI: ');
CALL READ(1,.BUFFER,LENGTH(BUFFER),.ACTUAL,.STATUS);
CALL FILE$ERROR(STATUS,INPUT$PTR,TRUE);
BUFFER(ACTUAL) = CR;
CALL FORCE$UPPER(.BUFFER);
INPUT$PTR,BUFFER$PTR = DEBLANK(.BUFFER);
CALL OPEN(.AFT$IN,INPUT$PTR,READ$MODE,0,.STATUS);
CALL FILE$ERROR(STATUS,INPUT$PTR,TRUE);
BUFFER$PTR = DEBLANK(DELIMIT(BUFFER$PTR));
IF SEQ(.('TO '),BUFFER$PTR,3) THEN
DO;
  OUTPUT$PTR,BUFFER$PTR = DEBLANK(BUFFER$PTR+2);
  BUFFER$PTR = DEBLANK(DELIMIT(BUFFER$PTR));
END;
ELSE
DO;
  CALL FILE$ERROR(INVALID$SYNTAX,BUFFER$PTR,TRUE);
END;
IF CHAR <> CR THEN
  CALL FILE$ERROR(INVALID$SYNTAX,BUFFER$PTR,TRUE);
/*                                  */
/*                                  */
CALL OPEN(.AFT$OUT,OUTPUT$PTR,WRITE$MODE,0,.STATUS);
CALL FILE$ERROR(STATUS,OUTPUT$PTR,TRUE);
STATUS = PATH(INPUT$PTR,.MODHDR.NAME$LEN);
MODHDR.NAME$LEN = 6;
DO WHILE MODHDR.NAME(MODHDR.NAME$LEN-1) = 0;
  MODHDR.NAME$LEN = MODHDR.NAME$LEN - 1;
END;
MODHDR.LENGTH = MODHDR.NAME$LEN + 4;
MODHDR.TYPE(MODHDR.NAME$LEN+4) = 0; /* TRN I D */
MODHDR.TYPE(MODHDR.NAME$LEN+5) = 0; /* TRN U N */
```

```
/*
  OUTPUT MODULE HEADER RECORD
*/

CALL OUT$RECORD(.MODHDR);
/*                                        */
/*     ASSEMBLE AND OUTPUT CONTENT RECORD(S) */
/*                                        */
BUFFER$SIZE = MEMCK - .MEMORY ;
CONTENT.TYPE = CONTENT$TYPE;
CONTENT.SEG$ID = ABS$SEG;
IPTR = 0;
OUT = 0;  /* SET BEGINNING VALUE FOR OUTPUT POINTER */
RECORD$PTR = 0;
END$REC = FALSE;     /* RESET END OF BIN FILE FLAG */
DO WHILE NOT END$REC;
       /* WE ARE AT THE BEGINNING OF A BIN RECORD */
    RECORD$PTR = 0;  /* RESET BIN RECORD OFFSET */
    BIN$BASE = .IBUF +IPTR;  /* UPDATE BASE OF BIN RECORD STRUCTURE */
    DO IN = 1 TO 2;  /* DUMMY LOOP TO ADVANCE POINTER IS HERE */
                     /* TO INSURE THAT A NEW RECORD IS READ INTO */
                     /* IBUF IF A BIN RECORD ENDS ON A TRACK     */
                     /* BOUNDARY                                 */
      JUNK = GET$NEXT$BIN$BYTE;
    END;
    IF (RLEN:=BIN$RCD.LENGTH) = 0 THEN
      END$REC = TRUE;
    RECORD$ADDRESS = BINRCD.ADDR;
    CONTENT.ADDR = RECORD$ADDRESS + RECORD$PTR;
    DO IN = 1 TO 2;
      JUNK = GET$NEXT$BIN$BYTE;
    END;
/*                                  */
/*                                  */
  DO WHILE RLEN <> 0 ;
                                /* PROCESS BINARY RECORDS UNTIL */
                                /* A RECORD WITH A LENGTH FIELD */
                                /* OF ZERO IS FOUND             */
                                /*                              */
/*                                    */
/* TRANSFER DATA TO THE OUTPUT BUFFER */
/*                                    */
    DO WHILE RLEN <> 0 AND OUT <= BUFFER$SIZE - 2;
      CONTENT.DAT(OUT) = GET$NEXT$BIN$BYTE;
      OUT = OUT + 1;
      RECORD$PTR = RECORD$PTR + 1;
      RLEN = RLEN - 1;
    END;
/*                                  */
/* WE HAVE REACHED THE END OF THE   */
/* INPUT OR OUTPUT BUFFER           */
/*                                  */
    IF OUT >= BUFFER$SIZE - 2 THEN

    DO;  /* IT WAS THE END OF THE OUTPUT BUFFER */
      CONTENT.LENGTH = OUT + 4;
      OUT = 0;
      CALL OUT$RECORD(.MEMORY);
      CONTENT.ADDR = RECORD$ADDRESS + RECORD$PTR;  /* UPDATE */
                                /* BASE ADDRESS FOR NEXT RECORD */
    END;
/*                                */
```

```
    IF RLEN = 0 THEN
    DO;  /* END OF INPUT DATA IN THIS BUFFER */
      IF OUT > 0 AND RLEN = 0 THEN
      DO;  /* FLUSH A PARTIAL RECORD */
        CONTENT.LENGTH = OUT + 4;
        OUT = 0;
        CALL OUT$RECORD(.MEMORY);
      END;
    END;
  END;  /* END OF READ BINARY RECORD LOOP */
END;

    /*
      INITIALIZE, ASSEMBLE, AND
      OUTPUT MODULE END RECORD
    */

    MODEND.REC$TYPE = MODEND$TYPE;
    MODEND.LENGTH = 5;
    MODEND.TYPE = 1;
    MODEND.SEG$ID = 0;
    MODEND.OFFSET = RECORD$ADDRESS;  /* SET TRANSFER ADDRESS */
    IF START THEN
      MODEND.OFFSET = START$VALUE;  /* USER SPECIFIED START ADDRESS */
    CALL OUT$RECORD(.MODEND);
/*                                  */
/*    INITIALIZE, ASSEMBLE, AND     */
/*    OUTPUT THE                    */
/*    MODULE END OF FILE RECORD     */
/*                                  */
    MODEOF.TYPE = EOF$TYPE;
    MODEOF.LENGTH = 1;
    CALL OUT$RECORD(.MODEOF);

CALL CLOSE(AFT$IN,.STATUS);
CALL FILE$ERROR(STATUS,INPUT$PTR,TRUE);
CALL CLOSE(AFT$OUT,.STATUS);
CALL FILE$ERROR(STATUS,OUTPUT$PTR,TRUE);
CALL EXIT;
END;

EOF
```

```
CLI:
DO;

DECLARE VERSION$LEVEL LITERALLY '02H',
        EDIT$LEVEL LITERALLY '10H';

DECLARE VERSION (*) BYTE DATA (VERSION$LEVEL,EDIT$LEVEL);

$INCLUDE (:F2:CPYRT5.NOT)
$INCLUDE (:F2:CPYRT5.DTA)
/*$NOLIST*/
$INCLUDE (:F2:COMMON.LIT)
$INCLUDE (:F2:CHAR.LIT)
$INCLUDE (:F2:OPEN.LIT)
$INCLUDE (:F2:READ.PEX)
$INCLUDE (:F2:WRITE.PEX)
$INCLUDE (:F2:LOAD.PEX)
$INCLUDE (:F2:RESCAN.PEX)
$INCLUDE (:F2:EXIT.PEX)
$INCLUDE (:F2:DLINIT.PEX)
$INCLUDE (:F2:DBLANK.PEX)
$INCLUDE (:F2:FUPPER.PEX)
$INCLUDE (:F2:MEMCK.PEX)
$INCLUDE (:F2:SEQ.PEX)
$INCLUDE (:F2:FERROR.PEX)
$INCLUDE (:F2:MDSMON.PEX)
$LIST

DECLARE BUFFER(128) BYTE;
DECLARE DEBUG BOOLEAN;
DECLARE BUFFER$PTR ADDRESS, CHAR BASED BUFFER$PTR BYTE;
DECLARE (PATHNAME$PTR,ACTUAL,STATUS,ENTRY,RETSU) ADDRESS;

/* *-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*
   *-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*

   BEGINNING OF PROGRAM.

   *-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*
   *-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-* */

STACKPTR = MEMCK;
OUTPUT(0FCH) = 0FCH; /* ENABLE CONSOLE INTERRUPTS 0 AND 1 */
ENABLE;
OUTPUT(0FDH) = 20H; /* SEND END OF INTERRUPT COMMAND */
BUFFER$PTR = .(':CI: ');
CALL RESCAN(1,.STATUS);
IF STATUS = 0 THEN
DO;
  CALL READ(1,.BUFFER,LENGTH(BUFFER),.ACTUAL,.STATUS);
  CALL FILE$ERROR(STATUS,BUFFER$PTR,TRUE);
END;
DO FOREVER;
  CALL WRITE(0,.('-'),1,.STATUS);
  CALL READ(1,.BUFFER,LENGTH(BUFFER),.ACTUAL,.STATUS);
  CALL FILE$ERROR(STATUS,BUFFER$PTR,TRUE);
  BUFFER(ACTUAL) = CR;
  CALL FORCE$UPPER(.BUFFER);
  BUFFER$PTR = DEBLANK(.BUFFER);
  IF CHAR = ';' THEN CHAR = CR;
  IF CHAR <> CR THEN
  DO;
    /* NOW CHECK FOR DEBUG MODE (PATHNAME PRECEDED BY 'DEBUG' */
```

```
    DEBUG = FALSE; /* ASSUME NORMAL CASE, NOT DEBUGGING */
    IF SEQ(.('DEBUG'),BUFFER$PTR,5)
    AND (DELIMIT(BUFFER$PTR)=BUFFER$PTR+5) THEN
    DO;
      BUFFER$PTR = DEBLANK(DELIMIT(BUFFER$PTR+5));
      DEBUG = TRUE;
      IF CHAR = CR THEN CALL MONITOR;
    END;
    PATHNAME$PTR = BUFFER$PTR;
    BUFFER$PTR = DELIMIT(DEBLANK(BUFFER$PTR));
    CALL RESCAN(1,.STATUS);
    CALL READ(1,.BUFFER,BUFFER$PTR-.BUFFER,.ACTUAL,.STATUS);
    CALL FORCE$UPPER(PATHNAME$PTR);
    IF DEBUG THEN RETSW = 2; ELSE RETSW = 1;
    CALL LOAD(PATHNAME$PTR,0,RETSW,.ENTRY,.STATUS);
    CALL FILE$ERROR(STATUS,PATHNAME$PTR,FALSE);
    CALL READ(1,.BUFFER,LENGTH(BUFFER),.ACTUAL,.STATUS);
  END;
END;
END;

EOF
```

```
COPY:
DO;

DECLARE VERSION$LEVEL LITERALLY '03H',
        EDIT$LEVEL LITERALLY '01H';

DECLARE VERSION (*) BYTE DATA (VERSION$LEVEL,EDIT$LEVEL);

/*
     THIS VERSION OF COPY HAS BEEN MODIFIED TO DO BULK COPIES,
    SINGLE DRIVE COPIES, COPY ATTRIBUTES, AND OTHER ASSORTED
    WONDERFUL THINGS.
*/

$INCLUDE (:F2:CPYRT5.DTA)
$INCLUDE (:F2:CPYRT5.NOT)
/*$NOLIST*/
$INCLUDE (:F2:SEEK.LIT)
$INCLUDE (:F2:COMMON.LIT)
$INCLUDE (:F2:ATTRIB.LIT)
$INCLUDE (:F2:GETLAB.PEX)
$INCLUDE (:F2:CHKLAB.PEX)
$INCLUDE (:F2:MASCII.PEX)
$INCLUDE (:F2:DMEM.PEX)
$INCLUDE (:F2:GETDSK.PEX)
$INCLUDE (:F2:MPATH.PEX)
$INCLUDE (:F2:OPEN.LIT)
$INCLUDE (:F2:ERROR.LIT)
$INCLUDE (:F2:UNPATH.PEX)
$INCLUDE (:F2:SERROR.PEX)
$INCLUDE (:F2:DEVICE.LIT)
$INCLUDE (:F2:SEEK.PEX)
$INCLUDE (:F2:MEMCK.PEX)
$INCLUDE (:F2:OPEN.PEX)
$INCLUDE (:F2:CHAR.LIT)
$INCLUDE (:F2:ATTRIB.PEX)
$INCLUDE (:F2:READ.PEX)
$INCLUDE (:F2:WRITE.PEX)
$INCLUDE (:F2:CLOSE.PEX)
$INCLUDE (:F2:RENAME.PEX)
$INCLUDE (:F2:EXIT.PEX)
$INCLUDE (:F2:DBLANK.PEX)
$INCLUDE (:F2:MDELIM.PEX)
$INCLUDE (:F2:FUPPER.PEX)
$INCLUDE (:F2:FERROR.PEX)
$INCLUDE (:F2:SEM.PEX)
$INCLUDE (:F2:UCASE.PEX)

$LIST


DECLARE TARGET$MODE BYTE;  /* UPDATE OR WRITE */
DECLARE PN(10) BYTE;  /* HOLDS INTERNAL FILENAME */
DECLARE (BUFFER$SIZE,SIZE) ADDRESS;  /* SIZE OF BUFFER USED, TOTAL SIZE */
DECLARE BUFFER(128) BYTE;  /* INPUT BUFFER FOR READ */
DECLARE SWITCH$PTR ADDRESS,CHAR BASED SWITCH$PTR BYTE;  /* PTR TO SWITCHES ON INPUT */
DECLARE PTR ADDRESS AT (.SWITCH$PTR);  /* PTR USED IN CON */
DECLARE BUFFER$PTR ADDRESS;  /* PTR TO INPUT BUFFER */
DECLARE TO$PTR ADDRESS,KEY BASED TO$PTR BYTE; /* WILL POINT TO KEYWORD 'TO' IN INPUT */
DECLARE (ACTUAL,STATUS) ADDRESS;  /* USED IN SYSTEM CALLS */
DECLARE I BYTE;  /* INDEX VARIABLE */
```

```
DECLARE FILE$NAME(15) BYTE;  /* HOLDS EXTERNAL FILENAME */
DECLARE I$NO BYTE;  /* INDEX OF DIRECTORY */
DECLARE (AFTN,ECHO$AFTN,DIR$AFTN,OUT$AFTN) ADDRESS;  /* FILE AFTN, DIRECTORY AFTN */
DECLARE NEXT ADDRESS;  /* PTR TO BUFFER WHICH HOLDS FILES TO BE COPIED */
DECLARE FILE BASED NEXT STRUCTURE  /* HEADER BLOCK FOR FILES */
    (OLDFILE(10) BYTE,  /* FILE TO BE COPIED */
     ATTRIB BYTE,  /* ATTRIBUTES OF THE FILE */
     NEWFILE(10) BYTE,  /* DESTINATION NAME OF FILE */
     MODE BYTE,  /* WRITE - UPDATE */
     LENGTH ADDRESS,  /* LENGTH OF FILE */
     ATEOF BYTE,  /* FALSE-ADD AT BEGINING, TRUE AT EOF */
     DONE BYTE,   /* TRUE-IF FILE IS DONE, FALSE-IF MORE TO FOLLOW */
     BEGIN BYTE);
DECLARE QUERY BOOLEAN,  /* QUERY SWITCH */
        FIRST BOOLEAN,  /* USED BY LABEL CHECKING */
        SYSTEM BOOLEAN,  /* COPY SYSTEM FILES? */
        NONSYSTEM BOOLEAN,  /* COPY NONSYSTEM FILES? */
        COPY$AT BOOLEAN,  /* COPY ATTRIBUTES? */
        NO$GOOD BOOLEAN,  /* INTERNAL LOOP CONDITIONAL */
        THE$ONE BOOLEAN,  /*  USED BY QUERY IN WILD$CARD  */
        SAME BOOLEAN,  /* USED IN WILDCARD FOR AUTOMATIC SINGLE$DRIVE */
        PRINT BOOLEAN,  /* PRINT THE BUFFER? */
        AMBIG BOOLEAN,  /* IS WILDCARD NAME AMBIGUOUS? */
        BRIEF BOOLEAN,  /* AUTOMATICALLY DELETE FILE IF EXISTS */
        CONCAT BOOLEAN;  /* CONCATENATION OR COPY? */
DECLARE SINGLE$DRIVE BOOLEAN PUBLIC;  /* SINGLE$DRIVE? */
DECLARE TEMP(128) BYTE;  /* TEMPORARY BUFFER */
DECLARE ATTRIB$LIST(*) BYTE DATA(1,2,4);  /* USED TO SET ATTRIBUTES */
DECLARE PAST$BYTE$LENGTH ADDRESS,  /* USED TO SEEK TO SPOT IN FILE */
        PAST$BLK$LENGTH ADDRESS;  /* USED TO SEEK TO BLOCK IN FILE */
DECLARE FILE$COUNT BYTE;  /* NUMBER OF FILES IN BUFFER */
DECLARE SOURCE(10) BYTE;  /* INTERNAL SOURCE NAME */
DECLARE OUTPUT(10) BYTE;  /* INTERNAL OUTPUT NAME */
DECLARE DIR$FILE(*) BYTE INITIAL (':FX:ISIS.DIR',0);
DECLARE IN$LABEL(9) BYTE;  /* LABEL OF SOURCE DISKETTE */
DECLARE OUT$LABEL(9) BYTE;  /* LABEL OF OUTPUT DISKETTE */
DECLARE ECHO$FILE(4) BYTE INITIAL (':X0:');
DECLARE (BYTE$TEMP,BLK$TEMP) ADDRESS;
DECLARE WRITE$PROTECT$FOUND BOOLEAN;  /* IF AN OUTPUT FILE IS WRITE PROTECTED */




LOOKUP:
    PROCEDURE(FILE$PTR,NO$RETURN) BOOLEAN;

    /* FILE$PTR - PTR TO A EXTERNAL FILE NAME
       NO$RETURN - BOOLEAN, TO FIND OUT WHETHER TO PRINT DELETE MESSAGE
       ==================================
       RETURNS TRUE - IF FILE DOES NOT EXIST,
                      OR IT CAN BE DELETED (NO$RETURN = TRUE)
              FALSE - OTHERWISE                                  */

       DECLARE FILE$PTR ADDRESS;
       DECLARE NO$RETURN BOOLEAN;


       CALL RENAME(FILE$PTR,FILE$PTR,.STATUS);
       IF STATUS <> NO$SUCH$FILE THEN
          DO;

          /* SPECIAL EXIT POINT FOR WRITE PROTECTED FILES */
```

```
      IF STATUS = WRITE$PROTECT THEN
         DO;
         IF CONCAT THEN RETURN FALSE;
            ELSE RETURN TRUE;
         END;

      IF STATUS <> MULTIDEFINED THEN
         CALL S$FILE$ERROR(STATUS,FILE$PTR);

      IF NOSRETURN THEN RETURN FALSE;

      CALL WRITE(0,FILE$PTR,WDELIMIT(FILE$PTR) - FILE$PTR,.STATUS);
      CALL WRITE(0,.(' FILE ALREADY EXISTS',CR,LF,'DELETE? '),30,.STATUS);
      CALL READ(1,.TEMP,128,.ACTUAL,.STATUS);

      IF UPPERCASE(TEMP(0)) <> 'Y' THEN RETURN FALSE;

      END;

   RETURN TRUE;

   END;


FILE$PRINT:
   PROCEDURE;

      /* PRINTS:  <SOURCE> TO <OUTPUT>  */

   CALL UNPATH(.FILE.OLDFILE,.TEMP);
   CALL WRITE(0,.TEMP,WDELIMIT(.TEMP)-.TEMP,.STATUS);
   CALL WRITE(0,.(' TO '),4,.STATUS);
   CALL UNPATH(.FILE.NEWFILE,.TEMP);
   CALL WRITE(0,.TEMP,WDELIMIT(.TEMP)-.TEMP,.STATUS);

   END;



ARRANGE:
   PROCEDURE;

   /* EITHER GETS THE LABEL OF THE DISKETTE (FIRST = TRUE),
      OR IF IT ALREADY HAS IT, IT CHECKS IT AGAINST THE CURRENT
      LABEL TO SEE IF THEY MATCH                                  */

   IF FIRST THEN
      DO;

      CALL GET$DISK(2);
      CALL GET$LABEL(.OUT$LABEL,OUTPUT(0));
      FIRST = FALSE;

      END;
      ELSE
         CALL CHECK$LABEL(.OUT$LABEL,OUTPUT(0),2);

   END;



WRITE$BUFFER:
```

```
PROCEDURE;

/* AFTER THE BUFFER HAS BEEN FILLED BY SUCCESSIVE CALLS TO
   READ$BUFFER, THIS ROUTINE TRANSFERS ALL THE FILES THAT IT
   CAN TO THE APPROPRIATE OUTPUT DEVICE.  IT CHECKS TO SEE IF
   THE OUTPUT FILE ALREADY EXISTS, AND WHETHER IT
   SHOULD BE DELETED (DONE AUTOMATICALLY WITH U OR B SWITCHES)  */

   DECLARE COPY BOOLEAN;  /* PRINT THIS FILE? */


   NEXT = .MEMORY;

   DO WHILE FILE$COUNT <> 0;
      COPY = TRUE;

      CALL UNPATH(.FILE.NEWFILE,.FILE$NAME);

      /* CHECK IF OUTPUT FILE EXISTS, AND DETERMINE WHETHER TO OVERWRITE */
      IF NOT BRIEF AND FILE.MODE <> UPDATE$MODE
         AND FILE.NEWFILE(0) <= F5DEV THEN
            COPY = LOOKUP(.FILE$NAME,FALSE);

       IF COPY THEN
          IF (NOT FILE.ATEOF) OR SINGLE$DRIVE THEN
             DO;
             CALL OPEN(.OUT$AFTN,.FILE$NAME,FILE.MODE,0,.STATUS);
             IF STATUS <> WRITE$PROTECT OR CONCAT THEN
                CALL S$FILE$ERROR(STATUS,.FILE$NAME);
                ELSE DO;
                   CALL FILE$ERROR(STATUS,.FILE$NAME,FALSE);
                   COPY = FALSE;
                   WRITE$PROTECT$FOUND = TRUE;
                   END;
             END;

       IF COPY THEN
          DO;

          IF FILE.ATEOF AND (FILE.NEWFILE(0) <= F5DEV) AND (SINGLE$DRIVE) THEN
             DO;
             CALL SEEK(OUT$AFTN,SEEK$ABS,.BLK$TEMP,.BYTE$TEMP,.STATUS);
             CALL S$FILE$ERROR(STATUS,.FILE$NAME);
             END;
          CALL WRITE(OUT$AFTN,.FILE.BEGIN,FILE.LENGTH,.STATUS);
          CALL S$FILE$ERROR(STATUS,.FILE$NAME);
          IF (FILE.DONE AND NOT CONCAT) OR
             (CONCAT AND PTR >= TO$PTR AND (FILE$COUNT = 1)) OR
              SINGLE$DRIVE THEN
             DO;
             IF CONCAT THEN
                CALL SEEK(OUT$AFTN,SEEK$RETURN,.BLK$TEMP,.BYTE$TEMP,.STATUS);
             CALL CLOSE(OUT$AFTN,.STATUS);
             CALL S$FILE$ERROR(STATUS,.FILE$NAME);
             END;

          IF FILE.DONE THEN
             DO;
             IF COPY$AT AND (FILE.NEWFILE(0) <= F5DEV) THEN        /* COPY ATTRIBUTES */
                DO I = 0 TO LAST(ATTRIB$LIST);
                IF (FILE.ATTRIB AND ATTRIB$LIST(I)) <> 0 THEN
                    DO;
                    CALL ATTRIB(.FILE$NAME,I,TRUE,.STATUS);
```

```
                    CALL S$FILE$ERROR(STATUS,.FILE$NAME);
                    END;
                 END;

              IF CONCAT THEN
                 CALL WRITE(0,.('APPENDED '),9,.STATUS);
                 ELSE
                    CALL WRITE(0,.('COPIED '),7,.STATUS);
              CALL FILE$PRINT;
              CALL WRITE(0,.(CR,LF),2,.STATUS);

              END;

           END;
           ELSE  /* DO NOT COPY THIS FILE */
              DO;
              IF CONCAT THEN
                 PTR = TO$PTR;
                 ELSE

                    /* IF FILE IS SPLIT, MAKE SURE NOT TO PRINT SECOND PART */
                    IF NOT FILE.DONE THEN
                       DO;
                       ISNO = ISNO + 1;
                       FILE$COUNT,PAST$BYTE$LENGTH,PAST$BLK$LENGTH = 0;
                       RETURN;
                       END;
              END;

        NEXT = .FILE.BEGIN + FILE.LENGTH;
        FILE$COUNT = FILE$COUNT - 1;
        END;

    END;


READ$BUFFER:
   PROCEDURE BOOLEAN;

   /* READS A FILE INTO THE BUFFER AND FILLS UP THE HEADER BLOCK */

   DECLARE START$ADD ADDRESS;


      /* ASSUMES THAT THE HEADER HAS BEEN FILLED WITH OLD AND NEW FILE NAMES */


      FILE$COUNT=FILE$COUNT + 1;
      BUFFER$SIZE = BUFFER$SIZE - 26;  /* LENGTH OF FILE STRUCTURE */
      CALL UNPATH(.FILE.OLDFILE,.FILE$NAME);
      IF FILE.OLDFILE(0) = TIDEV OR FILE.OLDFILE(0) = VIDEV THEN
         DO;
         IF FILE.OLDFILE(0) = VIDEV THEN ECHO$FILE(1) = 'V';
            ELSE ECHO$FILE(1) = 'T';
         CALL OPEN(.ECHO$AFTN,.ECHO$FILE,WRITE$MODE,0,.STATUS);
         CALL OPEN(.AFTN,.FILE$NAME,READ$MODE,ECHO$AFTN,.STATUS);
         END;
         ELSE
         CALL OPEN(.AFTN,.FILE$NAME,READ$MODE,0,.STATUS);
      CALL S$FILE$ERROR(STATUS,.FILE$NAME);
      FILE.LENGTH = 0;
      ACTUAL = 1;
      START$ADD =  FILE.AFTN;
```

```
      /* IF PART OF THE FILE HAS BEEN READ ALREADY, THEN SEEK TO THAT POINT */
      IF PAST$BYTE$LENGTH <> 0 OR PAST$BLK$LENGTH <> 0 THEN
         DO;
           IF FILE.NEWFILE(0) <= F5DEV THEN FILE.MODE = UPDATE$MODE;
           FILE.ATEOF = TRUE;
           IF FILE.OLDFILE(0) <= F5DEV THEN
             DO;
             CALL SEEK (AFTN,SEEK$ABS,.PAST$BLK$LENGTH,.PAST$BYTE$LENGTH,.STATUS);
             CALL S$FILE$ERROR (STATUS,.FILE$NAME);
             END;
         END;

      /* PERFORM THE READ, STOP WHEN ACTUAL IS 0 */
      DO WHILE (ACTUAL <> 0) AND (BUFFER$SIZE <> 0);
         CALL READ(AFTN,START$ADD,BUFFER$SIZE,.ACTUAL,.STATUS);
         CALL S$FILE$ERROR(STATUS,.FILE$NAME);
         BUFFER$SIZE=BUFFER$SIZE - ACTUAL;
         FILE.LENGTH = ACTUAL + FILE.LENGTH;
         START$ADD = START$ADD + ACTUAL;
         END;

      IF FILE.OLDFILE(0) = VIDEV OR FILE.OLDFILE(0) = TIDEV THEN
         CALL CLOSE(ECHO$AFTN,.STATUS);
      CALL CLOSE(AFTN,.STATUS);
      CALL S$FILE$ERROR(STATUS,.FILE$NAME);
      IF (PAST$BYTE$LENGTH <> 0 OR PAST$BLK$LENGTH <> 0) AND (NOT CONCAT) THEN
        DO;
          BYTE$TEMP = PAST$BYTE$LENGTH;
          BLK$TEMP = PAST$BLK$LENGTH;
        END;

      /* THIS MEANS THE COMPLETE FILE WOULD NOT FIT IN THE BUFFER */
      IF BUFFER$SIZE = 0 THEN
         DO;
         PAST$BLK$LENGTH = PAST$BLK$LENGTH + SHR(PAST$BYTE$LENGTH,7);
         PAST$BYTE$LENGTH = (PAST$BYTE$LENGTH MOD 128) + FILE.LENGTH;
         FILE.DONE = FALSE;
         RETURN TRUE;
         END;

      /* THE FILE FITS IN THE BUFFER */
      PAST$BLK$LENGTH,PAST$BYTE$LENGTH = 0;
      FILE.DONE = TRUE;
      NEXT = .FILE.BEGIN + FILE.LENGTH;
      RETURN FALSE;

      END;



WILD$CARD:
   PROCEDURE;
    DECLARE DISK BYTE;
    DECLARE (FINISHED,NO$FILE) BOOLEAN;

    /* THIS PROCEDURE IS CALLED WHEN THE USER IS NOT GOING TO
       CONCATENATE ANY FILES TOGETHER.  IT WILL COPY ONE, OR MANY
       FILES FROM A DEVICE, AND SEE THAT THEY ARE WRITTEN TO THE
       PROPER OUTPUT DEVICE.  IF THE INPUT DEVICE IS A DISK, THEN
       WILD$CARD OPENS THE DIRECTORY AND SEARCHES FOR ALL OCCURENCES
       OF THE PATHNAME IN THE DIRECTORY.  THE LOOPS, CHECK FOR THE
```

```
   FOLLOWING CONDITIONS. 1) IF WE HAVE REACHED THE END OF THE
   DIRECTORY, IN WHICH CASE WE ARE FINISHED. 2) IF THE BUFFER
   IS FILLED, IN WHICH CASE, WE WANT TO WRITE THE BUFFER OUT AND
   RETURN TO READ SOMEMORE.  3) IF THE BUFFER IS FILLED, AND WE HAVE
   NOT READ ALL OF A FILE, IN THIS CASE WE WANT TO READ OUT THE BUFFER
   AND THEN FINISH READING THE FILE AND APPEND IT TO THE ALREADY
   COPIED PORTION OF THE FILE.  THIS PROCEDURE ALSO CHECKS TO SEE
   IF THE OUTPUT FILE IS EQUAL TO THE SOURCE FILE, IN WHICH CASE
   IT AUTOMATICALLY GOES INTO SINGLE DRIVE MODE.                    */

I$NO = 0;
FINISHED,NO$FILE,SAME = TRUE;

CALL FILE$ERROR(UPATH(.BUFFER,.SOURCE),.BUFFER,TRUE);
STATUS = UPATH(BUFFER$PTR,.OUTPUT);
IF STATUS <> NULL$FILENAME THEN
   CALL FILE$ERROR(STATUS,BUFFER$PTR,TRUE);

IF OUTPUT(1) = 0 THEN  /* FILENAME = NULL THEN SET TO *.* */
   DO I = 1 TO 9;
      OUTPUT(I) = '?';
      END;

/*  TEST MASKS FOR SINGLE$DRIVE,AMBIG, AND ERROR  */
   DO I = 0 TO 9;
      IF SOURCE(I) = '?' OR SOURCE(I) = '*' THEN AMBIG = TRUE;
         ELSE IF SOURCE(I) <> OUTPUT(I) AND OUTPUT(I) <> '*'
                 AND OUTPUT(I) <> '?' THEN SAME = FALSE;
      IF (SOURCE(I) = '?' AND (OUTPUT(I) <> '?' AND OUTPUT(I) <> '*')) OR
         (SOURCE(I) = '*' AND OUTPUT(I) <> '*') OR
         (SOURCE(I) = 0 AND OUTPUT(I) = '?') THEN
            DO;
            /*  FILE MASK ERROR  */
            CALL WRITE(0,.('FILE MASK ERROR',CR,LF),17,.STATUS);
            CALL EXIT;
            END;
      END;

DIR$FILE(2) = (DISK := SOURCE(0)) + '0';
SOURCE(0) = 0;

SINGLE$DRIVE = SINGLE$DRIVE OR SAME;
IF SINGLE$DRIVE THEN
   DO;
   CALL GET$DISK(1);
   CALL GET$LABEL(.IN$LABEL,DISK);
   END;


DO WHILE I$NO <> 201;


   IF DISK <= F5DEV AND (I$NO = 0 OR SINGLE$DRIVE) THEN
      DO;
      CALL OPEN(.DIR$AFTN,.DIR$FILE,READ$MODE,0,.STATUS);
      CALL S$FILE$ERROR(STATUS,.DIR$FILE);
      END;

   NO$GOOD = TRUE;

   DO WHILE NO$GOOD;

      IF BUFFER$STZE > 100 THEN
```

```
DO;

IF DISK <= F5DEV THEN
   ISND = DMEQ(DIR$AFTN,.SOURCE,ISND,.FILE);
   ELSE DO;
      CALL MOVE$ASCII(.FILE.OLDFILE(1),.(0,0,0,0,0,0,0,0,0,0),9);
      FILE.ATTRIB = 0;
      END;
IF ISND = 201 THEN
   DO;
   NO$GOOD = FALSE;
   IF FILE$COUNT > 0 THEN PRINT = TRUE;
   END;
   ELSE
      DO;

         IF (NOT AMBIG) OR
            (((FORMAT$ATTRIBUTE AND FILE.ATTRIB)=0) AND
            ((NONSYSTEM AND
              ((SYSTEM$ATTRIBUTE AND FILE.ATTRIB)=0)) OR
            (SYSTEM AND
              ((SYSTEM$ATTRIBUTE AND FILE.ATTRIB)<>0)))) THEN
              DO;

              FILE.OLDFILE(0)=DISK;
              FILE.ATEOF = FALSE;
              FILE.MODE = TARGET$MODE;
              IF (FILE.NEWFILE(0) := OUTPUT(0)) <= F5DEV THEN
                 DO I = 1 TO 9;
                    IF (FILE.NEWFILE(I):=OUTPUT(I)) = '?' OR
                       OUTPUT(I) = '*' THEN
                       FILE.NEWFILE(I) = FILE.OLDFILE(I);
                       END;
                       ELSE CALL MOVE$ASCII(.FILE.NEWFILE(1),.(0,0,0,0,0,0,0,0,0,0),9);

              THE$ONE = TRUE;
              NO$FILE = FALSE;
              IF QUERY AND FINISHED THEN  /* PERFORM QUERY */
                 DO;
                 CALL WRITE(0,.('COPY '),5,.STATUS);
                 CALL FILE$PRINT;
                 CALL WRITE(0,.('? '),2,.STATUS);
                 CALL READ(1,.TEMP,128,.ACTUAL,.STATUS);
                 IF UPPERCASE(TEMP(0)) <> 'Y' THEN THE$ONE = FALSE;
                 END;

              FINISHED = TRUE;
              IF THE$ONE THEN
                 IF (PRINT := READ$BUFFER) THEN
                    DO;
                    ISND = ISND -1;
                    FINISHED = FALSE;
                    NO$GOOD = FALSE;
                    END;

              END;
      END;

END;
ELSE DO;  /* BUFFER$SIZE < 100, SO DUMP BUFFER */
    NO$GOOD = FALSE;
    PRINT = TRUE;
    END;
```

```
      IF FINISHED AND DISK > F5DEV THEN I$ND = 201;

      END;   /* WHILE */

   IF DISK <= F5DEV AND (I$ND = 0 OR SINGLE$DRIVE) THEN
      DO;
      CALL CLOSE(DIR$AFTN,.STATUS);
      CALL S$FILE$ERROR(STATUS,.DIR$FILE);
      END;

   IF PRINT THEN
      DO;

      /*  GET DISK  */
      IF SINGLE$DRIVE THEN CALL ARRANGE;

      CALL WRITE$BUFFER;
      PRINT = FALSE;

      /*   GET DISK   */
      IF SINGLE$DRIVE AND (I$ND <> 201) THEN CALL CHECK$LABEL(.IN$LABEL,DISK,1);

      BUFFER$SIZE = SIZE;
      NEXT = .MEMORY;

      END;


   END;  /* WHILE */

IF NO$FILE THEN CALL S$FILE$ERROR(NO$SUCH$FILE,.BUFFER);

IF SINGLE$DRIVE THEN CALL GET$DISK(0);

END;
```

```
CON:
   PROCEDURE;
    DECLARE (NON$EXIST,CHECK) BOOLEAN;

    /* THIS PROCEDURE PERFORMS THE OPERATION OF CONCATENATION.
       IF THERE IS A ',' IN THE COMMAND LINE, THEN THIS PROCEDUREIS
       INVOKED.  IT FIRST CHECKS TO SEE IF THE SOURCE FILE IS EQUAL
       TO THE OUTPUT FILE, AND THEN TO SEE IF THERE ARE ANY WILDCARD
       CHARACTERS IN THE LINE.  THEN IT GOES THROUGH THE INPUT BUFFER
       FILE BY FILE AND LOADS THE BUFFER TILL THE BUFFER IS FILLED, OR
       THERE ARE NO MORE FILES TO CONCATENATE.                        */

   CHECK = FALSE;
   PTR = DEBLANK(.BUFFER);
   CALL FILE$ERROR(UPATH(BUFFER$PTR,.OUTPUT),BUFFER$PTR,TRUE);

   DO WHILE PTR < TO$PTR;

      CALL FILE$ERROR(UPATH(PTR,.SOURCE),PTR,TRUE);
         IF SEQ(.SOURCE,.OUTPUT,10) THEN
            DO;
```

```
         /*  SOURCE FILE EQUALS DESTINATION FILE  */
         CALL WRITE(0,.('SOURCE FILE EQUALS OUTPUT FILE ERROR',CR,LF),38,.STATUS);
         CALL EXIT;
         END;
   PTR = DEBLANK(DEBLANK(WDELIMIT(PTR))+1);

   END;

SWITCHSPTR = .BUFFER;

DO WHILE CHAR <> CR;

   IF CHAR = '?' OR CHAR = '*' THEN
      DO;
      /* WILDCARD DELIMITER IN CONCATENATE */
      CALL WRITE(0,.('WILDCARD DELIMITERS DURING CONCATENATE',CR,LF),40,.STATUS);
      CALL EXIT;
      END;

   SWITCHSPTR = SWITCHSPTR + 1;

   END;

PTR = DEBLANK(.BUFFER);

FILE.ATEOF = FALSE;

COPYSAT = FALSE;

IF SINGLE$DRIVE THEN CALL GET$DISK(1);

DO WHILE PTR < TOSPTR;

   NONSEXIST = FALSE;
   CALL S$FILE$ERROR(WPATH(PTR,.FILE.OLDFILE),PTR);
   IF FILE.OLDFILE(0) < F5DEV THEN
      IF (NONSEXIST := LOOKUP(PTR,TRUE)) AND FILE$COUNT > 0 THEN
         PRINT = TRUE;

   IF NOT NONSEXIST THEN
      DO;

      CALL MOVE$ASCII(.FILE.NEWFILE,.OUTPUT,10);
      FILE.MODE = TARGET$MODE;

      IF NOT ( PRINT := READ$BUFFER) THEN
         DO;
         PTR = DEBLANK(DEBLANK(WDELIMIT(PTR))+1);
         IF PTR >= TOSPTR THEN PRINT = TRUE;
         END;
         ELSE
            DO;
            PRINT = TRUE;
            CHECK = TRUE;
            CALL GET$LABEL(.IN$LABEL,FILE.OLDFILE(0));
            END;

      IF OUTPUT(0) <= F5DEV THEN TARGET$MODE = UPDATE$MODE;

      END;

   IF PRINT THEN
      DO;
```

```
         IF SINGLE$DRIVE THEN CALL ARRANGE;

         CALL WRITE$BUFFER;

         PRINT = FALSE;

         IF SINGLE$DRIVE AND (PTR < TO$PTR)
            AND (NOT NON$EXIST) THEN
               DO;
               IF CHECK THEN
                  DO;
                  CALL CHECK$LABEL(.IN$LABEL,FILE.OLDFILE(0),1);
                  CHECK = FALSE;
                  END;
                  ELSE CALL GET$DISK(1);
               END;

         BUFFER$SIZE = SIZE;
         NEXT = .MEMORY;

         END;

      FILE.ATEOF = TRUE;

      IF NON$EXIST THEN CALL S$FILE$ERROR(NO$SUCH$FILE,PTR);

      END;   /* WHILE */

      IF SINGLE$DRIVE THEN CALL GET$DISK(0);

   END;



/* *-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*
   *-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*

     BEGINNING OF MAIN PROGRAM

   *-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*
   *-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-* */

AMBIG,WRITE$PROTECT$FOUND,PRINT,COPY$AT,BRIEF,SINGLE$DRIVE,QUERY,CONCAT = FALSE;
SYSTEM,FIRST,NONSYSTEM = TRUE;
FILE$COUNT=0;
TARGET$MODE = WRITE$MODE;
PAST$BLK$LENGTH,BYTE$TEMP,BLK$TEMP,PAST$BYTE$LENGTH = 0;
CALL READ(1,.BUFFER,LENGTH(BUFFER),.ACTUAL,.STATUS);
BUFFER(ACTUAL) = CR;
CALL FORCE$UPPER(.BUFFER);
/*
  ADVANCE POINTER TO KEYWORD 'TO' AND CHECK IT ...
*/
TO$PTR = DEBLANK(WDELIMIT(DEBLANK(.BUFFER)));
IF KEY = ',' THEN CONCAT = TRUE;  /* GO INTO CONCATENATION MODE */
DO WHILE KEY = ',';
  TO$PTR = DEBLANK(WDELIMIT(DEBLANK(TO$PTR+1)));
END;
IF SEQ(.('TO '),TO$PTR,3) THEN
  BUFFER$PTR = DEBLANK(WDELIMIT(TO$PTR));
ELSE CALL FILE$ERROR(INVALID$SYNTAX,TO$PTR,TRUE);
```

```
/*
  BUFFER$PTR NOW POINTS TO THE TARGET FILENAME STRING.
  ADVANCE SWITCH$PTR BEYOND TARGET FILE NAME, TO SWITCH (IF ANY)...
*/
SWITCH$PTR = DEBLANK(@DELIMIT(BUFFER$PTR));
DO WHILE CHAR <> CR;
  IF CHAR = 'U' THEN TARGET$MODE = UPDATE$MODE;
    ELSE
    IF CHAR = 'S' THEN NONSYSTEM = FALSE;
       ELSE
       IF CHAR = 'N' THEN SYSTEM = FALSE;
          ELSE
          IF CHAR = 'B' THEN BRIEF = TRUE;
             ELSE
             IF CHAR = 'C' THEN COPY$AT = TRUE;
                ELSE
                IF CHAR = 'P' THEN SINGLE$DRIVE = TRUE;
                   ELSE
                   IF CHAR = 'Q' THEN QUERY = TRUE;
                      ELSE
                      IF CHAR <> '$' THEN
                         CALL FILE$ERROR(UNRECOG$SWITCH,SWITCH$PTR,TRUE);
  IF CONCAT AND QUERY THEN
     CALL FILE$ERROR(INVALID$SYNTAX,SWITCH$PTR,TRUE);
  SWITCH$PTR = DEBLANK(SWITCH$PTR+1);
  END;
   IF NOT(SYSTEM OR NONSYSTEM) THEN SYSTEM,NONSYSTEM = TRUE;
  BUFFER$SIZE,SIZE = MEMCK - .MEMORY;
  NEXT = .MEMORY;
  IF CONCAT THEN
     DO;
       IF SINGLE$DRIVE THEN TARGET$MODE = UPDATE$MODE;
       CALL CON;
     END;
     ELSE CALL WILD$CARD;
  IF WRITE$PROTECT$FOUND THEN
     CALL WRITE(0,.('WRITE PROTECTED FILE ENCOUNTERED',CR,LF),34,.STATUS);
  CALL EXIT;
END;
```

```
DELETE:
DO;

DECLARE VERSION$LEVEL LITERALLY '02H',
        EDIT$LEVEL LITERALLY '28H';

DECLARE VERSION (*) BYTE DATA (VERSION$LEVEL,EDIT$LEVEL);

$INCLUDE (:F2:CPYRT5.NOT)
$INCLUDE (:F2:CPYRT5.DTA)
/*$NOLIST*/
$INCLUDE (:F2:COMMON.LIT)
$INCLUDE (:F2:CHAR.LIT)
$INCLUDE (:F2:ERROR.LIT)
$INCLUDE (:F2:OPEN.LIT)
$INCLUDE (:F2:OPEN.PEX)
$INCLUDE (:F2:CLOSE.PEX)
$INCLUDE (:F2:READ.PEX)
$INCLUDE (:F2:WRITE.PEX)
$INCLUDE (:F2:DELETE.PEX)
$INCLUDE (:F2:EXIT.PEX)
$INCLUDE (:F2:DLIMIT.PEX)
$INCLUDE (:F2:DBLANK.PEX)
$INCLUDE (:F2:FUPPER.PEX)
$INCLUDE (:F2:UCASE.PEX)
$INCLUDE (:F2:GETDSK.PEX)
$INCLUDE (:F2:FERROR.PEX)
$INCLUDE (:F2:SERROR.PEX)
$INCLUDE (:F2:UNPATH.PEX)
$INCLUDE (:F2:WPATH.PEX)
$INCLUDE (:F2:DNEQ.PEX)
$INCLUDE (:F2:WDELIM.PEX)
$INCLUDE (:F2:RERROR.PEX)
$LIST

DECLARE (STATUS,ACTUAL,LEN) ADDRESS;
DECLARE BUFFER (128) BYTE;
DECLARE BUFFER$PTR ADDRESS, CHAR BASED BUFFER$PTR BYTE;
DECLARE PATHNAME(15) BYTE;
DECLARE PN(10) BYTE;
DECLARE BUF16(16) BYTE;
DECLARE DIR$AFTN ADDRESS;
DECLARE (I$NO,DISK) BYTE;
DECLARE (QUERY,FOUND) BOOLEAN;
DECLARE FILE$NAME ADDRESS;
DECLARE DIR$FIL(*) BYTE INITIAL (':F0:ISIS.DIR ');
DECLARE SINGLE$DRIVE BOOLEAN PUBLIC;
DECLARE QUES(*) BYTE DATA (', DELETE? ');

GOOD$CHAR:
   PROCEDURE (ITEM) BOOLEAN;
   DECLARE ITEM BYTE;

   RETURN (ITEM >= 'A' AND ITEM <= 'Z')
       OR (ITEM >= '0' AND ITEM <= '9')
       OR (ITEM = '?')
       OR (ITEM = '*')
       OR (ITEM = '.');

   END;


/* *-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*
```

```
   *-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*

     BEGINNING OF MAIN PROGRAM.

   *-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*
   *-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-* */

CALL READ(1,.BUFFER,LENGTH(BUFFER),.ACTUAL,.STATUS);
BUFFER(ACTUAL) = CR;
I$NO = ACTUAL - 1;
CALL FORCE$UPPER(.BUFFER);
CALL FILE$ERROR(STATUS,.(':CI:'),TRUE);
BUFFER$PTR = DEBLANK(.BUFFER);
SINGLE$DRIVE = FALSE;
DO WHILE (NOT GOOD$CHAR(BUFFER(I$NO))) AND I$NO > 1;
   I$NO=I$NO-1;
   END;
IF BUFFER(I$NO)='P' AND BUFFER(I$NO-1)=' ' THEN
   DO;
   I$NO=I$NO-1;
   DO WHILE I$NO > 0 AND (NOT GOOD$CHAR(BUFFER(I$NO)));
      I$NO = I$NO - 1;
      END;


   IF NOT(BUFFER(I$NO) = ',' OR I$NO = 0) THEN
      DO;
      SINGLE$DRIVE = TRUE;
      BUFFER(I$NO+1) = CR;
      END;
   END;
IF SINGLE$DRIVE THEN CALL GET$DISK(1);
DO FOREVER;
  /*
    PROCESS WILDCARDS.
  */
  FOUND = FALSE;
  FILE$NAME = BUFFER$PTR;
  STATUS = WPATH(BUFFER$PTR,.PN);
  CALL S$FILE$ERROR(STATUS,BUFFER$PTR);
  BUFFER$PTR = DEBLANK(WDELIMIT(BUFFER$PTR));
  DIR$FIL(2) = (DISK:=PN(0)) + '0';
  PN(0) = 0;
  CALL OPEN(.DIR$AFTN,.DIR$FIL,READ$MODE,0,.STATUS);
  QUERY = FALSE;
  IF CHAR <> ',' AND CHAR <> CR THEN
  DO;
    IF CHAR = 'Q' AND (DELIMIT(BUFFER$PTR)-BUFFER$PTR=1) THEN
    DO;
      QUERY = TRUE;
      BUFFER$PTR = DEBLANK(DELIMIT(BUFFER$PTR));
    END;
    ELSE
    DO;
      CALL S$FILE$ERROR(INVALID$SYNTAX,BUFFER$PTR);
    END;
  END;
  I$NO = 0;
  DO WHILE I$NO < 200;
    I$NO = DMEQ(DIR$AFTN,.PN,I$NO,.BUF16);
    IF I$NO <= 200 THEN
    DO;
      FOUND = TRUE;
```

```
      BUF16(0) = DISK;
      CALL UNPATH(.BUF16,.PATHNAME);
      LEN = DELIMIT(.PATHNAME) - .PATHNAME;
      MEMORY(0) = 'Y';
      IF QUERY THEN
      DO;
        CALL WRITE(0,.(' '),1,.STATUS);
        CALL WRITE(0,.PATHNAME,LEN,.STATUS);
        CALL WRITE(0,.QUES,LENGTH(QUES),.STATUS);
        CALL READ(1,.MEMORY,128,.ACTUAL,.STATUS);
      END;
      IF UPPERCASE(MEMORY(0)) = 'Y' THEN
      DO;
        CALL WRITE(0,.(' '),1,.STATUS);
        CALL WRITE(0,.PATHNAME,LEN,.STATUS);
        CALL WRITE(0,.(','),1,.STATUS);
        CALL DELETE(.PATHNAME,.STATUS);
        IF STATUS <> 0 THEN CALL REPORT$ERROR(STATUS);
        ELSE
        DO;
          CALL WRITE(0,.(' DELETED',CR,LF),10,.STATUS);
        END;
      END;
    END;
  END;
  CALL CLOSE(DIR$AFTN,.STATUS);
  IF NOT FOUND THEN CALL FILE$ERROR(NO$SUCH$FILE,FILE$NAME,FALSE);
  IF CHAR = CR THEN
    DO;
    IF SINGLE$DRIVE THEN CALL GET$DISK(0);
    CALL EXIT;
    END;
  IF CHAR = ',' THEN
  DO;
    BUFFER$PTR = DEBLANK(BUFFER$PTR+1);
  END;
  ELSE
  DO;
    CALL S$FILE$ERROR(INVALID$SYNTAX,BUFFER$PTR);
  END;
END;
END;

EOF
```

```
DIR:
DO;
DECLARE VERSION$LEVEL LITERALLY '03H',
        EDIT$LEVEL LITERALLY '00H';

DECLARE VERSION (*) BYTE DATA (VERSION$LEVEL,EDIT$LEVEL);

/*
    THIS VERSION OF DIR HAS BEEN MODIFIED TO WORK ON THE EMDS,
    IT HAS A SINGLE DRIVE SWITCH (P), AND WILL WORK ON ANY DISK
    CONFIGURATION ALLOWED BY THE EMDS.
*/

$INCLUDE (:F2:CPYRT5.NOT)
$INCLUDE (:F2:CPYRT5.DTA)
/*$NOLIST*/
$INCLUDE (:F2:COMMON.LIT)
$INCLUDE (:F2:CHAR.LIT)
$INCLUDE (:F2:OPEN.LIT)
$INCLUDE (:F2:ATTRIB.LIT)
$INCLUDE (:F2:GETDSK.PEX)
$INCLUDE (:F2:ERROR.LIT)
$INCLUDE (:F2:DEVICE.LIT)
$INCLUDE (:F2:OPEN.PEX)
$INCLUDE (:F2:READ.PEX)
$INCLUDE (:F2:WRITE.PEX)
$INCLUDE (:F2:CLOSE.PEX)
$INCLUDE (:F2:EXIT.PEX)
$INCLUDE (:F2:FUPPER.PEX)
$INCLUDE (:F2:DLIMIT.PEX)
$INCLUDE (:F2:DBLANK.PEX)
$INCLUDE (:F2:FERROR.PEX)
$INCLUDE (:F2:D.PEX)
$INCLUDE (:F2:UNPATH.PEX)
$INCLUDE (:F2:DIRECT.DEX)
$INCLUDE (:F2:SEQ.PEX)
$INCLUDE (:F2:WDELIM.PEX)
$INCLUDE (:F2:WPATH.PEX)
$LIST

DECLARE (AFTN,ACTUAL,STATUS) ADDRESS;
DECLARE PN(10) BYTE;
DECLARE (DISK,I) BYTE;
DECLARE (INVIS,FAST) BOOLEAN INITIAL (FALSE,FALSE);
DECLARE BUFFER$PTR ADDRESS, CHAR BASED BUFFER$PTR BYTE;
DECLARE BUFFER(128) BYTE;
DECLARE (TOS,FORS) BYTE;
DECLARE SINGLE$DRIVE BOOLEAN PUBLIC;

/* *-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*
   *-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*

   BEGINNING OF MAIN PROGRAM.

   *-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*
   *-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-* */

TOS,FORS = 0;
SINGLE$DRIVE = FALSE;
DISK = 0FFH;
CALL READ(1,.BUFFER,LENGTH(BUFFER),.ACTUAL,.STATUS);
BUFFER(ACTUAL) = CR;
CALL FORCE$UPPER(.BUFFER);
```

```
BUFFER$PTR = DEBLANK(.BUFFER);
AFTN = 0;
DO I = 1 TO 9;
  PN(I) = '*';
END;
PN(0) = 0;
DO WHILE CHAR <> CR;
  IF SEA(BUFFER$PTR,.('FOR '),4) THEN
  DO;
    /*
      PROCESS WILDCARD.
    */
    IF (FORS := FORS + 1) = 2 THEN
      CALL FILE$ERROR(INVALID$SYNTAX,BUFFER$PTR,TRUE);
    BUFFER$PTR = DEBLANK(BUFFER$PTR+3);
    CALL FILE$ERROR(WPATH(BUFFER$PTR,.PN),BUFFER$PTR,TRUE);
    BUFFER$PTR = DEBLANK(WDELIMIT(BUFFER$PTR));
    IF NOT INVIS THEN
       DO;
       INVIS = TRUE;
       DO I = 1 TO 9;
          IF PN(I) = '?' OR PN(I) = '*' THEN INVIS = FALSE;
          END;
       END;
  END;
  ELSE
  IF SEA(.('TO '),BUFFER$PTR,3) THEN
  DO;
    /*
      PROCESS DESTINATION FILE.
    */
    IF (TOS := TOS + 1) = 2 THEN
      CALL FILE$ERROR(INVALID$SYNTAX,BUFFER$PTR,TRUE);
    BUFFER$PTR = DEBLANK(BUFFER$PTR+2);
    CALL OPEN(.AFTN,BUFFER$PTR,WRITE$MODE,0,.STATUS);
    CALL FILE$ERROR(STATUS,BUFFER$PTR,TRUE);
    BUFFER$PTR = DEBLANK(DELIMIT(BUFFER$PTR));
  END;
  ELSE
     DO;
     IF CHAR >= '0' AND CHAR < '6' THEN
         DISK=CHAR-'0';
         ELSE
         IF CHAR = 'I' THEN INVIS = TRUE;
            ELSE
            IF CHAR = 'F' THEN FAST = TRUE;
               ELSE
             IF CHAR = 'P' THEN SINGLE$DRIVE = TRUE;
               ELSE
               IF CHAR <> '$' THEN
                  CALL FILE$ERROR(UNRECOG$SWITCH,BUFFER$PTR,TRUE);
     BUFFER$PTR = DEBLANK(BUFFER$PTR+1);
     END;
END;
IF DISK <> 0FFH THEN PN(0) = DISK;
DISK = PN(0);
PN(0) = 0;
IF SINGLE$DRIVE THEN CALL GET$DISK(1);
CALL D(DISK,AFTN,FAST,INVIS,.PN);
IF SINGLE$DRIVE THEN CALL GET$DISK(0);
CALL EXIT;
END;
```

```
HEXOBJ:
DO;

DECLARE VERSION$LEVEL LITERALLY '02H',
        EDIT$LEVEL LITERALLY '18H';

DECLARE VERSION (*) BYTE DATA (VERSION$LEVEL,EDIT$LEVEL);

$INCLUDE (:F2:CPYRT5.DTA)
$INCLUDE (:F2:CPYRT5.NOT)
/*$NOLIST*/
$INCLUDE (:F2:COMMON.LIT)
$INCLUDE (:F2:CHAR.LIT)
$INCLUDE (:F2:OPEN.LIT)
$INCLUDE (:F2:SEG.LIT)
$INCLUDE (:F2:RECTYP.LIT)
$INCLUDE (:F2:MEMCK.PEX)
$INCLUDE (:F2:READ.PEX)
$INCLUDE (:F2:WRITE.PEX)
$INCLUDE (:F2:EXIT.PEX)
$INCLUDE (:F2:OPEN.PEX)
$INCLUDE (:F2:CLOSE.PEX)
$INCLUDE (:F2:DBLANK.PEX)
$INCLUDE (:F2:DLIMIT.PEX)
$INCLUDE (:F2:FUPPER.PEX)
$INCLUDE (:F2:SEQ.PEX)
$INCLUDE (:F2:SCANIN.PEX)
$INCLUDE (:F2:ERROR.LIT)
$INCLUDE (:F2:FERROR.PEX)
$INCLUDE (:F2:PATH.PEX)
$LIST

DECLARE BUFFER$SIZE ADDRESS;
DECLARE IBUF(3328) BYTE;
DECLARE IPTR ADDRESS;
DECLARE BUFFER(128) BYTE;
DECLARE BUFFER$PTR ADDRESS, CHAR BASED BUFFER$PTR BYTE;
DECLARE (OUTPUT$PTR,INPUT$PTR) ADDRESS;
DECLARE ACTUAL ADDRESS;
DECLARE STATUS ADDRESS;
DECLARE (START,ENDFILE) BOOLEAN;
DECLARE (AFT$OUT,AFT$IN) ADDRESS;
DECLARE START$VALUE ADDRESS;
DECLARE RECORD$PTR ADDRESS;
DECLARE MEMORY$PTR ADDRESS, MEM BASED MEMORY$PTR BYTE;
/*                                */
/*     CONTENT RECORD DEFINITION  */
/*                                */
DECLARE CONTENT STRUCTURE(
                TYPE BYTE,
                LENGTH ADDRESS,
                SEG$ID BYTE,
                ADDR ADDRESS,
                DAT BYTE
                ) AT (.MEMORY);
DECLARE RECORD$ADDRESS ADDRESS;
DECLARE RLEN BYTE;
DECLARE TYPE BYTE;
DECLARE I BYTE;
DECLARE CHECKSUM BYTE;
/*                                */
/*     MODULE HEADER RECORD DEFINITION */
/*                                */
```

```
DECLARE MODHDR STRUCTURE(
              TYPE(1)  BYTE,
              LENGTH   ADDRESS,
              NAME$LEN  BYTE,
              NAME(31) BYTE,
              TRN$ID   BYTE,
              TRN$VN   BYTE,
              CHKSUM   BYTE);
/*                                  */
/*     MODULE END RECORD DEFINITION */
/*                                  */
DECLARE MODEND STRUCTURE(
              REC$TYPE  BYTE,
              LENGTH    ADDRESS,
              TYPE      BYTE,
              SEG$ID    BYTE,
              OFFSET    ADDRESS,
              CHKSUM    BYTE);
/*                                  */
/*     MODULE END OF FILE RECORD    */
/*     DEFINITION                   */
/*                                  */
DECLARE MODEOF STRUCTURE(
              TYPE    BYTE,
              LENGTH  ADDRESS,
              CHKSUM  BYTE);
/*                                  */


DECLARE TEMP$PTR ADDRESS;
DECLARE MODLOC STRUCTURE (
              RECTYPE BYTE,
              LENGTH ADDRESS,
              SEGID BYTE,
              OFFSET ADDRESS,
              NAMELEN BYTE,
              NAME(35) BYTE);
DECLARE TEMP(17) BYTE;

OUT$RECORD:
  PROCEDURE (PTR);
    DECLARE PTR ADDRESS, CHAR BASED PTR(1) BYTE;
    DECLARE P1 ADDRESS, ADDR BASED P1 ADDRESS;
    DECLARE (I,STATUS) ADDRESS;
    DECLARE CHECKSUM BYTE;

    P1 = PTR + 1;
    CHECKSUM = 0;
    DO I = 0 TO ADDR + 1;
      CHECKSUM = CHECKSUM + CHAR(I);
    END;
    CHAR(ADDR+2) = 0-CHECKSUM;
    CALL WRITE(AFT$OUT,PTR,ADDR+3,.STATUS);
    CALL FILE$ERROR(STATUS,OUTPUT$PTR,TRUE);
  END OUT$RECORD;

GNC:
  PROCEDURE BYTE;

    IF IPTR = LENGTH(IBUF) THEN
    DO;
      CALL READ(AFT$IN,.IBUF,LENGTH(IBUF),.ACTUAL,.STATUS);
      CALL FILE$ERROR(STATUS,INPUT$PTR,TRUE);
```

```
      IF ACTUAL = 0 THEN
      DO;
        CALL FILE$ERROR(EARLY$EOF,INPUT$PTR,TRUE);
        CALL EXIT;
      END;
      IPTR = 0;
    END;
    IPTR = IPTR + 1;
    RETURN IBUF(IPTR-1) AND 7FH;
  END GNC;

HEX:
  PROCEDURE BYTE;
    DECLARE CHAR BYTE;

    IF (CHAR:=GNC) >= '0' AND CHAR <= '9' THEN RETURN CHAR - '0';
    IF CHAR >= 'A' AND CHAR <= 'F' THEN RETURN CHAR - 37H;
    RETURN 0FFH;
  END HEX;

BYTES:
  PROCEDURE BYTE;
    DECLARE CHAR BYTE;

    CHAR = SHL(HEX,4) + HEX;
    CHECKSUM = CHECKSUM + CHAR;
    RETURN CHAR;
  END BYTES;

START = FALSE;
ENDFILE = FALSE;

/* *-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*
   *-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*

     BEGINNING OF MAIN PROGRAM.

   *-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*
   *-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-* */

                    /* INITIALIZE MODULE HEADER RECORD AREA */
MODHDR.TYPE(0) = MODHDR$TYPE;
DO I = 1 TO SIZE(MODHDR) - 1;
  MODHDR.TYPE(I) = 0;
END;
/*                                            */
INPUT$PTR = .(':CI: ');
CALL READ(1,.BUFFER,LENGTH(BUFFER),.ACTUAL,.STATUS);
CALL FILE$ERROR(STATUS,INPUT$PTR,TRUE);
BUFFER(ACTUAL) = CR;
CALL FORCE$UPPER(.BUFFER);
INPUT$PTR,BUFFER$PTR = DEBLANK(.BUFFER);
CALL OPEN(.AFT$IN,INPUT$PTR,READ$MODE,0,.STATUS);
CALL FILE$ERROR(STATUS,INPUT$PTR,TRUE);
BUFFER$PTR = DEBLANK(DELIMIT(BUFFER$PTR));
IF SEQ(.('TO '),BUFFER$PTR,3) THEN
DO;
  OUTPUT$PTR,BUFFER$PTR = DEBLANK(BUFFER$PTR+2);
  BUFFER$PTR = DEBLANK(DELIMIT(BUFFER$PTR));
END;
ELSE
DO;
  CALL FILE$ERROR(INVALID$SYNTAX,OUTPUT$PTR,TRUE);
```

```
END;
DO WHILE CHAR <> CR;
  IF CHAR = '$' THEN
  BUFFER$PTR = DEBLANK(BUFFER$PTR + 1);
  IF SEQ(BUFFER$PTR,.('START'),5) THEN
  DO;
    START = TRUE;
    BUFFER$PTR = DEBLANK(BUFFER$PTR+5);
    IF CHAR <> '(' THEN
    DO;
      CALL FILE$ERROR(INVALID$SYNTAX,BUFFER$PTR,TRUE);
    END;
    BUFFER$PTR = BUFFER$PTR + 1;
    START$VALUE = SCAN$INTEGER(.BUFFER$PTR);
    BUFFER$PTR = DEBLANK(BUFFER$PTR);
    IF CHAR <> ')' THEN
    DO;
      CALL FILE$ERROR(INVALID$SYNTAX,BUFFER$PTR,TRUE);
    END;
    BUFFER$PTR = DEBLANK(BUFFER$PTR+1);
  END;                    /*END OF SEARCH LOOP */
  ELSE
  DO;                    /* UNRECOGNIZED  OPTION */
    CALL FILE$ERROR(UNRECOG$SWITCH,BUFFER$PTR,TRUE);
  END;
END;                  /* END OF COMMAND LINE SEARCH */
/*                                */
/*                                */
CALL OPEN(.AFT$OUT,OUTPUT$PTR,WRITE$MODE,0,.STATUS);
CALL FILE$ERROR(STATUS,OUTPUT$PTR,TRUE);
STATUS = PATH(INPUT$PTR,.MODHDR.NAME$LEN);
MODHDR.NAME$LEN = 6;
DO WHILE MODHDR.NAME(MODHDR.NAME$LEN-1) = 0;
  MODHDR.NAME$LEN = MODHDR.NAME$LEN - 1;
END;
MODHDR.LENGTH = MODHDR.NAME$LEN + 4;
MODHDR.TYPE(MODHDR.NAME$LEN+4) = 0; /* TRN I D */
MODHDR.TYPE(MODHDR.NAME$LEN+5) = 0; /* TRN V N */

/*
  OUTPUT MODULE HEADER RECORD
*/

CALL OUT$RECORD(.MODHDR);
/*                                      */
/*     ASSEMBLE AND OUTPUT CONTENT RECORD(S) */
/*                                        */
BUFFER$SIZE = MEMCK - .MEMORY - 64;
CONTENT.TYPE = CONTENT$TYPE;
CONTENT.LENGTH = 0;
CONTENT.SEG$ID = ABS$SEG;
CONTENT.ADDR = 0;
MEMORY$PTR = .MEMORY+6;
RECORD$PTR = 0;
IPTR = LENGTH(IBUF);
RLEN = 1;
DO WHILE RLEN <> 0;
  DO WHILE (CHAR := GNC) <> ':';
    IF CHAR >= '0' AND CHAR <= '9' THEN
      DO;
      DO WHILE GNC <> ' ';
        ;
        END;
```

```
    DO WHILE (CHAR := GNC) = ' ';
      ;
      END;
    MODLOC.NAME(0) = CHAR;
    I=1;
    DO WHILE (MODLOC.NAME(I) := GNC) <> ' ';
      I=I+1;
      END;
    MODLOC.NAME(I) = 0;
    MODLOC.NAMELEN = I;
    MODLOC.LENGTH = I+6;
    DO WHILE (CHAR := GNC) < '0' OR CHAR > '9';
      ;
      END;
    DO I = 0 TO 9; TEMP(I) = ' '; END;
    I=1;
    TEMP(0) = CHAR;
    DO WHILE (CHAR := GNC) <> ' ' AND CHAR <> '$' AND CHAR <> CR;
      TEMP(I) = CHAR;
      I=I+1;
      END;
    TEMP$PTR = .TEMP;  /* THIS IS LUDICRIOUS, BUT NEEDED DO TO SCAN$INTEGER */
    MODLOC.OFFSET=SCAN$INTEGER(.TEMP$PTR);
    MODLOC.SEGID = 0;
    MODLOC.RECTYPE = 12H;
    CALL OUT$RECORD(.MODLOC);
    END;
  END;
  CHECKSUM = 0;
  RLEN = BYTES;
  IF RLEN <> 0 THEN
  DO;
    RECORD$ADDRESS = BYTES*256 + BYTES;
    IF RECORD$PTR <> RECORD$ADDRESS OR
      CONTENT.LENGTH > BUFFER$SIZE THEN
    DO;
      IF CONTENT.LENGTH <> 0 THEN
      DO;
        CONTENT.LENGTH = CONTENT.LENGTH + 4;
        CALL OUT$RECORD(.MEMORY);
      END;
      CONTENT.LENGTH = 0;
      RECORD$PTR = RECORD$ADDRESS;
      MEMORY$PTR = .MEMORY+6;
      CONTENT.ADDR = RECORD$ADDRESS;
    END;
    TYPE = BYTES;
    DO I = 1 TO RLEN;
      MEM = BYTES;
      MEMORY$PTR = MEMORY$PTR + 1;
      RECORD$PTR = RECORD$PTR + 1;
      CONTENT.LENGTH = CONTENT.LENGTH + 1;
    END;
    TYPE = BYTES; /* COMPUTE CHECKSUM */
    IF CHECKSUM <> 0 THEN
    DO;
      CALL FILE$ERROR(CHECKSUM$ERROR,INPUT$PTR,TRUE);
      CALL EXIT;
    END;
  END;
  ELSE
  DO;
    IF CONTENT.LENGTH <> 0 THEN
```

```
    DO;
      CONTENT.LENGTH = CONTENT.LENGTH + 4;
      CALL OUT$RECORD(.MEMORY);
    END;

    /*
      INITIALIZE, ASSEMBLE, AND
      OUTPUT MODULE END RECORD
    */

    MODEND.REC$TYPE = MODEND$TYPE;
    MODEND.LENGTH = 5;
    MODEND.TYPE = 1;
    MODEND.SEG$ID = 0;
    MODEND.OFFSET = BYTES*256+BYTES;
    IF START THEN
      MODEND.OFFSET = START$VALUE;  /* START ADDRESS WAS SPECIFIED */
    CALL OUT$RECORD(.MODEND);
/*                                  */
/*    INITIALIZE, ASSEMBLE, AND     */
/*    OUTPUT THE                    */
/*    MODULE END OF FILE RECORD     */
/*                                  */
    MODEOF.TYPE = EOF$TYPE;
    MODEOF.LENGTH = 1;
    CALL OUT$RECORD(.MODEOF);
  END;
END;

CALL CLOSE(AFT$IN,.STATUS);
CALL FILE$ERROR(STATUS,INPUT$PTR,TRUE);
CALL CLOSE(AFT$OUT,.STATUS);
CALL FILE$ERROR(STATUS,OUTPUT$PTR,TRUE);
CALL EXIT;
END HEXOBJ;

EOF
```

```
IDISK:
DO;

DECLARE VERSION$LEVEL LITERALLY '03H',
        EDIT$LEVEL LITERALLY '00H';

DECLARE VERSION(*) BYTE DATA (VERSION$LEVEL,EDIT$LEVEL);

/*
    IDISK - THIS IS A CUSP FOR INITIALIZING DISKETTES.  IT WILL WORK
    ON SINGLE OR DOUBLE DENSITY DRIVES, AND IN SINGLE$DRIVE MODE.  IT
    DOES TWO THINGS.  1) NORMALLY, IT CREATES A NON-SYSTEM DISKETTE, BY
    PUTTING ISIS.DIR,ISIS.LAB,ISIS.TO,ISIS.MAP ONTO  THE DISKETTE.
    2) WITH THE S SWITCH, IT CREATES A SYSTEM DISKETTE, BY PUTTING
    T0BOOT, ISIS.BIN, AND ISIS.CLI ON THE DISKETTE.
    ON A 32K SYSTEM WITH SINGLE$DRIVE MODE (P SWITCH) IT WILL
    REQUIRE 4 DISKETTE SWAPS TO CREATE A SYSTEM DISKETTE.  ALL THE
    FILES THAT IDISK PUTS ON THE DISKETTE, HAVE THE FORMAT ATTRIBUTE
    SET.
    IN ORDER TO SIMULATE FORMAT USING IDISK, IT IS NECESSARY TO USE COPY
    TO COPY ALL THE FILES THAT YOU WANT ON THE DISKETTE.
*/

$INCLUDE (:F2:CPYRT5.NOT)
$INCLUDE (:F2:CPYRT5.DTA)
/*$NOLIST*/
$INCLUDE (:F2:COMMON.LIT)
$INCLUDE (:F2:DISK.LIT)
$INCLUDE (:F2:CHAR.LIT)
$INCLUDE (:F2:DEVICE.LIT)
$INCLUDE (:F2:GEOG.LIT)
$INCLUDE (:F2:ATTRIB.LIT)
$INCLUDE (:F2:ERROR.LIT)
$INCLUDE (:F2:ALLOC.DEX)
$INCLUDE (:F2:DIRECT.DEX)
$INCLUDE (:F2:NONSYS.BLK)
$INCLUDE (:F2:MEMCK.PEX)
$INCLUDE (:F2:OPEN.PEX)
$INCLUDE (:F2:OPEN.LIT)
$INCLUDE (:F2:READ.PEX)
$INCLUDE (:F2:WRITE.PEX)
$INCLUDE (:F2:CLOSE.PEX)
$INCLUDE (:F2:EXIT.PEX)
$INCLUDE (:F2:ATTRIB.PEX)
$INCLUDE (:F2:DELETE.PEX)
$INCLUDE (:F2:DISKIO.PEX)
$INCLUDE (:F2:GETDSK.PEX)
$INCLUDE (:F2:SERROR.PEX)
$INCLUDE (:F2:FERROR.PEX)
$INCLUDE (:F2:DBLANK.PEX)
$INCLUDE (:F2:DLIMIT.PEX)
$INCLUDE (:F2:MASCII.PEX)
$INCLUDE (:F2:CLBUF.PEX)
$INCLUDE (:F2:SETBLK.PEX)
$INCLUDE (:F2:ABSIO.PEX)
$INCLUDE (:F2:FMTTRK.PEX)
$INCLUDE (:F2:FUPPER.PEX)
$INCLUDE (:F2:WD.PEX)
$INCLUDE (:F2:SPATH.PEX)
$INCLUDE (:F2:UNPATH.PEX)
$LIST

DECLARE BUFFER(128) BYTE;
```

```
DECLARE MEM$SIZE ADDRESS;
DECLARE ACTUAL ADDRESS;
DECLARE PN(12) BYTE;
DECLARE BUFFER$PTR ADDRESS, CHAR BASED BUFFER$PTR BYTE;
DECLARE (I,J,K) ADDRESS;
DECLARE (FILE$NUMBER,HAD$PRINT) BYTE;
DECLARE NEXT ADDRESS;
DECLARE (SYSTEM,PRINT,FIRST,COPY) BOOLEAN;
DECLARE (AFT$IN,AFT$OUT,DIR$AFT) ADDRESS;
DECLARE STATUS ADDRESS;
DECLARE ATTRIB$LIST(*) BYTE DATA (1,2,4);
DECLARE DISK$TYPE BOOLEAN PUBLIC;
DECLARE INPUT$STRING(16) BYTE,
        OUTPUT$STRING(16) BYTE;
DECLARE FILE(6) STRUCTURE
   (NAME(13) BYTE)
   INITIAL (':F0:ISIS.BIN ',
            ':F0:ISIS.T0  ',
            ':F0:ISIS.CLI ',
            ':F0:ISIS.MAP ',
            ':F0:ISIS.DIR ',
            ':F0:ISIS.LAB ');
DECLARE LAB$BLK STRUCTURE(
                        NAME(9)       BYTE,
                        VERSION(2)    BYTE,
                        LEFT$OVER(38) BYTE,
                        CRLF(2)       BYTE,
                        FMT$TABLE(77) BYTE) AT (.BUFFER);
DECLARE SINGLE$DRIVE BOOLEAN PUBLIC;
DECLARE INFO BASED NEXT STRUCTURE
   (NUMBER BYTE,
    LENGTH ADDRESS,
    BEGIN BYTE);


INITIALIZE:
   PROCEDURE;

   /* THIS PROCEDURE IS CALLED TO DO THE PHYSICAL FORMAT OF THE
      DISKETTE, AND THEN SET UP THE FILES THAT MUST EXIST FOR THE
      REST OF THE COPYING TO TAKE PLACE                    */

   CALL FORMAT$TRACK(PN(0),0,0,LAB$BLK.FMT$TABLE(0)-'0');
   CALL FORMAT$TRACK(PN(0),1,1,LAB$BLK.FMT$TABLE(1)-'0');
   CALL FORMAT$TRACK(PN(0),2,76,LAB$BLK.FMT$TABLE(2)-'0');

   CALL WRITE$DIRECTORY(PN(0));
   FILE(5).NAME(2) = PN(0) + '0';

   CALL OPEN(.AFT$OUT,.FILE(5).NAME,UPDATE$MODE,0,.STATUS);
   CALL S$FILE$ERROR(STATUS,.FILE(5).NAME);

   CALL MOVE$ASCII(.LAB$BLK,.PN+1,9);

   CALL WRITE(AFTOUT,.LAB$BLK,SIZE(LAB$BLK),.STATUS);
   CALL S$FILE$ERROR(STATUS,.FILE(5).NAME);

   CALL CLOSE(AFTOUT,.STATUS);
   CALL S$FILE$ERROR(STATUS,.FILE(5).NAME);

   END;
```

```
/* *-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*
   *-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*

     BEGINNING OF MAIN PROGRAM.

   *-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*
   *-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-* */

NEXT = .MEMORY;
MEM$SIZE = MEMCK - .MEMORY;
FIRST = TRUE;
FILE$NUMBER,HAD$PRINT = 0;
SINGLE$DRIVE,PRINT,DISK$TYPE,SYSTEM = FALSE;
/*                                                              */
/* READ AND PARSE COMMAND TAIL                                  */
/*                                                              */
CALL READ(1,.BUFFER,LENGTH(BUFFER),.ACTUAL,.STATUS);
BUFFER(ACTUAL) = CR;
CALL FORCE$UPPER(.BUFFER);
BUFFER$PTR = DEBLANK(.BUFFER);
PN(0) = 0FFH;
CALL SPATH(BUFFER$PTR,.PN,.STATUS);
IF PN(11) = 1 THEN DISK$TYPE = TRUE;
CALL FILE$ERROR(STATUS,BUFFER$PTR,TRUE);
IF PN(0) > F5DEV OR CHAR <> ':'  THEN CALL FILE$ERROR(BAD$LABEL,BUFFER$PTR,TRUE);
BUFFER$PTR = DEBLANK(DELIMIT(BUFFER$PTR));
DO WHILE CHAR <> CR;
   IF CHAR = 'P' THEN SINGLE$DRIVE = TRUE;
   ELSE
      IF CHAR = 'S' THEN SYSTEM = TRUE;
      ELSE
         IF CHAR <> '$' THEN
            CALL FILE$ERROR(UNRECOG$SWITCH,BUFFER$PTR,TRUE);
   BUFFER$PTR = DEBLANK(BUFFER$PTR+1);
   END;
IF PN(0) = F0DEV THEN SINGLE$DRIVE = TRUE;

/* READ AND DECODE THE FORMAT TABLE FROM THE SOURCE DISKETTE         */
/* INTO MEMORY TO BE USED AS THE PROTOTYPE FOR THE NEW               */
/* DISKETTE THAT WE ARE GOING TO CREATE                              */
/*                                                                   */
CALL OPEN(.AFT$IN,.FILE(5).NAME,READ$MODE,0,.STATUS);
CALL FILE$ERROR(STATUS,.FILE(5).NAME,TRUE);

CALL READ(AFT$IN,.LAB$BLK,SIZE(LAB$BLK),.ACTUAL,.STATUS);
CALL FILE$ERROR(STATUS,.FILE(5).NAME,TRUE);

CALL CLOSE (AFT$IN,.STATUS);
CALL FILE$ERROR(STATUS,.FILE(5).NAME,TRUE);


IF NOT (LAB$BLK.VERSION(0) = '3' AND LAB$BLK.VERSION(1) >= '0') THEN
   DO;
   CALL WRITE(0,.('SYSTEM DISKETTE NOT COMPATIBLE WITH IDISK',CR,LF),43,.STATUS);
   CALL EXIT;
   END;

  /* SET UP THE INTERLEAVE FACTORS ON THE DISK  */
LAB$BLK.FMT$TABLE(0)='1';
IF DISK$TYPE THEN
   DO;
   LAB$BLK.FMT$TABLE(1)='0'+24;
   LAB$BLK.FMT$TABLE(2)='5';
```

```
   END;
   ELSE DO;
      LAB$BLK.FMT$TABLE(1)='0'+12;
      LAB$BLK.FMT$TABLE(2)='6';
      END;

DO I = 3 TO 76;
   LAB$BLK.FMT$TABLE(I) = LAB$BLK.FMT$TABLE(2);
   END;

IF SYSTEM THEN CALL WRITE(0,.('SYSTEM DISKETTE',CR,LF),17,.STATUS);

IF SYSTEM THEN

   DO WHILE FILE$NUMBER < 3;

      INFO.NUMBER = FILE$NUMBER;
      MEM$SIZE = MEM$SIZE - 3;

      CALL OPEN(.AFT$IN,.FILE(FILE$NUMBER).NAME,READ$MODE,0,.STATUS);
      CALL S$FILE$ERROR(STATUS,.FILE(FILE$NUMBER).NAME);

      CALL READ(AFT$IN,.INFO.BEGIN,MEM$SIZE,.ACTUAL,.STATUS);
      CALL S$FILE$ERROR(STATUS,.FILE(FILE$NUMBER).NAME);
      CALL CLOSE(AFT$IN,.STATUS);
      CALL S$FILE$ERROR(STATUS,.FILE(FILE$NUMBER).NAME);
      INFO.LENGTH = ACTUAL;
      NEXT = .INFO.BEGIN + ACTUAL;


      IF (MEM$SIZE := MEM$SIZE - ACTUAL) = 0 THEN
         DO;
         FILE$NUMBER = FILE$NUMBER - 1;
         PRINT = TRUE;
         END;

      IF FILE$NUMBER = 2 THEN PRINT = TRUE;

      FILE$NUMBER = FILE$NUMBER + 1;


      IF PRINT THEN
         DO;

         IF SINGLE$DRIVE THEN CALL GET$DISK(2);

         IF FIRST THEN CALL INITIALIZE;

         NEXT = .MEMORY;

         DO WHILE HAD$PRINT < FILE$NUMBER;

            FILE(HAD$PRINT).NAME(2) = PN(0) + '0';
            CALL OPEN(.AFT$OUT,.FILE(HAD$PRINT).NAME,UPDATE$MODE,0,.STATUS);
            CALL S$FILE$ERROR(STATUS,.FILE(HAD$PRINT).NAME);
            CALL WRITE(AFT$OUT,.INFO.BEGIN,INFO.LENGTH,.STATUS);
            CALL S$FILE$ERROR(STATUS,.FILE(HAD$PRINT).NAME);
            CALL CLOSE(AFTOUT,.STATUS);
            CALL S$FILE$ERROR(STATUS,.FILE(HAD$PRINT).NAME);

            NEXT = .INFO.BEGIN + INFO.LENGTH;
            HAD$PRINT = HAD$PRINT + 1;
```

```
            END;

         IF SINGLE$DRIVE AND FILE$NUMBER < 3 THEN CALL GET$DISK(0);

         FIRST,PRINT = FALSE;
         NEXT = .MEMORY;
         MEM$SIZE = MEMCK - .MEMORY;
         END;

      END;
      ELSE DO;
         /* CREATE NON-SYSTEM DISKETTE */
         IF SINGLE$DRIVE THEN CALL GET$DISK(2);
         CALL INITIALIZE;
         FILE(0).NAME(2) = PN(0) + '0';
         FILE(1).NAME(2) = PN(0) + '0';
         CALL OPEN(.AFT$OUT,.FILE(1).NAME,UPDATE$MODE,0,.STATUS);
         CALL S$FILE$ERROR(STATUS,.FILE(1).NAME);
         CALL WRITE(AFT$OUT,.NONSYS,LENGTH(NONSYS),.STATUS);
         CALL S$FILE$ERROR(STATUS,.FILE(1).NAME);
         CALL CLOSE(AFT$OUT,.STATUS);
         CALL S$FILE$ERROR(STATUS,.FILE(1).NAME);
         CALL DELETE(.FILE(0).NAME,.STATUS);
         CALL WRITE(0,.('NON-SYSTEM DISKETTE',CR,LF),21,.STATUS);
         END;

DO I = 0 TO 5;
   FILE(I).NAME(2) = PN(0) + '0';
   CALL ATTRIB(.FILE(I).NAME,3,TRUE,.STATUS);
   END;
   CALL ATTRIB(.FILE(2).NAME,0,TRUE,.STATUS);
   CALL ATTRIB(.FILE(2).NAME,1,TRUE,.STATUS);
  IF SINGLE$DRIVE THEN CALL GET$DISK(0);

CALL EXIT;

END IDISK;
```

```
OBJHEX:
DO;

DECLARE VERSION$LEVEL LITERALLY '02H',
        EDIT$LEVEL LITERALLY '19H';

DECLARE VERSION(*) BYTE DATA (VERSION$LEVEL,EDIT$LEVEL);

$INCLUDE (:F2:CPYRT5.DTA)
$INCLUDE (:F2:CPYRT5.NOT)
/*$NOLIST*/
$INCLUDE (:F2:ERROR.LIT)
$INCLUDE (:F2:COMMON.LIT)
$INCLUDE (:F2:CHAR.LIT)
$INCLUDE (:F2:OPEN.LIT)
$INCLUDE (:F2:SEG.LIT)
$INCLUDE (:F2:RECTYP.LIT)
$INCLUDE (:F2:MEMCK.PEX)
$INCLUDE (:F2:READ.PEX)
$INCLUDE (:F2:WRITE.PEX)
$INCLUDE (:F2:EXIT.PEX)
$INCLUDE (:F2:OPEN.PEX)
$INCLUDE (:F2:CLOSE.PEX)
$INCLUDE (:F2:SEA.PEX)
$INCLUDE (:F2:DLIMIT.PEX)
$INCLUDE (:F2:DBLANK.PEX)
$INCLUDE (:F2:NUMOUT.PEX)
$INCLUDE (:F2:FUPPER.PEX)
$INCLUDE (:F2:FERROR.PEX)
$INCLUDE (:F2:PATH.PEX)
$LIST

DECLARE BUFFER$COUNT ADDRESS;
DECLARE SEG$ID BYTE;
DECLARE RECLEN ADDRESS;
DECLARE TYPE BYTE;
DECLARE CHECKSUM BYTE;
DECLARE (I,J) ADDRESS;
DECLARE HEXLEN ADDRESS;
DECLARE ADDR ADDRESS;
DECLARE TEMP BYTE;
DECLARE BUFFER$SIZE ADDRESS;
DECLARE IPTR ADDRESS;
DECLARE BUFFER(128) BYTE;
DECLARE BUFFER$PTR ADDRESS, CHAR BASED BUFFER$PTR BYTE;
DECLARE (OUTPUT$PTR,INPUT$PTR) ADDRESS;
DECLARE ACTUAL ADDRESS;
DECLARE STATUS ADDRESS;
DECLARE (AFT$OUT,AFT$IN) ADDRESS;
/*
  HEXADECIMAL CONTENT RECORD.
*/
DECLARE HEXRECORD STRUCTURE(
                  HEADER BYTE,
                  LENGTH ADDRESS,
                  ADDR(2) ADDRESS,
                  TYPE ADDRESS,
                  DAT(16) ADDRESS,
                  CHKSUM ADDRESS,
                  TRAILER(2) BYTE);

/*
  HEXADECIMAL END RECORD.
```

```
*/
DECLARE ENDRECORD STRUCTURE(
                  HEADER BYTE,
                  LENGTH ADDRESS,
                  ADDR(2) ADDRESS,
                  TYPE ADDRESS,
                  CHKSUM ADDRESS,
                  TRAILER(2) BYTE);

GET$BYTE:
  PROCEDURE BYTE;

    IF BUFFER$COUNT = 0 THEN
    DO;
      CALL READ(AFT$IN,.MEMORY,BUFFER$SIZE,.BUFFER$COUNT,.STATUS);
      CALL FILE$ERROR(STATUS,INPUT$PTR,TRUE);
      IF BUFFER$COUNT = 0 THEN
      DO;
        CALL FILE$ERROR(EARLY$EOF,INPUT$PTR,TRUE);
        CALL EXIT;
      END;
      IPTR = 0;
    END;
    BUFFER$COUNT = BUFFER$COUNT - 1;
    IPTR = IPTR + 1;
    RETURN MEMORY(IPTR-1);
  END GET$BYTE;

GET$ADDRESS:
  PROCEDURE ADDRESS;

    RETURN GET$BYTE + GET$BYTE*256;
  END GET$ADDRESS;

/* *-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*
   *-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*

     BEGINNING OF MAIN PROGRAM.

   *-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*
   *-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-* */

/*
  INITIALIZE RECORD STRUCTURES.
*/
HEXRECORD.HEADER = ':';
HEXRECORD.TYPE = '00';
ENDRECORD.HEADER = ':';
ENDRECORD.TRAILER(0) = CR;
ENDRECORD.TRAILER(1) = LF;
/*
  READ AND PROCESS COMMAND TAIL.
*/
INPUT$PTR = .(':CI: ');
CALL READ(1,.BUFFER,LENGTH(BUFFER),.ACTUAL,.STATUS);
CALL FILE$ERROR(STATUS,INPUT$PTR,TRUE);
BUFFER(ACTUAL) = CR;
CALL FORCE$UPPER(.BUFFER);
INPUT$PTR,BUFFER$PTR = DEBLANK(.BUFFER);
CALL OPEN(.AFT$IN,INPUT$PTR,READ$MODE,0,.STATUS);
CALL FILE$ERROR(STATUS,INPUT$PTR,TRUE);
BUFFER$PTR = DEBLANK(DELIMIT(BUFFER$PTR));
IF SEQ(.('TO '),BUFFER$PTR,3) THEN
```

```
DO;
  OUTPUT$PTR,BUFFER$PTR = DEBLANK(BUFFER$PTR+2);
  BUFFER$PTR = DEBLANK(DELIMIT(BUFFER$PTR));
END;
ELSE
DO;
  CALL FILE$ERROR(INVALID$SYNTAX,OUTPUT$PTR,TRUE);
END;
IF CHAR <> CR THEN CALL FILE$ERROR(INVALID$SYNTAX,BUFFER$PTR,TRUE);
CALL OPEN(.AFT$OUT,OUTPUT$PTR,WRITE$MODE,0,.STATUS);
CALL FILE$ERROR(STATUS,OUTPUT$PTR,TRUE);
/*
  COMPUTE SIZE OF WORKSPACE.
*/
BUFFER$SIZE = MEMCK - .MEMORY;
BUFFER$COUNT = 0;
/*
  READ OBJECT RECORDS, WRITE HEXADECIMAL RECORDS.
*/
DO FOREVER;
  TYPE = GET$BYTE;
  IF TYPE >= RELOC$TYPE THEN
    CALL FILE$ERROR(BAD$REC$TYP,INPUT$PTR,TRUE);
  IF TYPE = MODEND$TYPE THEN
  DO;
    RECLEN = GET$ADDRESS;
    TEMP = GET$BYTE;
    TEMP = GET$BYTE;
    ADDR = GET$ADDRESS;
    CHECKSUM = LOW(ADDR) + HIGH(ADDR) + 1;
    CALL NUMOUT(0,16,'0',.ENDRECORD.LENGTH,2);
    CALL NUMOUT(ADDR,16,'0',.ENDRECORD.ADDR,4);
    CALL NUMOUT(1,16,'0',.ENDRECORD.TYPE,2);
    CALL NUMOUT(-CHECKSUM,16,'0',.ENDRECORD.CHKSUM,2);
    CALL WRITE(AFT$OUT,.ENDRECORD,SIZE(ENDRECORD),.STATUS);
    CALL CLOSE(AFT$IN,.STATUS);
    CALL CLOSE(AFT$OUT,.STATUS);
    CALL EXIT;
  END;
  IF TYPE <> CONTENT$TYPE THEN
  DO;
    RECLEN = GET$ADDRESS;
    DO I = 1 TO RECLEN;
      TEMP = GET$BYTE;
    END;
  END;
  ELSE
  DO;
    RECLEN = GET$ADDRESS;
    SEG$ID = GET$BYTE;
    ADDR = GET$ADDRESS;
    RECLEN = RECLEN - 4;
    DO WHILE RECLEN <> 0;
      HEXLEN = RECLEN;
      IF HEXLEN > LENGTH(HEXRECORD.DAT) THEN
        HEXLEN = LENGTH(HEXRECORD.DAT);
      RECLEN = RECLEN - HEXLEN;
      DO I = 0 TO LENGTH(HEXRECORD.DAT)+1;
        HEXRECORD.DAT(I) = 0A0DH;
      END;
      CHECKSUM = HEXLEN + LOW(ADDR) + HIGH(ADDR);
      CALL NUMOUT(HEXLEN,16,'0',.HEXRECORD.LENGTH,2);
      CALL NUMOUT(ADDR,16,'0',.HEXRECORD.ADDR,4);
```

```
      DO J = 0 TO HEXLEN - 1;
        ADDR = ADDR + 1;
        TEMP = GET$BYTE;
        CHECKSUM = CHECKSUM + TEMP;
        CALL NUMOUT(TEMP,16,'0',.HEXRECORD.DAT(J),2);
      END;
      CALL NUMOUT(-CHECKSUM,16,'0',.HEXRECORD.DAT(HEXLEN),2);
      CALL WRITE(AFT$OUT,.HEXRECORD,HEXLEN+HEXLEN+13,.STATUS);
    END;
    TEMP = GET$BYTE;
  END;
END;

END OBJHEX;

EOF
```

```
RENAME:
DO;

DECLARE VERSION$LEVEL LITERALLY '02H',
        EDIT$LEVEL LITERALLY '11H';

DECLARE VERSION (*) BYTE DATA (VERSION$LEVEL,EDIT$LEVEL);

$INCLUDE (:F2:CPYRT5.NOT)
$INCLUDE (:F2:CPYRT5.DTA)
/*$NOLIST*/
$INCLUDE (:F2:COMMON.LIT)
$INCLUDE (:F2:CHAR.LIT)
$INCLUDE (:F2:ERROR.LIT)
$INCLUDE (:F2:READ.PEX)
$INCLUDE (:F2:WRITE.PEX)
$INCLUDE (:F2:RENAME.PEX)
$INCLUDE (:F2:EXIT.PEX)
$INCLUDE (:F2:DELETE.PEX)
$INCLUDE (:F2:DLIMIT.PEX)
$INCLUDE (:F2:DBLANK.PEX)
$INCLUDE (:F2:FUPPER.PEX)
$INCLUDE (:F2:UCASE.PEX)
$INCLUDE (:F2:SEQ.PEX)
$INCLUDE (:F2:FERROR.PEX)
$LIST

DECLARE (ACTUAL,STATUS) ADDRESS;
DECLARE BUFFER (128) BYTE;
DECLARE BUFFER$PTR ADDRESS;
DECLARE ALREADY(*) BYTE DATA (', ALREADY EXISTS, DELETE? ');

/* *-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*
   *-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*

   BEGINNING OF MAIN PROGRAM

   *-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*
   *-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-* */

CALL READ(1,.BUFFER,LENGTH(BUFFER),.ACTUAL,.STATUS);
BUFFER(ACTUAL) = CR;
CALL FORCE$UPPER(.BUFFER);
CALL FILE$ERROR(STATUS,.(':CI: '),TRUE);
BUFFER$PTR = DEBLANK(DELIMIT(DEBLANK(.BUFFER)));
/*
  BUFFER$PTR SHOULD NOW POINT TO 'TO '.
*/
IF SEQ(.('TO '),BUFFER$PTR,3) THEN
DO;
  BUFFER$PTR = DEBLANK(BUFFER$PTR + 3);
  CALL RENAME(.BUFFER,.BUFFER,.STATUS);
  IF STATUS <> MULTIDEFINED THEN CALL FILE$ERROR(STATUS,.BUFFER,TRUE);
  CALL RENAME(.BUFFER,BUFFER$PTR,.STATUS);
  IF STATUS = MULTIDEFINED THEN
  DO;
    CALL WRITE(0,.(' '),1,.STATUS);
    /*
      WRITE OUT NEW FILE NAME.
    */
    CALL WRITE(0,BUFFER$PTR,DELIMIT(BUFFER$PTR)-BUFFER$PTR,
              .STATUS);
    CALL WRITE(0,.ALREADY,LENGTH(ALREADY),.STATUS);
```

```
    CALL READ(1,.MEMORY,128,.ACTUAL,.STATUS);
    IF UPPER$CASE(MEMORY(0)) = 'Y' THEN
    DO;
      CALL DELETE(BUFFER$PTR,.STATUS);
      CALL FILE$ERROR(STATUS,BUFFER$PTR,TRUE);
      CALL RENAME(.BUFFER,BUFFER$PTR,.STATUS);
      CALL FILE$ERROR(STATUS,BUFFER$PTR,TRUE);
    END;
    ELSE CALL EXIT;
  END;
  ELSE IF(STATUS=WRITE$PROTECT) THEN
    CALL FILE$ERROR(STATUS,.BUFFER,TRUE);
    ELSE
      CALL FILE$ERROR(STATUS,BUFFER$PTR,TRUE);
END;
ELSE CALL FILE$ERROR(INVALID$SYNTAX,BUFFER$PTR,TRUE);
CALL EXIT;
END;

EOF
```

```
SUBMIT:
DO;

DECLARE VERSION$LEVEL LITERALLY '02H',
          EDIT$LEVEL LITERALLY '12H';

DECLARE VERSION (*) BYTE DATA (VERSION$LEVEL,EDIT$LEVEL);

$INCLUDE (:F2:CPYRT5.NOT)
/*
    THIS VERSION OF SUBMIT HAS BEEN MODIFIED TO WORK ON DOUBLE DENSITY.
    IT RECOGNIZES DISK DRIVES 4 AND 5.
*/

$INCLUDE (:F2:CPYRT5.DTA)
/*
 THIS CUSP MAY BE CALLED BY EITHER OF 2 COMMAND STRINGS:

 1.  -SUBMIT RESTORE (MACRO-FILENAME) ((PATHNAME),(BLOCKNO),(BYTENO))

       WHEN INVOKED IN THIS FASHION, SUBMIT WILL REPLACE THE
     CURRENT CONSOLE INPUT DEVICE (:CI:) BY THE FILE SPECIFIED
     BY (PATHNAME); THEN THE FILE CALLED :FX:(MACRO-FILENAME).CF
     IS DELETED.

       FURTHERMORE, IF (PATHNAME) SPECIFIES A DISK FILE, THEN
     A SEEK IS PERFORMED ON IT, AFTER IT BECOMES THE NEW :CI:
     FILE, USING THE (BLOCKNO) AND (BYTENO) PARAMETERS, WHICH
     ARE ASSUMED TO BE INTEGERS.

 2.  -SUBMIT (MACRO-FILENAME)((ARG0),(ARG1),...,(ARG9))

       WHEN INVOKED IN THIS FASHION, SUBMIT WILL CREATE A FILE
     :FX:(MACRO-FILENAME).CF BY SUBSTITUTING THE ACTUAL
     PARAMETERS ((ARG)'S) GIVEN FOR THE FORMAL PARAMETERS IN THE
     FILE SPECIFIED BY (MACRO-FILENAME). (THE K'TH FORMAL PARAMETER
     IS %K , K A DIGIT.) THE CURRENT CONSOLE INPUT DEVICE IS THEN
     TEMPORARILY REDEFINED AS :FX:(MACRO-FILENAME).CF; WHEN END OF
     FILE ON :FX:(MACRO-FILENAME).CF IS REACHED, IT IS DELETED,
     AND :CI: IS RESUMED AS BEFORE. (NOTE THAT SUBMITS CAN BE NESTED).
*/
/*$NOLIST*/
$INCLUDE (:F2:COMMON.LIT)
$INCLUDE (:F2:CHAR.LIT)
$INCLUDE (:F2:ERROR.LIT)
$INCLUDE (:F2:DEVICE.LIT)
$INCLUDE (:F2:OPEN.LIT)
$INCLUDE (:F2:SEEK.LIT)
$INCLUDE (:F2:OPEN.PEX)
$INCLUDE (:F2:READ.PEX)
$INCLUDE (:F2:WRITE.PEX)
$INCLUDE (:F2:CLOSE.PEX)
$INCLUDE (:F2:SEEK.PEX)
$INCLUDE (:F2:DELETE.PEX)
$INCLUDE (:F2:EXIT.PEX)
$INCLUDE (:F2:CONSOL.PEX)
$INCLUDE (:F2:WHOCON.PEX)
$INCLUDE (:F2:RESCAN.PEX)
$INCLUDE (:F2:SER.PEX)
$INCLUDE (:F2:PATH.PEX)
$INCLUDE (:F2:UNPATH.PEX)
$INCLUDE (:F2:DBLANK.PEX)
$INCLUDE (:F2:DLIMIT.PEX)
```

```
$INCLUDE (:F2:UCASE.PEX)
$INCLUDE (:F2:NUMOUT.PEX)
$INCLUDE (:F2:FERROR.PEX)
$INCLUDE (:F2:SCANIN.PEX)
$LIST

/*
  STRUCTURE TO STORE ACTUAL PARAMETERS AND CORRESPONDING LENGTHS.
*/
DECLARE PARAMS(10) STRUCTURE (
        DAT(31) BYTE,
        LENGTH BYTE);

DECLARE BUFFER(1024) BYTE;
DECLARE BUFFER$PTR ADDRESS;
DECLARE BUFFER$COUNT ADDRESS;
DECLARE CHAR BASED BUFFER$PTR BYTE;
DECLARE AUXPTR ADDRESS;
DECLARE PN(10) BYTE;
DECLARE I ADDRESS;
DECLARE L BYTE;
DECLARE (RESTORE,SCANNING,DEBUG,PARAM$SCAN) BOOLEAN;
DECLARE (CSD,CS,STATUS,ACTUAL,BLOCKNO,BYTENO) ADDRESS;
DECLARE (CSDNAME,CSNAME,NICKNAME,CI) (15) BYTE;

/* *-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*
   *-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*

        BEGINNING OF MAIN PROGRAM

   *-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*
   *-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-* */

/*
  INITIALIZE PARAMETER ARRAYS.
*/
DO I = 0 TO SIZE(PARAMS);
  PARAMS(0).DAT(I) = 0;
END;
CALL RESCAN(1,.STATUS);
CALL READ(1,.BUFFER,LENGTH(BUFFER),.ACTUAL,.STATUS);
BUFFER(ACTUAL) = CR;
BUFFER$PTR = .BUFFER;  /* FORCE$UPPER THAT STOPS AT PARAMS */
DO WHILE CHAR <> CR AND CHAR <> '(';
   CHAR = UPPER$CASE(CHAR);
   BUFFER$PTR = BUFFER$PTR + 1;
   END;
BUFFER$PTR = DEBLANK(.BUFFER);
/*
  SAVE NAME SUBMIT WAS INVOKED BY.
*/
DEBUG = FALSE;
IF SEQ(BUFFER$PTR,.('DEBUG '),6) THEN
DO;
  DEBUG = TRUE;
  BUFFER$PTR=DEBLANK(BUFFER$PTR+5);
END;
STATUS = PATH(BUFFER$PTR,.PN);
CALL UNPATH(.PN,.NICKNAME);
BUFFER$PTR = DEBLANK(DELIMIT(BUFFER$PTR));
/*
  TEST FOR RESTORE COMMAND.
*/
```

```
RESTORE = FALSE;
IF SEG(BUFFERSPTR,.('RESTORE '),8) THEN
DO;
  RESTORE = TRUE;
  BUFFERSPTR = DEBLANK(BUFFERSPTR+8);
END;
/*
  PARSE FILENAMES.
*/
CALL FILESERROR(PATH(BUFFERSPTR,.PN),BUFFERSPTR,TRUE);
IF PN(7) = 0 THEN
DO;
  PN(7) = 'C';
  PN(8) = 'S';
  PN(9) = 'D';
END;
CALL UNPATH(.PN,.CSDNAME);
PN(7) = 'C';
PN(8) = 'S';
PN(9) = 0;
CALL UNPATH(.PN,.CSNAME);
BUFFERSPTR = DEBLANK(DELIMIT(BUFFERSPTR));
/*
  PROCESS ACTUAL PARAMETERS.
*/
I = 0; /* PARAMETER COUNTER */
IF CHAR = '(' THEN
DO;
  BUFFERSPTR = BUFFERSPTR + 1;
  SCANNING = TRUE;
  DO WHILE SCANNING;
    AUXPTR,BUFFERSPTR = DEBLANK(BUFFERSPTR);
    IF I = LENGTH(PARAMS) THEN
      CALL FILESERROR(TOO$MANY$PARAMS,AUXPTR,TRUE);
    L = 0; /* PARAMETER LENGTH COUNTER */
    IF CHAR = '''' THEN
    DO;
      PARAMSSCAN = TRUE;
      BUFFERSPTR = BUFFERSPTR + 1;
      DO WHILE PARAMSSCAN;
        IF L = LENGTH(PARAMS.DAT) THEN
          CALL FILESERROR(ARG$TOO$LONG,AUXPTR,TRUE);
        PARAMS(I).DAT(L) = CHAR;
        IF CHAR = '''' THEN
        DO;
          PARAMSSCAN = FALSE;
          BUFFERSPTR = BUFFERSPTR + 1;
          IF CHAR = '''' THEN
          DO;
            PARAMSSCAN = TRUE;
            BUFFERSPTR = BUFFERSPTR + 1;
          END;
        END;
        ELSE BUFFERSPTR = BUFFERSPTR + 1;
        L = L + 1;
      END;
      L = L - 1;
    END;
    ELSE
    DO;
      PARAMSSCAN = TRUE;
      DO WHILE PARAMSSCAN;
        IF L = LENGTH(PARAMS.DAT) THEN
```

```
        CALL FILE$ERROR(ARG$TOO$LONG,.AUXPTR,TRUE);
      IF CHAR > ' ' AND CHAR <= 1CZ
      AND CHAR <> ','
      AND CHAR <> ')' THEN
      DO;
        PARAMS(I).DAT(L) = CHAR;
        BUFFER$PTR = BUFFER$PTR + 1;
        L = L + 1;
      END;
      ELSE PARAM$SCAN = FALSE;
    END;
  END;
  BUFFER$PTR = DEBLANK(BUFFER$PTR);
  IF CHAR = ',' THEN
  DO;
    BUFFER$PTR = BUFFER$PTR + 1;
  END;
  ELSE
  IF CHAR = ')' THEN
  DO;
    SCANNING = FALSE;
    BUFFER$PTR = DEBLANK(BUFFER$PTR+1);
  END;
  ELSE CALL FILE$ERROR(INVALID$SYNTAX,.AUXPTR,TRUE);
  PARAMS(I).LENGTH = L;
  I = I + 1;
 END;
END;
IF CHAR <> CR THEN CALL FILE$ERROR(INVALID$SYNTAX,.BUFFER$PTR,TRUE);
IF RESTORE THEN
DO;
 /*
  CHANGE CONSOLE TO PREVIOUS FILE.
 */
 CALL CONSOL(.PARAMS(0).DAT,.(':CO: '),.STATUS);
 STATUS = PATH(.PARAMS(0).DAT,.PN);
 IF PN(0) <= F5DEV THEN /* DISK FILE */  /* DO */
 DO;
   BLOCKNO = .PARAMS(1).DAT;
   BLOCKNO = SCAN$INTEGER(.BLOCKNO);
   BYTENO = .PARAMS(2).DAT;
   BYTENO  = SCAN$INTEGER(.BYTENO);
   CALL SEEK(1,SEEK$ABS,.BLOCKNO,.BYTENO,.STATUS);
 END;
 /*
   DELETE FILE WHICH JUST WAS THE CONSOLE.
 */
 CALL DELETE(.CSNAME,.STATUS);
 CALL FILE$ERROR(STATUS,.CSNAME,TRUE);
END;
ELSE
DO;
 /*
   WRITE (MACRO-FILENAME).CS.
 */
 GET$INPUT:
   PROCEDURE BYTE;
     DECLARE TEMP BYTE;

     IF BUFFER$COUNT = 0 THEN
     DO;
       BUFFER$PTR = .BUFFER;
       CALL READ(CSO, BUFFER,LENGTH(BUFFER), BUFFER$COUNT, STATUS);
```

```
    END;
    IF BUFFER$COUNT = 0 THEN RETURN 0;
    TEMP = CHAR;
    BUFFER$PTR = BUFFER$PTR + 1;
    BUFFER$COUNT = BUFFER$COUNT - 1;
    RETURN TEMP;
  END GET$INPUT;

CALL OPEN(.CSD,.CSDNAME,READ$MODE,0,.STATUS);
CALL FILE$ERROR(STATUS,.CSDNAME,TRUE);
CALL OPEN(.CS,.CSNAME, WRITE$MODE,0,.STATUS);
CALL FILE$ERROR(STATUS,.CSNAME,TRUE);
BUFFER$COUNT = 0;
DO WHILE (L:=GET$INPUT) <> 0;
  IF L = CONTROL$P THEN
  DO;
    L = GET$INPUT;
    CALL WRITE(CS,.L,1,.STATUS);
  END;
  ELSE
  IF L <> '^' THEN CALL WRITE(CS,.L,1,.STATUS);
  ELSE
  DO;
    L = GET$INPUT - '@';
    IF L > LAST(PARAMS) THEN
      CALL FILE$ERROR(BAD$PARAM,BUFFER$PTR-1,TRUE);
    CALL WRITE(CS,.PARAMS(L).DAT,PARAMS(L).LENGTH,.STATUS);
  END;
END;
/*
  ADD COMMAND TO RESTORE PRIOR CONSOLE.
*/
CALL CLOSE(CSD,.STATUS);
IF DEBUG THEN
DO;
  CALL WRITE(CS,.('DEBUG '),6,.STATUS);
END;
CALL WRITE(CS,.NICKNAME,DELIMIT(.NICKNAME)-.NICKNAME+1,.STATUS);
CALL WRITE(CS,.('RESTORE '),8,.STATUS);
CALL WRITE(CS,.CSNAME,DELIMIT(.CSNAME)-.CSNAME,.STATUS);
CALL WRITE(CS,.('('),1,.STATUS);
/*
  CI := CURRENT CONSOLE INPUT DEVICE.
*/
CALL WHOCON(1,.CI);
CALL WRITE(CS,.CI,DELIMIT(.CI)-.CI,.STATUS);
STATUS = PATH(.CI,.PN);
IF PN(0) <= F5DEV THEN  /* DO */
DO;
  PRINT$BLOCK$OR$BYTE:
    PROCEDURE (X);
      DECLARE (X,PTR) ADDRESS;
      DECLARE BUF(6) BYTE;

      BUF(5) = ' ';
      CALL WRITE(CS,.(','),1,.STATUS);
      CALL NUMOUT(X,10,' ',.BUF,5);
      PTR = DEBLANK(.BUF);
      CALL WRITE(CS,PTR,DELIMIT(PTR)-PTR,.STATUS);
    END PRINT$BLOCK$OR$BYTE;

  CALL SEEK(1,SEEK$RETURN,.BLOCKNO,.BYTENO,.STATUS);
  CALL PRINT$BLOCK$OR$BYTE(BLOCKNO);
```

```
    CALL PRINT$BLOCK$OR$BYTE(BYTEND);
  END;
  CALL WRITE(CS,.(')',CR,LF),3,.STATUS);
  CALL CLOSE(CS,.STATUS);
  CALL CONSOL(.CSNAME,.(':CO: '),.STATUS);
END;
CALL EXIT;
END;

EOF
```