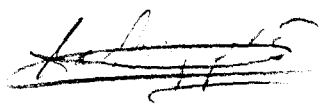
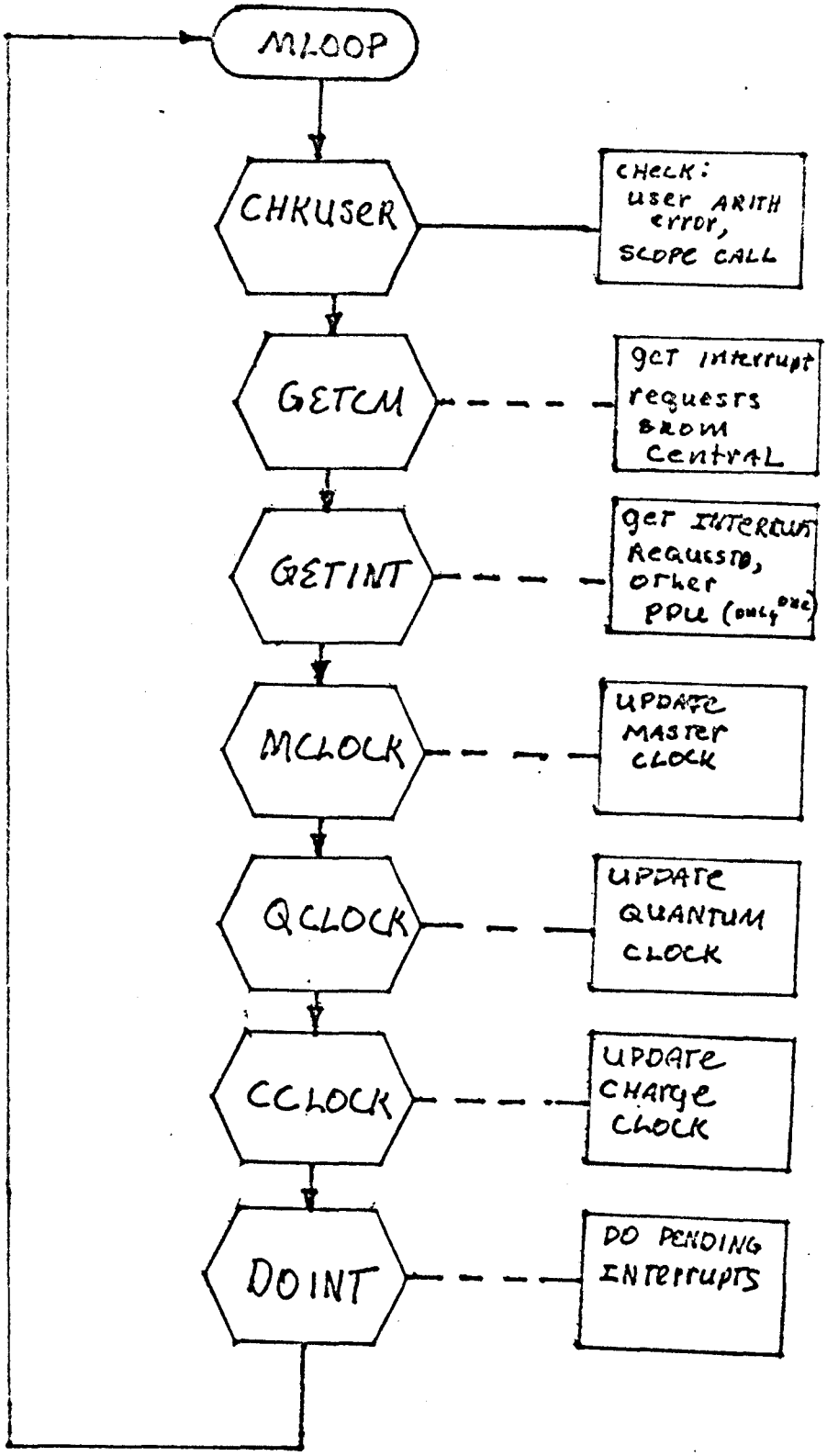
The Cal TSS Interrupt System

Gene A McDaniel

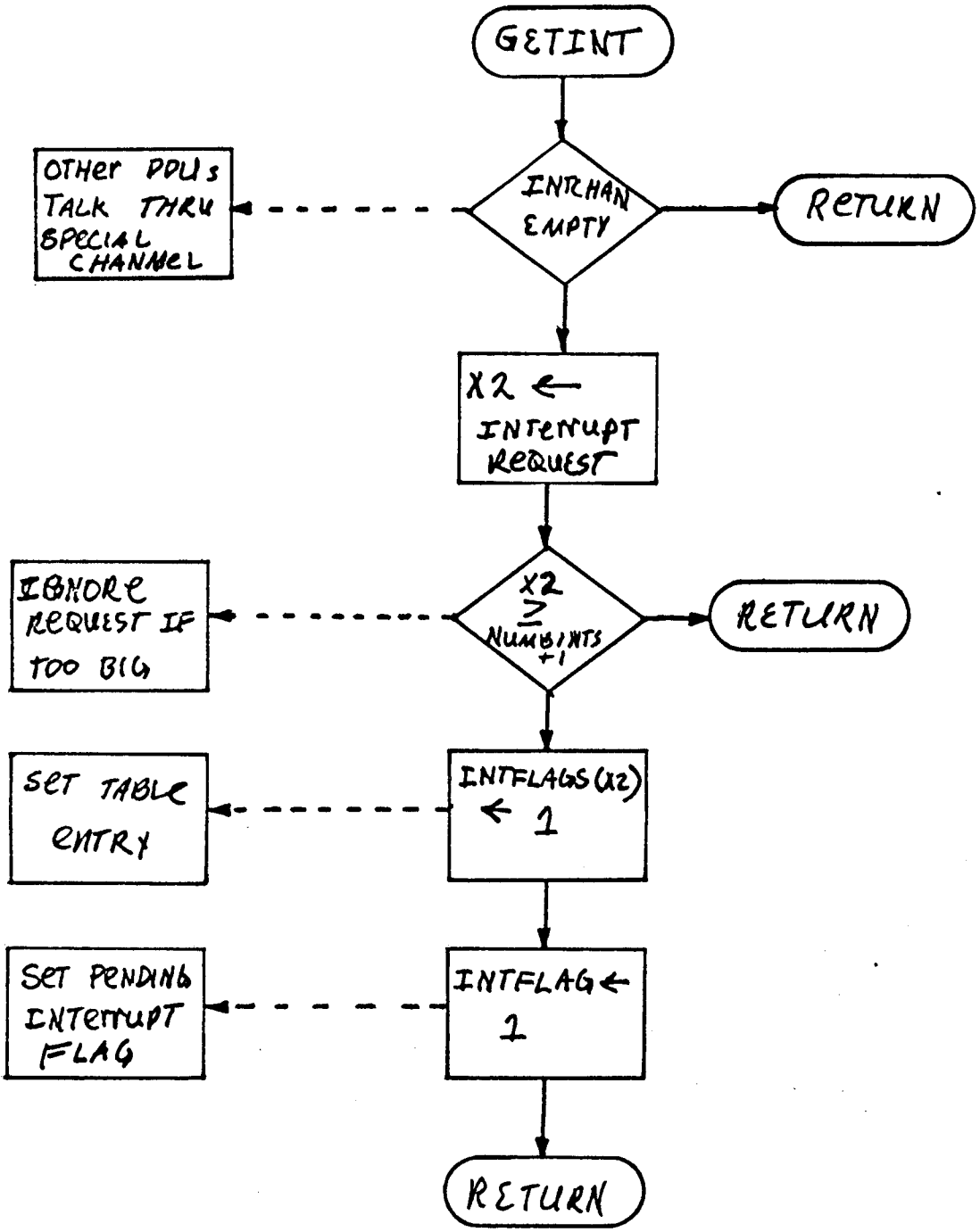


THE INTERRUPT MASTER LOOP

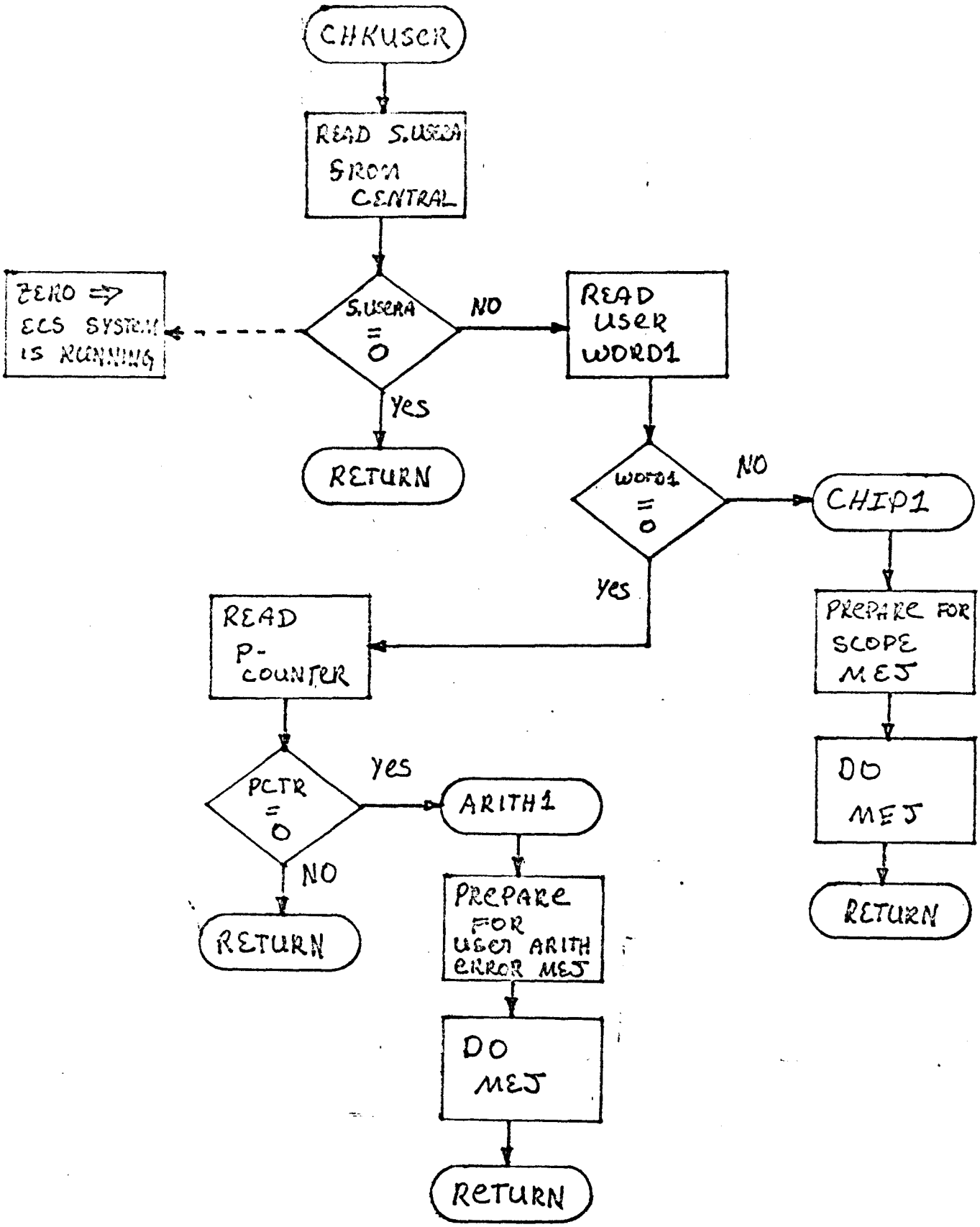


GETINT

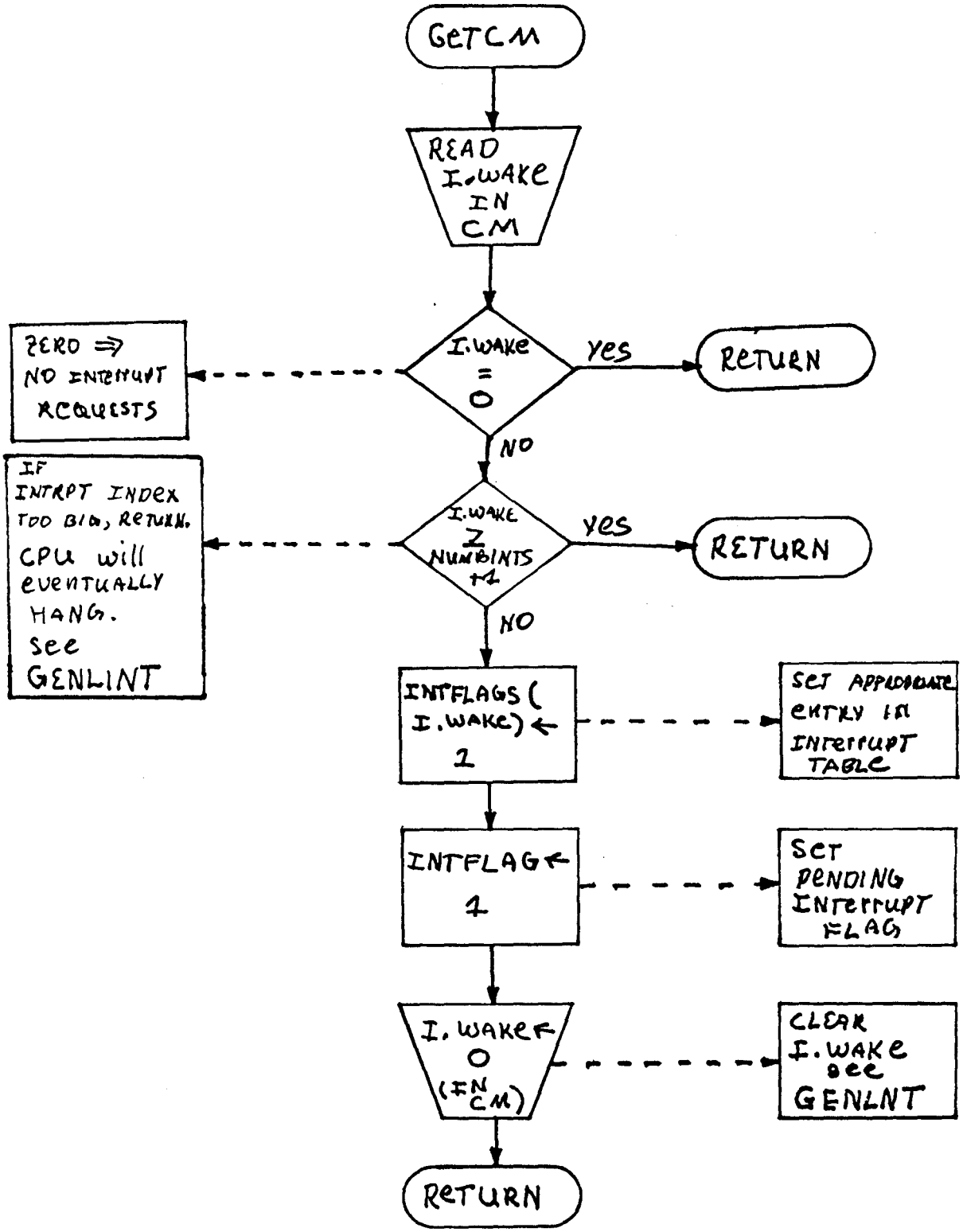
GET INTERRUPT REQUESTS FROM OTHER PPU S
(ONLY GET ONE)



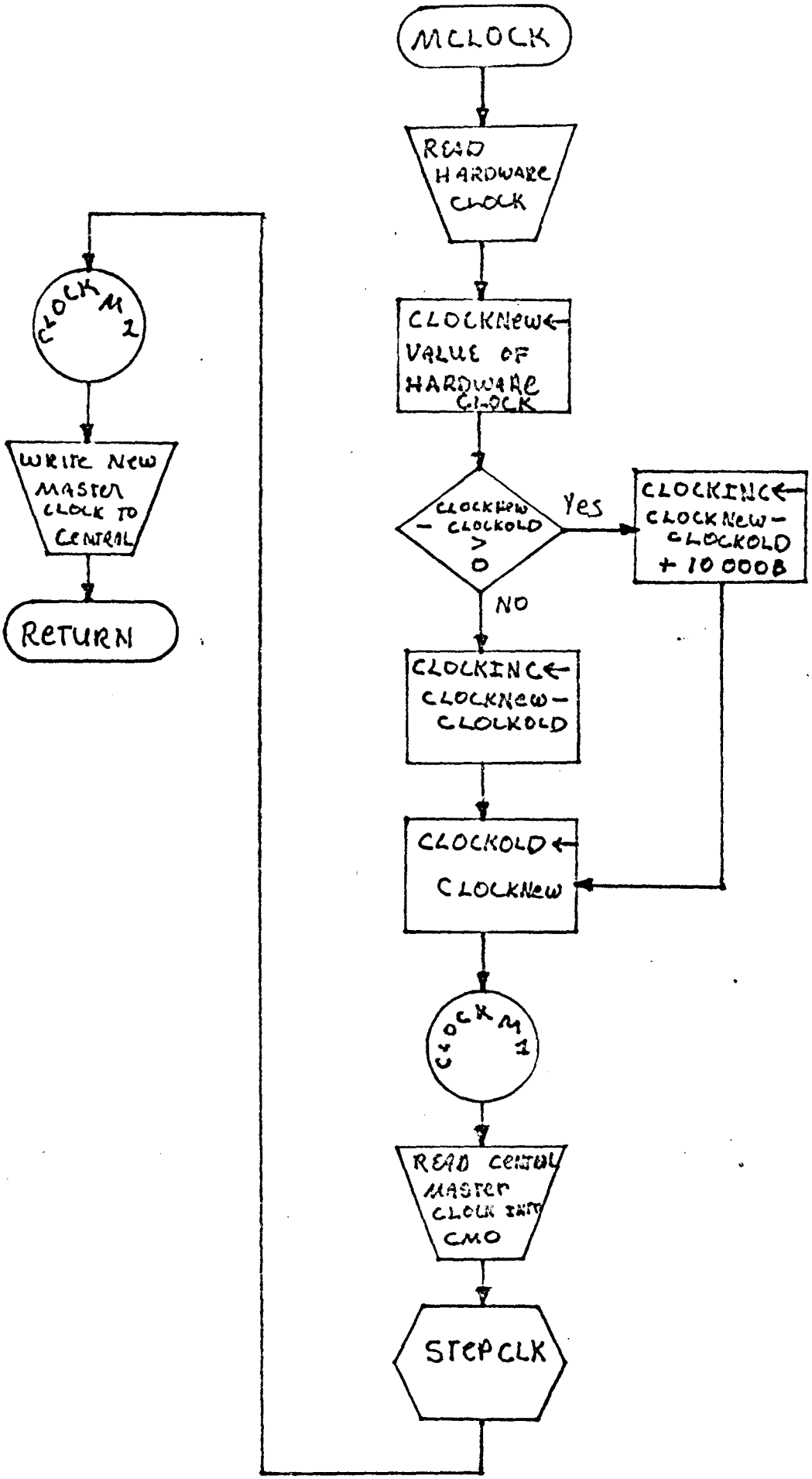
CHECK: USER ERROR, SCOPE CALL



GETCM
INTERRUPT REQUESTS



UPDATE MASTER CLOCK in CENTRAL



The CAL Timesharing system is implemented on a CDC 6400 with the following system attributes: There is 32 K of central memory (60 bit words, no parity bit) with a 1.2 micro second cycle time. Theprimary secondary storage is Extended Core Storage (ECS). ECS has a 3.2 microsecond access time and (with the UC Berkeley configuration) 1 word per 125 nanoseconds transfer rate. The CPU executes instructions in the range of .5-1.2 microseconds. There are 10 PPUs (Peripheral Processor Units) with 4K of memory ad aninstruction time of 1/ microseconds. The PPUs are asynchronous, used primarily for i/o and may communicate with each other through channels or with the CPU through central memory or aninterrupt (exchange jump). There are three flavors of interrupts to the CPU. The EXN (executed by a PPU) is unconditional and causes the CPU to exchange states with a "package" located in CM at a place specified by the PPU. There are two conditional interrupts: CEJ ad MXN, executed by the CPU and PPU respectively. If the CPU is in monitor mode, the CEJ causes the exchange jump to occur with a "package" which has been pre-determined, if the CPU is in user mode, the "x-pack" specified in the CEJ instruction is exchanged with the one currently in the CPU. MXN is a no-op if the CPU is in monitor mode. When one of the conditional interrupts occurs the CPU automatically "switches" mode. There is a 6638 disk used for large storage and a locally manufactured multiplexor which is capable of handling 256 teletypes simultaneously.

The software architecture was designed to reflect a highly modular, "layered system". The bottom most layer is the ECS system(fully operative) This should be understood in the context that it works pretty much as designed, BUT the subprocess call stack logic is being redesigned.

Redesign and coding will hopefully be completed in about six months (including some testing). The remaining layers consist of the Disk File System, Directory System, and Command Processor. These layers are in partial stages of completion. Mostly complete, I believe. The preceding is offered as a very brief introduction to system configuration. Comments about the status of the various parts of the system are purely personal opinion.

The description of the interrupt system will be presented as successive iterations in a discussion of designed structures and algorithms. Each iteration should include greater detail.

The basic interrupt system is run by a PPU which uses EXNs to force interrupts upon the system. The EXN doesn't affect the monitor flag; it is unconditional. Other PPUs "hang" on a channel, waiting for the MPPU to take their interrupt requests. The MPPU checks the interrupt channel (INTCHAN) and a CM cell (I.Wake) for interrupt requests. Interrupts are locked out by CPU routines which set a cell (I.lock, accessible only when the ECS, Cayer "~~system~~" is running). Interrupts automatically set another cell, I.Wait to the value of I.lock, so that if interrupts are locked out, routines which tend to keep I.LOCK set for a long time will pause with interrupts enabled.

User code and much system code executes on the CAL TSS system in the context of what are known as processes. Each process has several subprocess. All code actually executes as part of a subprocess (other than ECS system code or interrupt code). Among other things, subprocesses implement basc protection mechanisms in TSS. We shall now leave the theological discussion of processes and delve into the question of interrupts and their place in the system: Clearly for I/O and certain control functions. I/O implies perfoming tasks for user processes and this introduces the question of interprocess communication.

At the level of the ECS system and above there are two basic mechanisms for interprocess communication; software interrupts ad event channels. Software interrupts force the calling of a specific subprocess in an interrupted

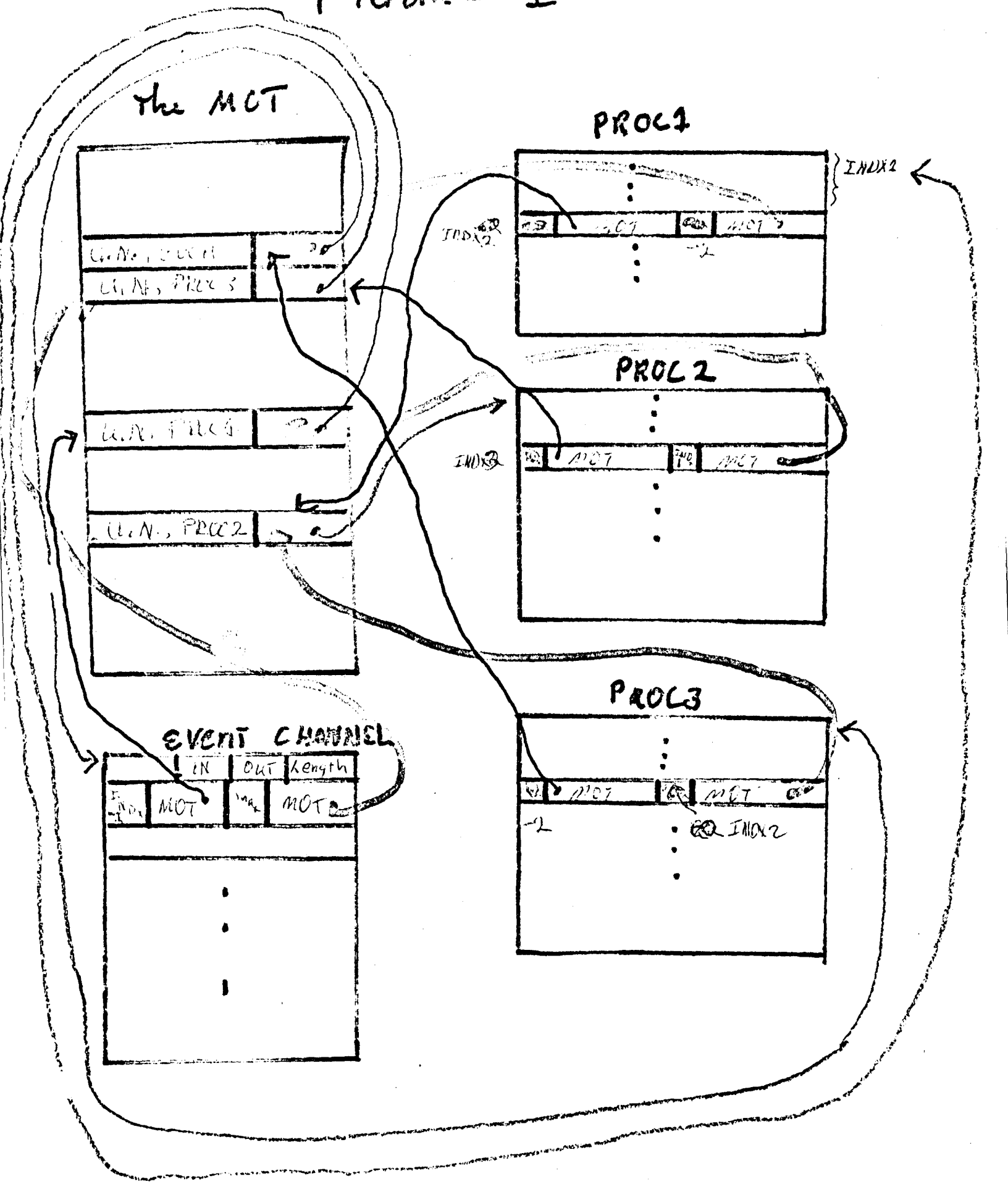
process. This is contingent upon an inherent priority of the "interrupt subprocess". This is in turn determined by the static subprocess tree and currently executing subprocess. In any event, when such an interrupt strikes, the interrupt subprocess is entered at location equal to the normal entry point minus two (wds of CM) and an 18 bit datum is mdde available to the interruptsubprocess. Event channels serve to synchronize processes. They act as a test and set mchanism: a process can get an event (from one or more event channels) or "hang or F-return" depending upon how he called the system. These are atomic, noninterruptable actions. A 60 bit datum and the sender's MOT (Master Object Table, discussed shortly) are provided to the receiver of the event. If a process hangs on an event channel, a doubly linked list connects the process to the event channel and/or other processes which are hanging upon the event channel. The MOT is a table in low ECS which provides the unique name and abs. ECS address of every object which is current in the ECS system. (See diagram 1).

As could be easily surmised, the interrupt system-ECS system communication occurs primarily throuh the use of event channels and specially created interrupt objects. The interrupt logic uses these special objects called pseudo-processes (henceforth, pprocs) to perform three fu nctions:

1) Synchonize i/o
2) Communicate with user processes
3) Systematize and simplify cooperation between cpu interrupt code and ppu interrupt code.

There is a point during system initialization when ECS is compacted and interrupt objects (or devices) are then created. This is done so that absolute ECS addresses may be used without fear of invaldation after future Ecs garbage collections-compactions. These are three kinds of objects,

FIGURE 1

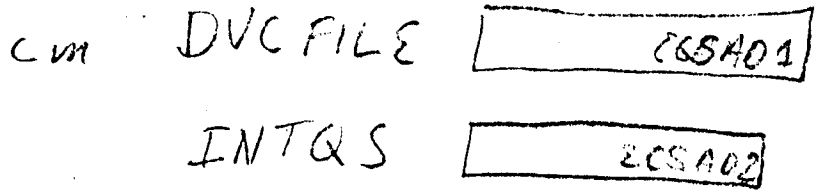

The MCT
PROC1
PROC2
PROC3
EVENT CHANNEL
IN
OUT
length
MOT
INDX2

created in groups called classes, during system initialization. These classes represent the objects manufactured for each of the different interrupts that a PPU may want to send to the CPU. Within each class there may be many "devices" as is the case with the objects created for the multiplexor interrupts (requested by MUX ppu). The three objects are files, event channels and pseudoprocesses (pprocs).

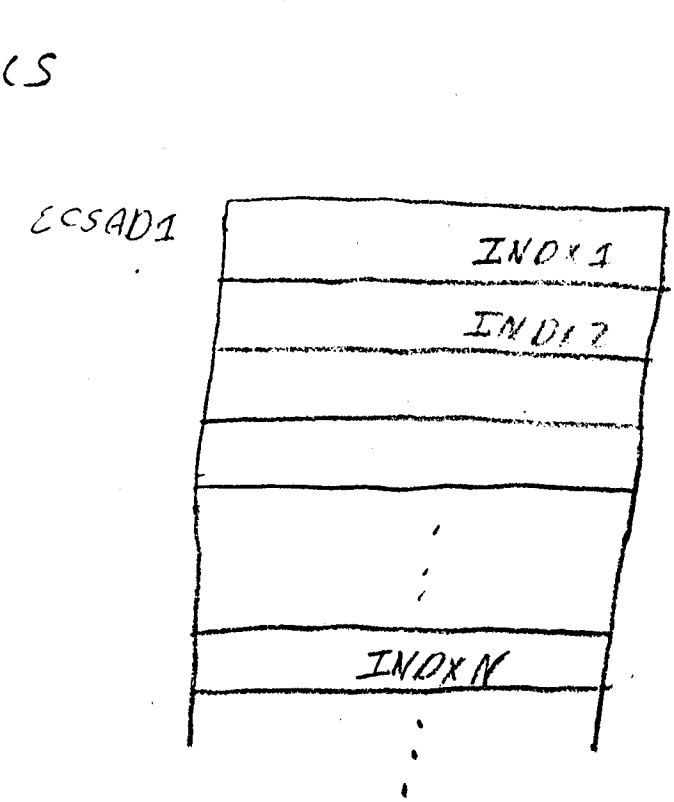
Files are used for two main purposes: buffers or pointers. Files as buffers contain I/O data going to and from the CPU. Files as pointers tend to contain pointers to objects or pointers to queues of objects. For example, there is a cell in CM called DVCFILE. DVCFILE contains the abs. ECS addr of a file which contains one entry per interrupt type (=interrupt index). That entry is a pointer into the master clist of the first pproc for that class. (Presumably that entry in the master clist is followe d by more pprocs of the same class or by other objects, created in aknown order by the system initialization code. Another CM cell, INTQS contains the abs ECS address of a file which contains a "first" and "last" pointers for the interrupt pproc list for each type of interrupt. At this point honesty forces the admission that not all interrupt routines make use of all the facilities about to be described. Some interrupts (like those from the MUX) make use of all data structures described here. Others like DSKINT only make use of the pprocs, file buffers and event channels created at initialization.

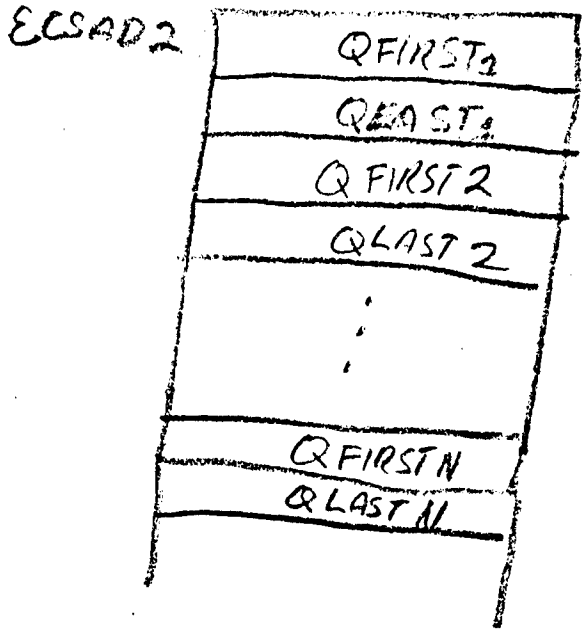
Event channels are created for data storage and as a mechanism for hanging user processes while they are waiting for I/O (data storage in the sense that they are able to communicate requests foraction to the interrupt system). These requests foraction go to pprocs which are hung on the event channels.

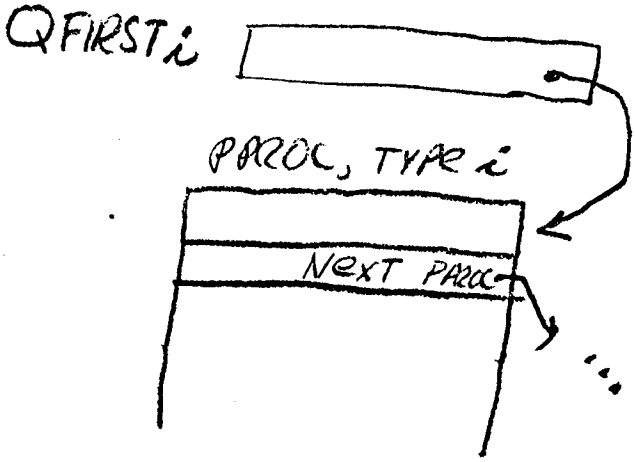
FIGURE 2

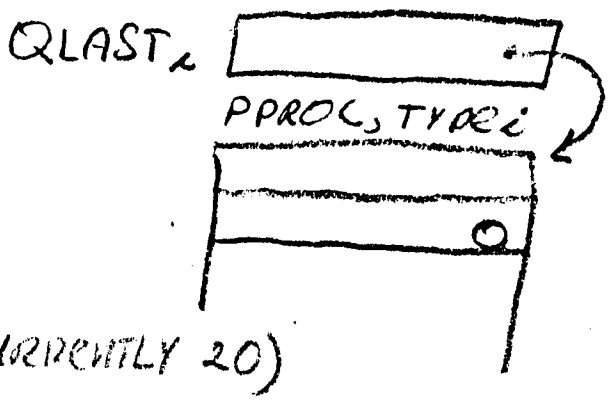


ECS









Indxi = index in MASTER CLIST FOR FIRST PPROC of INTERRUPT TYPE i

n = MAX # of INTERRUPT TYPES = MAX INDEX in PPU INTERRUPT TABLE (CURRENTLY 20)

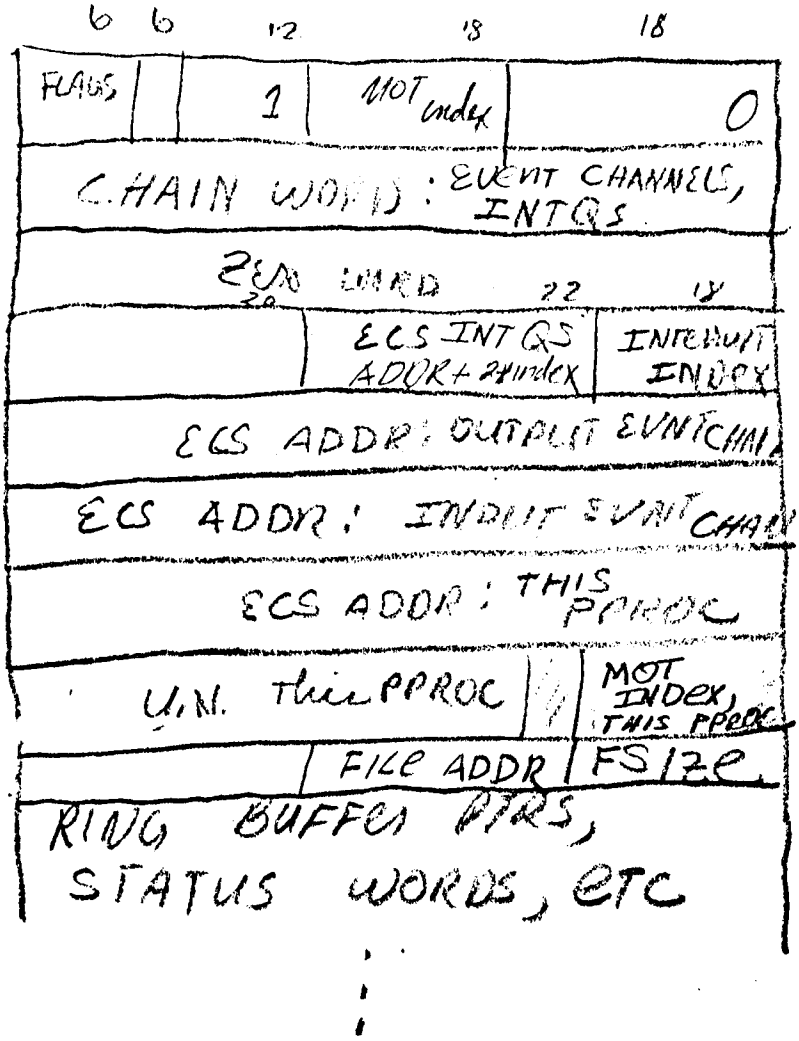
The event channel code recognizes when a pproc has received an event and transfers control (temporarily) to appropriate interrupt subroutines (in GENLINT, discussed later). The result is that the pproc is unchained from the event channel process queue and the MPPU is notified that the CPU wants an interrupt. Then control is relinquished to the "normal" event channel code.

Pseudo-processes are very pseudo. In fact they aren't processes at all. A pproc is a collection of contiguous words in ECS which contains sundry interrupt system data and three words which are a concession to the interrupt system code. The first word is a header word which looks like the first word of a normal process, except the flag bits indicate that it is a pseudo process. The second word is a queue word which the event channel code uses to chain the pproc to the event channel (the event channel code is passed the appropriate queue word offset (index)=1). Third word is zero. There is also a great deal of built in information in the pprocs for each interrupt index. Typically a single pproc will "belong" to a single teletype or to an external device like the printer or a tape driver. During initialization the file location and size for the particular device, ring buffer pointers, interrupt index and room for two event words are built into the pproc. MUX pprocs, for example are full of pointers used by the MUXINT routine--in part as a check against pointers maintained by the user.

There are, of course, "non-object" data structures which are fairly common across interrupt code: CM buffers for immediate PPU use, CM buffers and words which record current action requests and responses on the part of

FIGURE 3

A TYPICAL PSEUDO PROCESS



FIRST WORD, LOOKS LIKE P.ROMAD FOR EVENT CHANNEL CODE. FLAGS ARE SET TO R̲U̲N̲N̲I̲N̲G and E̲ - pseudo process

the relevant PPU, the interrupt code or both. (Ring buffers are used in MUXINT in such a way that while an interrupt is running, a "dialogue" between the CPU and the PPU may take place, and several actions may be serviced (this may occur inthe TOCMQ and FRMCMQ ring buffers. MUXINT requires that the aforesaid ring buffers be located in the bottom 4K (dec) ofCM--due to the addressing used by the PPU code.)

INTINIT is the routine which is responsible for initializing interrupt objects. There are three entry points, INTINIC, INTINIA, INTINIB, which are called in that order.

INTINIC, called from INITL before the master clist for the system is created, computes the master clist needs for the interrupt objects and places that sum in the cell, MC.INT. It also initializes a pointer word, M.ADDR which is used later in initialization to decrease option bits on the capabilities for interrupt objects.

INTINIA, called from INITL after the masterclist and before the ecs system operations have been created, creates the file to hold device C list indeces and puts the beginning address (in ECS) of that file in DVCFILE.

INTINIB, created after the ecs system operations have been created and immediately following a call to the compactor (to guarantee the ECS addresses will always be good), creates the interrupt objects for the different classes of interrupts. First the interrupt queue file is created and its ecs address is placed in INTQS. Then the MUX, S-device,Disk, Display Driver, and clocked event channel interrupt objects are created. Each of the peceding represents an interrupt class and a single, valid interrupt index to the MPPU.

Each such subroutine (MMUX, MSDVC, etc) makes an single call on the subroutine, NEWCLASS, and then creates the objects necessary forits internal design. This occurs in roughly the following way: For each class of interrupts, a prototype, or template, pproc has been assembled into the INTINIT code. NEWCLASS initializes some values into those templates each time it is called. The subroutines actually making the "real" interrupt objects then add whatever information NEWCLASS didn't know. After the pproc has been created, the template is written into ECS ontop of the pproc. In that way the pproc is "wired" with information peculiar to the type of interrupt it represents. MMUX initialization shall now be followed in somewhat greater detail.

MMUX passes five parameters to NEWCLASS: 1) the cm address of the pproc template queue word (see diag 3), 2) the location of two cells to contain pointer information (MUXPNTS and MUXQADD in MUXINT), 3) the interrupt index (=MUXINTX=1), and 5) the number of devices in the class (MUXWRBIT+1=257). NEWCLASS saves its parameters and then begins its task at initialization. First it checks if if the interrupt index is valid (error if not). Then it creates a file of size=number of devices in the INTERRUPT CLASS. The ECS address of this file is then placed in the first pointer (MUXPNTS) this file is used to contain the ECS addresses of each of the pprocs. Then the current position in INTQS is cmmputed and that number is placed in the second pointer word (MUXQADD); it is also placed in the queue word of the current pproc. Then the first clist index for objects of this class is placed in the current index in DVCFILE. NEWCLASS RETURNs to MMUX.

MMUX now stores the current clist index in a cell in IPROC (JPROC). This is the special tty. A file of size ttyfsize is created by calling the subroutine MFILE.* The abs ECS address of the file data block (returned in x0 by MFILE) is merged with the contents of the current filewd (see fig.3) for the pproc template. A standard tty template file block is written into ECS where the current file data block is located. An output event channel of appropriate size (ECOUTCNT) is created, all but some option bits are turned off (OB.SENDEV+OB.CHNAM) and the ECS address of the event channel is writen into the outchn word of the pproc template. The same thing is done for the teletype input event channel (ECINCNT,INCHN, and OB.GETEV+OB.CHNAM). A pproc of size PPROCSZ is created, the MOT unique name and MOT index are written into the template and then the entire template is written into ECS at the beginning of the pproc. Then the ECS address of the pproc is written into the current index in the file pointed to by MUXPNTS. This process is repeated for as many times as there are teletypes "turned on" currently equal 24 (S.NUMTT in SYSPAR).

The S (simple) Device interrupt objects are basically created in the same way, but with different parameters. The Disk interrupt objectsdiffer considerably in their use: the pproc pointer file (corresponding to the file pointed to by MUXPNTS) is not used and a considerable amount of buffer space is created. Display interrupt objects are likewise pretty similar except there is also a file created for event channel pointers (DSPKPNT) as well as pproc MOT entries (DSPPPNP). The Event channel clocks are a red herring in this respect : no pprocs are created for them.

* The capabilities for all such files are "overlayed" in the MASTER CLIST. Since the addresses are known, there need be no entry in the CLIST — AND NO WAY A BUG CAN CAUSE the FILE TO BE DESTROYED!

Figure 4

TABLE DESCRIPTION OF VARIOUS ASPECTS of PPROCS FOR * EACH INTERRUPT TYPE

	MUX	S-device	DISK	Display	CLOCK EVENTS
QUEUE WD IN PROC TEMPLATE	PQUEUE	SPQUEUE	DPQUEUE	DSPQUEUE	DSPQUEUE **
PPROC PTR FILE CELL	MUXPNTS (MUXINT)	SDVCPNT (SDVCINT)	DSKPNT (DSKINT)	DSPPNT (DSPINT)	DSPCLK (DSPINT)
INTERRUPT QUEUE CELL	MUXGADD (MUXINT)	SDVCQPT (SDVCINT)	DS_{KINTQ} (DSKINT)	DSPQPNT (DSPINT)	INTSCD ** (CLKINT)
INTERRUPT INDEX	MUXINTX (= 1)	SDVCINTX (= 2)	DSKINTX (= 3)	DSPINTX (=4)	DSPINTX+1 (= 5)
# DEVICES IN CLASS	MUXWRBIT +1 (= 257)	~~CONSOLEX~~ SDVCCNT (=3)	9	DSPLRN (DSPINT) (=8)	DSPCLKN (DSPINT) (=6)
PPROC SIZE	PPROCSZ 17	SPPROCZ 19	$DPPROC_Z$ 8	DSPPROCZ 8	DSPINT =6
IN-EACH SIZE	3/PPROC	6,4,4	15 +65	4 / PPROC	2
OUT EACH SIZE	10 PY PPROC	6,4,4	+?	12 / PPROC	
FILE SIZE	17 PY PPROC	2058, 512, 512	832 +?	16+2048	X

* INTERRUPT TYPE = INDEX

** DUMMY

Now its time for the MPPU (or MPP, as the source is titled). MPP stays in a tight loop, worrying about user errors, cm interrupt requests, clock updating, and sending PPU originated interrupts. The USERRA routine checks if the current value of the p-counter is zero. if this is the case, S.USERRA is checked, if it is zero, the subroutine does a return. Otherwise appropriate action is taken, whether the problem is user arith. error or a scope call. GETCM reads I.Wake to obtain an interrupt index. If the value is bigger than the table, the subroutine merely returns (note that this hangs up Central as soon as a CM request for another interrupt strikes-- code will wait forever for MPP to clear I.Wake.) If the index value is ok, the'index'th entry in the interrupt table in the MPPU mem ory is set to one. GETINT examines the interrupt channel (INTCHAN) if it is empty, GETINT returns, otherwise, GETInt sets the 'index'th entry in the interrupt table to one and returns (if index is greater than table size, index is ignored). The code then updates the master clock in the following fashion:

1) read the hardware clock
2) save the new value
3) compute the elapsed time from the oldvalue stored inPPU memory.
4) read the CM master clock
5) updat the master clock
6) return

The Q Clock is examined. If it is positive, the user has exceeded his quantum, and the system tries an MXN to swapout the user. Otherwise, it computes a new value for the q-clock and writes it into CM. If Q-clock goes greater than zero, code tries to do an MXN and then returns.

The value of the charge clock is read fom CM, its value is updated and than rewritten. The Nitty Grity code now executes.

If INTFLAG (set each timean interrupt was found) is zero, return to the main loop. Otherwise the codeloops, looking through the INttable for an interrupt (ie., cell not equal 0). When an interrupt is found, CM is read, 'index' words more than I.points in LOWCM. I.Points is a table of P-counters in CM for use in the interrupt X pack--for the new p-counter. Now the MPPU reads the user P-counter. If an ECS transfer is occuring the P-counter will be negative, and the MPPU will loop, waiting for it to finish. The PCTR is written into I.Box (the interrupt XPACK). and the MPP performsan EXN. The currentclock value is read ad abs cm 2 is read and stored (all this to allow the interrupt code to have time to find out that it can't (or can) strike at this time.

If the P-counter=0 then the interrupt failed, and MPP does an EXN to restore the CPU and then returns to the master loop. If the P-counter is not equal to zero, the interrupt held. The interrupt table entry (at index) is zeroed and the old clock value is incremented. We now enter a loop inwh ich

1) Getint is called if (INTCHAN) is empty, otherwise we loop by looking at the master clockand updating it, then beginning loop again (check for PPU interrupts, CPU interrupts, and updating the master clock.

If the interrupt has finished, we

1) EXN to restore the state of the cpu
2) Restore the system state word (=abs cm 2) [used by display drives]
3) Return to master loop.

One can envisign the CPU interrupt logic as performing in one of four modes: It is acting upon a PPU request of some sort; it is acting upon a user initiated request of some sort; acting upon a backlog of both; or "conversing" with the PPUs against a background of one of theaformentioned three. The last possibility was singled out because, 1) it is possible with the interrupt system as it now stands, and 2) it could result in one type of interrupt running for a disastrously longtime (thereby preventing a speed daemon--hypothetical, construct of the TSS system--or disk interrupt from running). That seems rather remote, and the idea of letting interrupts which have bunched together run at one time seems attracive if it doesn't get too extravagent. The drift of the design "worry" has been to the opposite direction: the CPU locking out the PPU interrupts for to long. Nothing now, and not too much in thenear future is planned which will want to change this. The description of the I.LOCK, I.WAIT, and I.WAKE flags has been presented, but no clear motivation for I.LOCK has been offered, tho I suppose that must be fairly evident. The interrupt system sends and gets events. The event mechanism is used by the system to block and unblock processes. Portions of that system are bending pointers which would result in a complete disastr if an interrupt could suddenly strike and cause a process to get scheduled, or to have its queuing word pointers manipulated. Hence, I.Lock.

It is fairly clear that not all interrupts may have the potential to cause this damage. In particular, there is a class of display driver interrupts which are allowed to strike without checking the value of I.lock. This is done in the firm belief that the only thing such interrupts will do is read (not even write) ECS and set operator time and date for the screen. The particular algorithm followed by the display driver to

insure that this is the case is described later. A description of the three general interrupt routines now follows.

GENLINT acts as the interrupt system's interface with the rest of the world. It consists ofthree subroutines: Hang1, which hangs a pproc on an event channel; Unhung1, which adds a pproc to appropriate interrupt queue and notifies the MPPU that CENTRAL needs an interrupt; and Dintq, which removes a pproc from an interrupt queue,doing all the right pointer bending in the process.

HANG1s first action is to serve as a sort of register interface with the event channel code. It is entered almost ready to call the event channel code. Its return link is saved (b7 is set to a new value 'inside' HANG1), the ECS address of the pproc is saved at HANG1.B, b1 is set to INTSCR (GENLINT) and transfers control to HANG in the event channel code. If an event was already waiting for the pproc when the event code tried to hang it on the event channel, the pproc was 'unhung' again, x6 and x7 were set to the event and x2 was set greater or equal to zero (otherwise there was no event and HANG1 can return normally to the location it stored at HANG1.A). In the event of an event, the ECS address of the pproc is refetched from HANG1.B and control passes to UNHUNG1.

UNHUNG1 is entered after a pproc has received an event. The pproc has been unchained fom the event channel process queue; x6 and x7 contain the event sent to the pproc. x1 contains the ECS addr of the pproc ad b1 is the pointer to a suitable scratch area for the interrupt system. First the pproc read into CM from ECS, then the event is stored into the CM version of the pproc. I.WAKE is examined. If nonzero, UNHUNG1 loops until the MMPU clears the cell; when I.WAKE is zero, the interrupt index for this pproc is stored into I.WAKE. This signals the MPPU THAT AN INTERRUPT of the SAME index type as THAT of the pproc is being requested by the CPU interrupt code. Now the appropriate INTQS entry must be updated (see figure 2). The two words

of the INTQS entry must be read from ECS. The current value for QLAST is saved, and QLAST is reset to the ECS address of the pproc. If the old value of QLAST was zero, then the queue was empty and control passes directly to UNHUNG3. Otherwise the foreward pointer of the pproc named by the old value of QLAST must be reset to point to the new pproc. Control then passes to UNHUNG2.

UNGUNG3 was entered because the interrupt queue had been emptry. QFIRST is now set to the ecs address of the pproc (QLAST HAs already been set) and control passes to UNHUNG2. UNHUNG2 zeroes the chainwd of the pproc (it is the end of the pproc queue) and the pproc is then written into ECS. QFIRT andQLAST are written into ECS and UNHUNG now returns.

DINTQ is the interrupt subroutine which removes a pseudoprocess from an interrupt queue. DINTQ is entered with x1=ECS address of the interrupt queue and it exits with x2=the ECS address of the pproc or zero (if the interrupt queue was empty). The two interrupt queue entries are read into CM at locations QFIRSTX and QLASTX. Fetch the current value of QFIRSTX into x2. Return to user if QFIRSTX=0 (queue is empty). Otherwise,read the pproc named by QFIRSTX from ECS. Set QFIRSTX to the chainwd in the pproc. If the chainwd=0, reset QLASTX to 0 as well (queue is now empty). Write the CM versions of QFIRSTX and QLASTX into the interrupt queue in ECS. Set the chainwd in the pproc to 0 and procede to write the pproc back into ECS. Return to the user (X2 is set to value of ECS address of pproc.

As a matter of course, the event channel code in the ECS system recognizes the difference between a process and a pproc. When an event arrives at an empty channel, the event channel process queue is checked to see if anyone is hung on the event channel. If so, the process (pproc) is unhung from the event channel. At this point the code notices the

"e" flag is set on its process--which means that the"process" is really a pproc. b7 was set to the appropriate point in the event channel code (return link) and control passes directly to UNHUNG1 which places the pproc on the proper interrupt queue, notifies the MPPU that Central wants an interrupt, places the two event words directly in the pproc and then returns to the event channel code. The event channel code clears I.LOCK (which is always set when the event channel routines are running) and returns to the sender of the event (an interrupt routine could have sent the event--entry point is EVENT1, several registers must be set, including b7=return), and the MPPU will attempt to interrupt the CPU fairly soon there after. If an interrupt routine wants to hang a pproc on an event channel, it enters the event channel routine HANG (with appropriate registers set) which is happy to hang process or pproc on a specified event channel. This complete the description of the core interrupt routines. This mechanism allows the individual interrupt taks to be completed with a reasonably clean interface with the ECS system and to take advantage of parts of the ECS system's services--interprocess communication and syschronization through event channels.

Two brief interrupt scenarios would be as follows: 1) The user sends an event to an event channel on which a pproc is hung. The event channel routines recognize the pproc and pass control to UNHUNG1 UHNUNG1 queues the pproc with its new event and notifies the MPPU t at an interrupt is desired. It then returns to the event channel code which will begin to return to the user. At the same time, the MPPU eventually notices that I.WAKE is set and notes the index of the interrup being requested. After all other parts of the MPPU loop have been completed, the MPPU tries servicing interrupt requests, beginning with the smallest interrupt index. Eventually an interrupt from the MPPU strikes the CPU. 2) (the P-counter for the interrupt from theMPPU was set by the MPPU)

The interrupt is running. It has a choice of checking its local storage to see if there are additional "instructions" in the form of prearranged commands which where put there by the PPU "overseeing" the real device that this interrupt services, or it can call DINTQ to remove a pproc from its interrupt (index) queue. At this point the interrupt routine ha s in its hands the pproc which received an event (or which was, perhaps, just hung on the interrupt queue for some other reason by a different piece of interrupt code for that level of interrupt) and whatever information there may be in its CM buffers, and it can presumably begin to do whatever it was supposed to do. When the interrupt routine is finished it jumps to cell 0 (absolute) and the MPPU eventually returns the CPU to t e former state.

As a whole the interrupt system seems to be a reasonably clean and very general interface with the ECS system. It allows for multi-level interrupts (but interrupts may not be interrupted), and has the virtue of "cheating" very few of the rules made by the ECS system--for example, the ECS system tends to abhor absolute ECS addresses unless they have just come from the MOT. The interrupt routine use the absolute addresses of the file data blocks (and of the pprocs and event channels),but ECS was compacted before those objects were created,so they should never, ever move.

Adding another device to the system would be trivial from the point of view of the interrupt system interface. Another subroutine would have to be added to INTINIT. This subroutine would make a call on the subroutine NEWCLASS to initialize some of its pproc template values and to initialize pointers relevant to the interrupt type that it represents.. Then the

new subroutine would procede to make the pprocs, buffers, and event channels requisite to its needs. Presumably another PPU could be fired up with the appropriate code to interface with the new device. The MPPU doesn't need to know anything at all about the new interrupt. The interrupt index represents a cell in the interrupt table and an offset from a point in CM where the P-counter value can be found.

BIBLIOGRAPHY

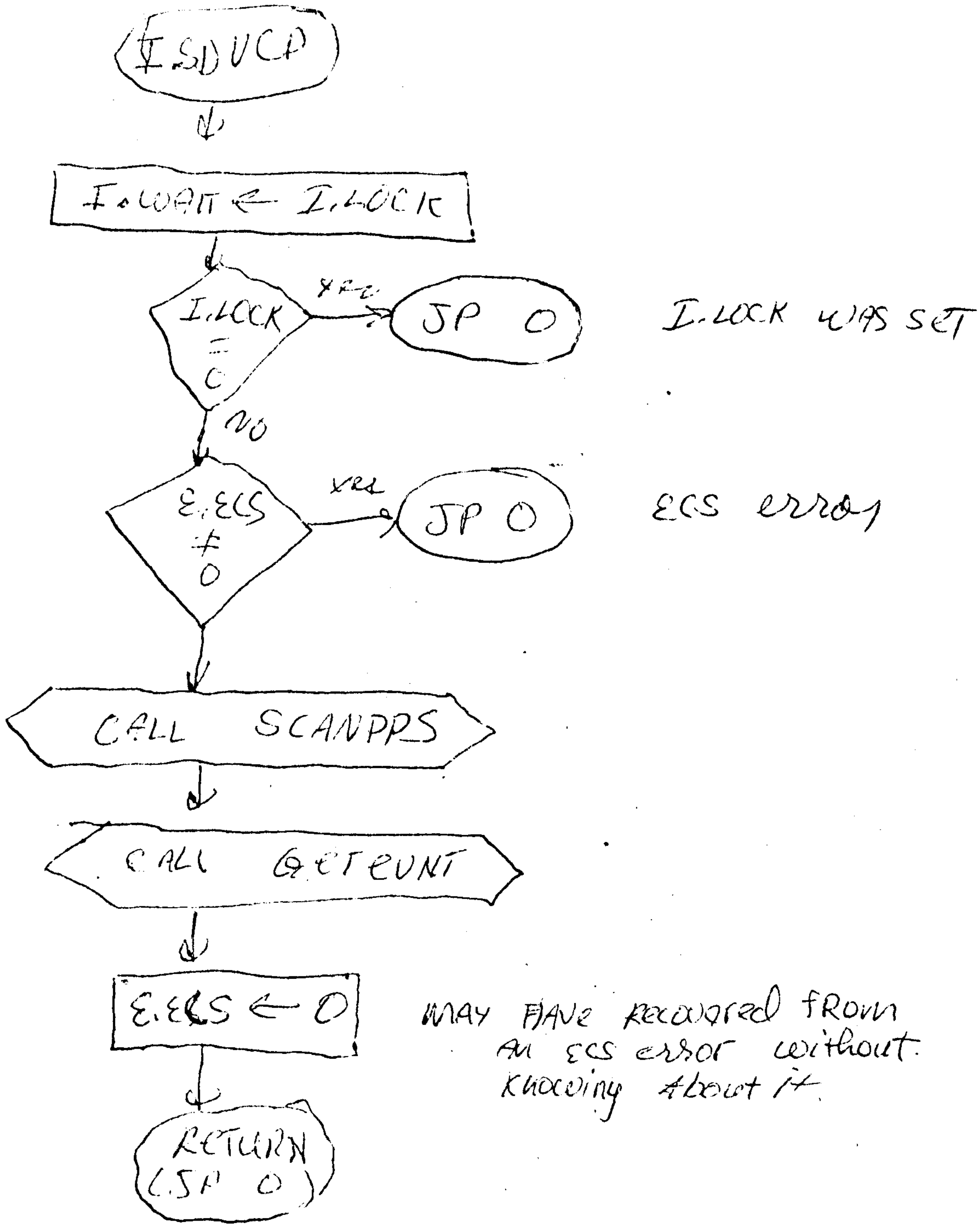
~~BIOGRAPHY~~

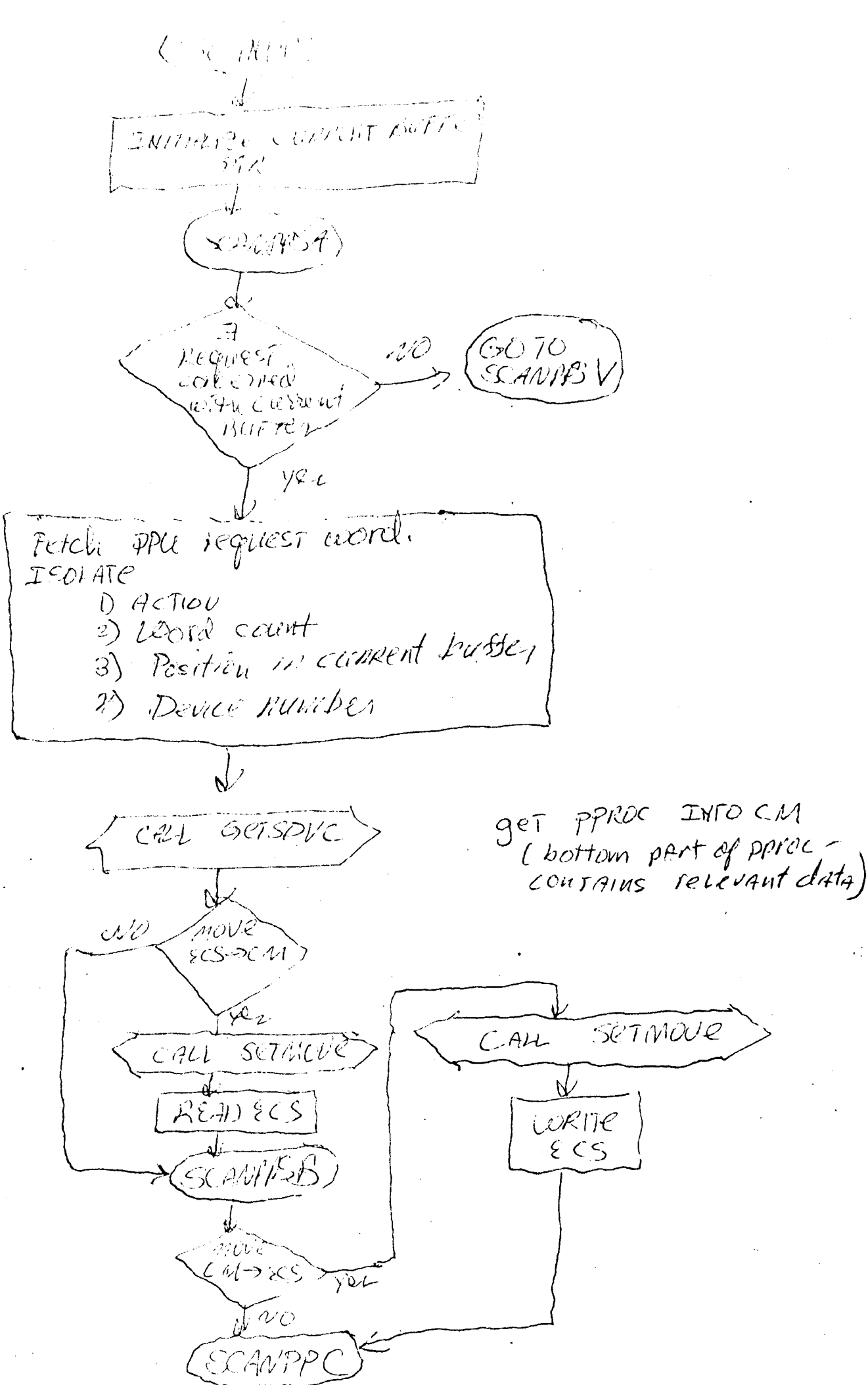
Anyone wishing to pursue this issue any further may check the following (very often undocumented, I'm afraid) computer programs.

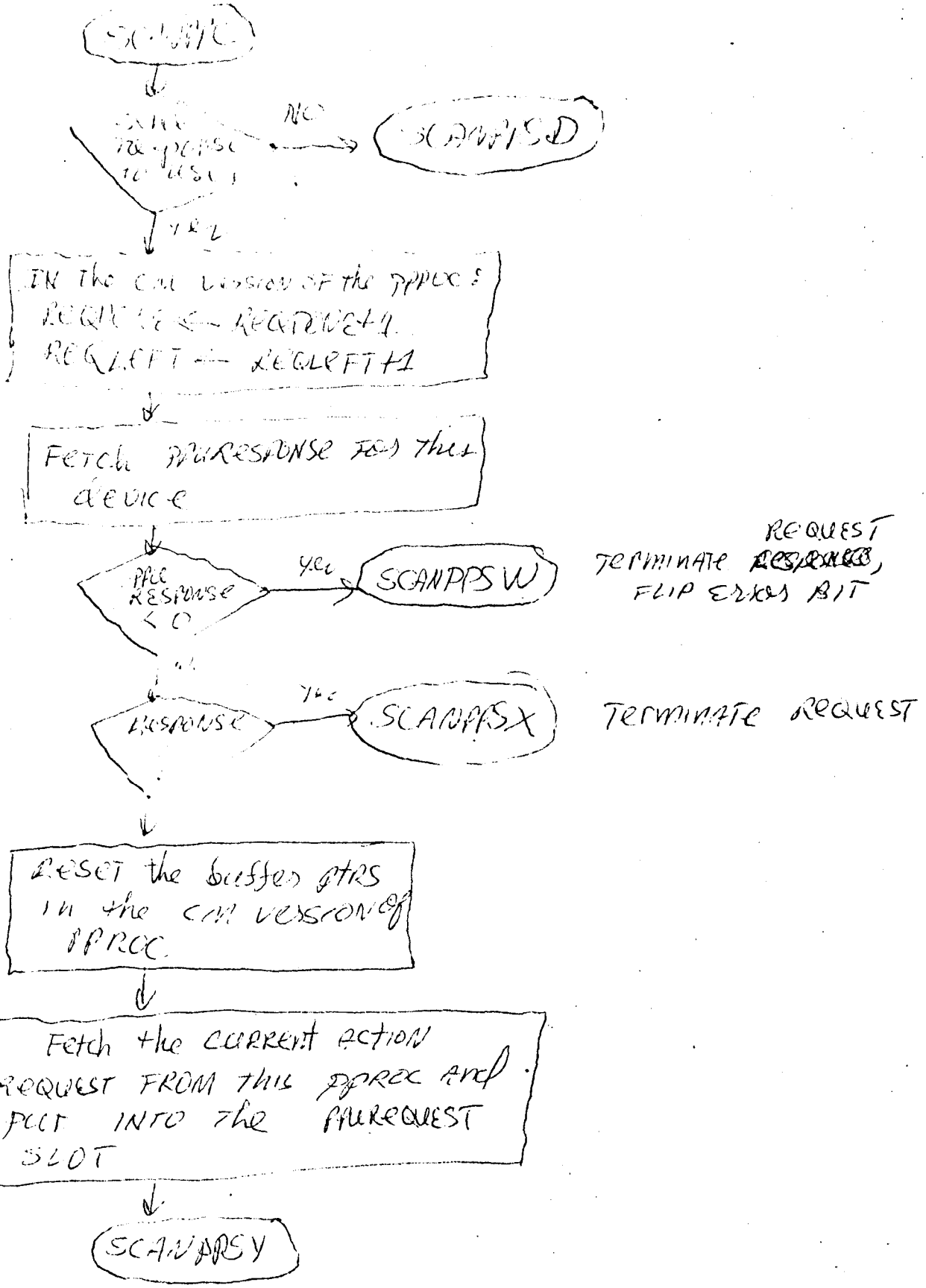
DSPINT,SOURCE
EVENT,SOURCE
GENLINT,SOURCE
INI,SOURCE
INITL,SOURCE
INTINIT,SOURCE
LOWCM,SOURCE
MSM,SOURCE
MPP,SOURCE
MUX,SOURCE
MUXINT,SOURCE
SDVCINT,SOURCE
SYSPAR
SYSYMB,SOURCE

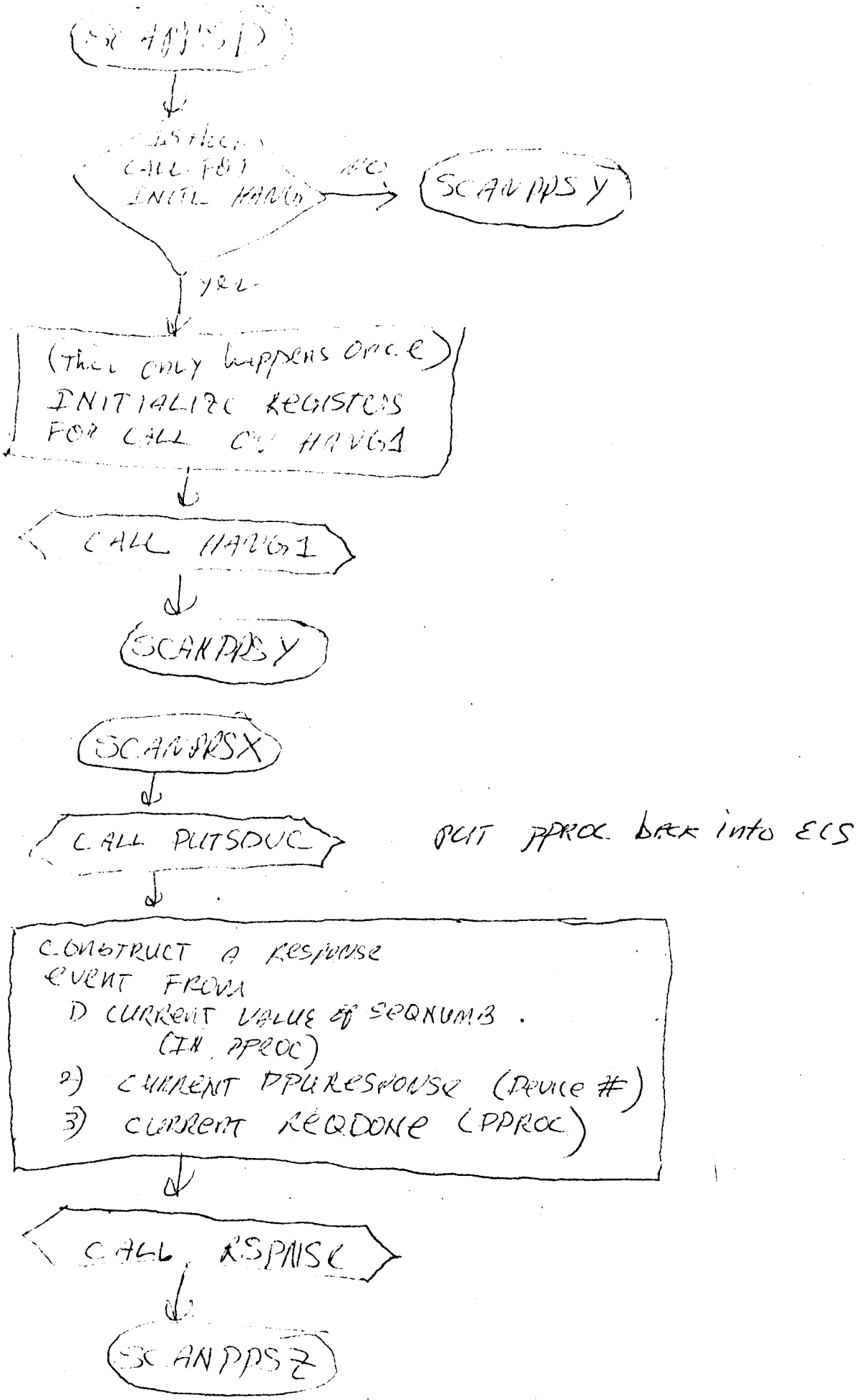
I didn't attempt to unravel the disk system interrupts, to be found under DSKINT,SOURCE. These two part names are representative of the way the code files are stored on the disk when the ECS system, or perhaps better described as the BEAD system is running on the B machine at the Cal Computer Center.

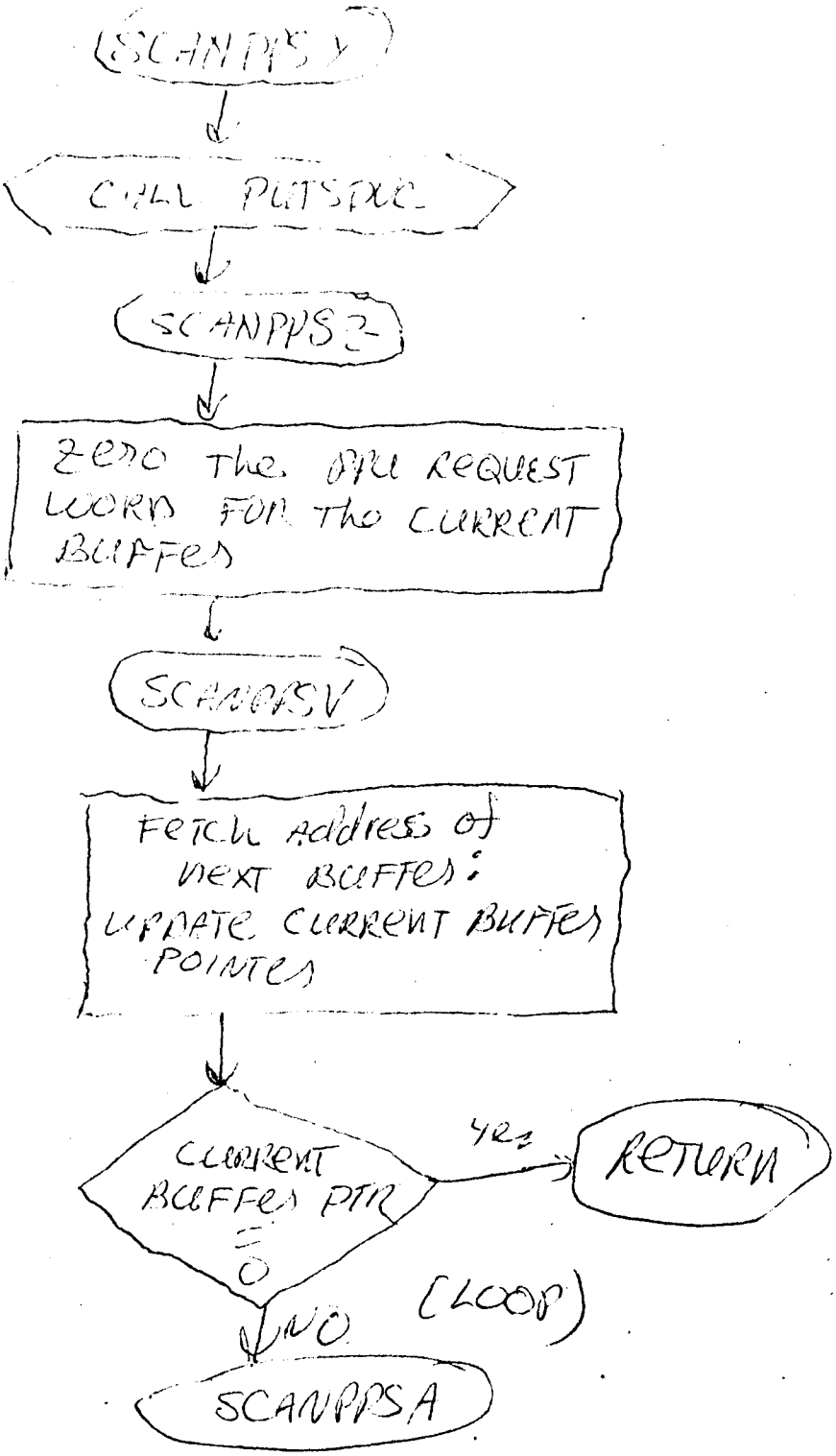
AN EXAMPLE:
S. DEVICE
INTERRUPTS
in CMI

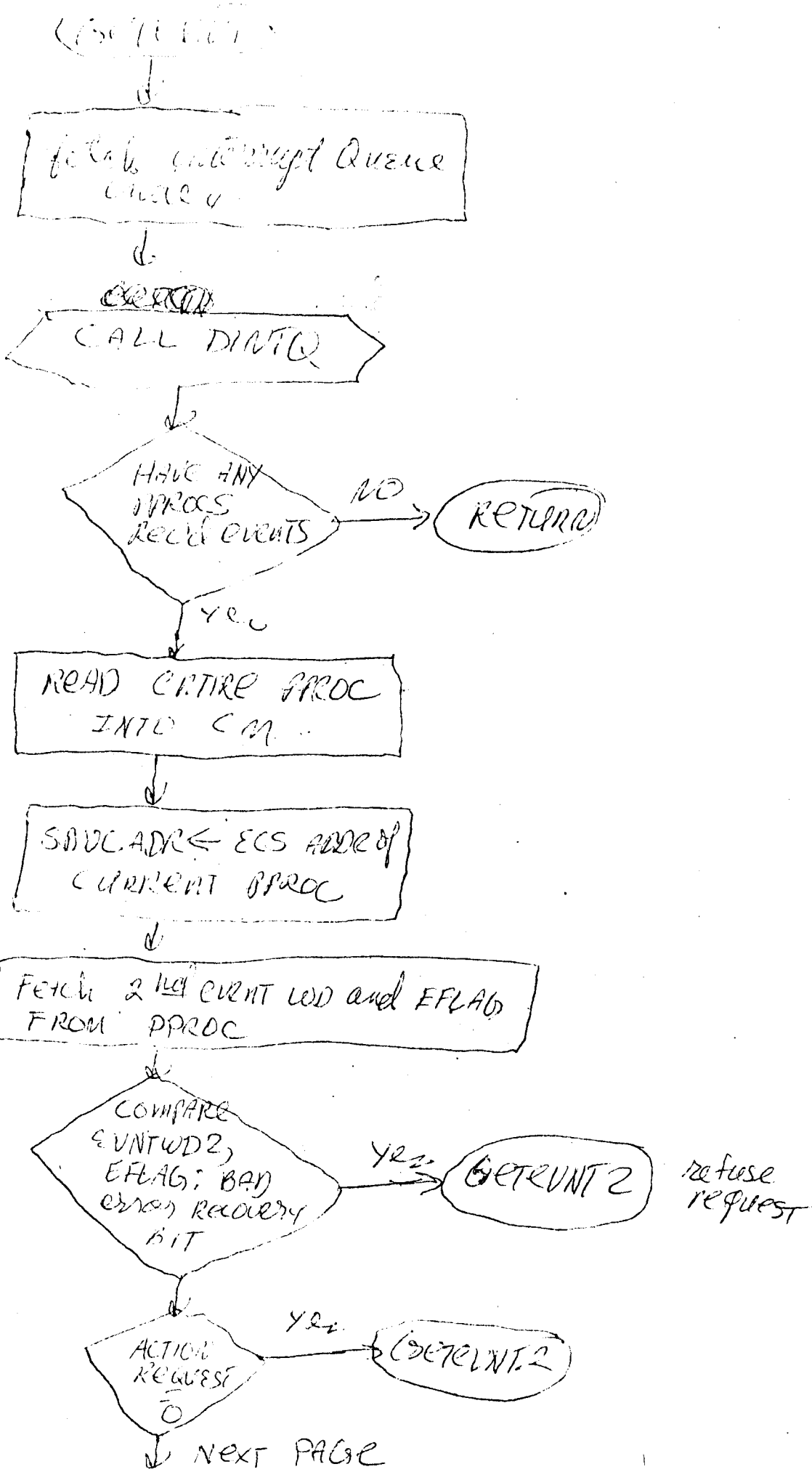


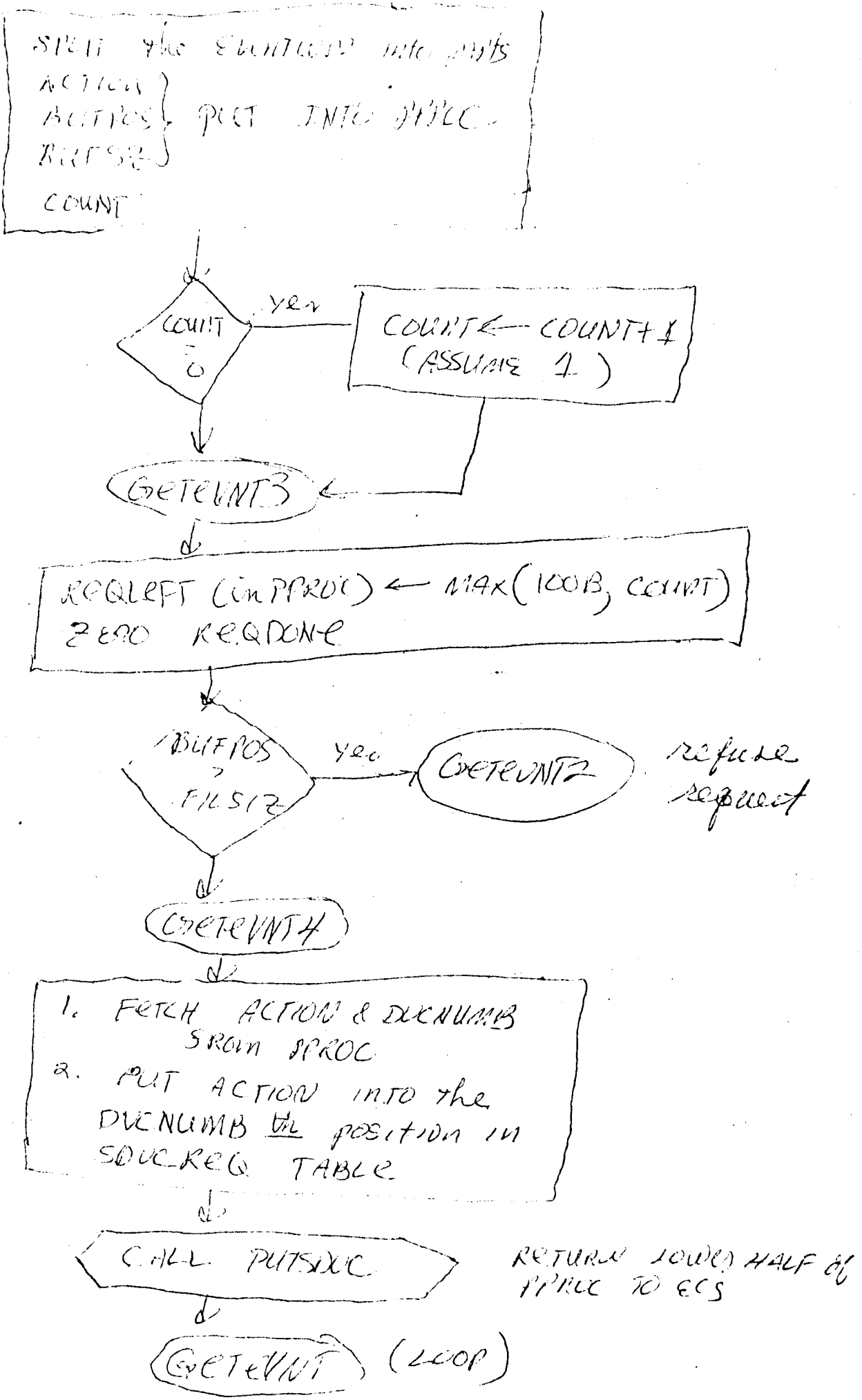

INITIALIZE CURRENT BUFFER PTR
SCANBS
Request contained with current buffer?
no
GO TO SCANBS V
yes
Fetch PPU request word.
ISOLATE
1) Action
2) Word count
3) Position in current buffer
4) Device number
CALL SETSDVC
get PPROC INTO CM
(bottom part of PPROC contains relevant data)
no
MOVE ECS->CM
yes
CALL SETMOVE
READ ECS
SCANBS
CALL SETMOVE
WRITE ECS
MOVE CM->ECS
yes
no
SCANPPC

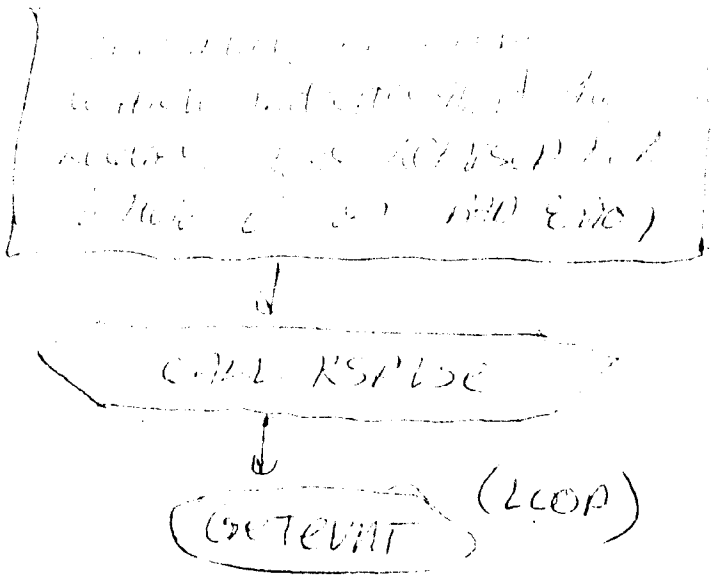

SCANPPC
NO
SCANPSD
YES
IN the CM version of the PPREQ:
REQUEST ← REQUEST+1.
REQLEFT ← REQLEFT+1
Fetch PPRESPONSE for this device
PPC RESPONSE < 0
yes
SCANPPSW
REQUEST
TERMINATE REQUEST, FLIP ERROR BIT
RESPONSE
SCANPPSX
TERMINATE REQUEST
RESET the buffer PTRS in the CM version of PPREQ
Fetch the current action REQUEST FROM this PPREQ and PUT INTO the PPREQUEST SLOT
SCANPPSY


SCANPPS
CALL FOR INITL HANG
NO
SCANPPSY
YES
(THIS ONLY HAPPENS ONCE)
INITIALIZE REGISTERS FOR CALL BY HANG1
CALL HANG1
SCANPPSY
SCANPPSX
CALL PUTSDUC
PUT PPROC BACK INTO ECS
CONSTRUCT A RESPONSE EVENT FROM
1) CURRENT VALUE OF SEQNUMB. (IN PPROC)
2) CURRENT PPURESPONSE (DEVICE #)
3) CURRENT REQDONE (PPROC)
CALL RSPAISC
SCANPPSZ

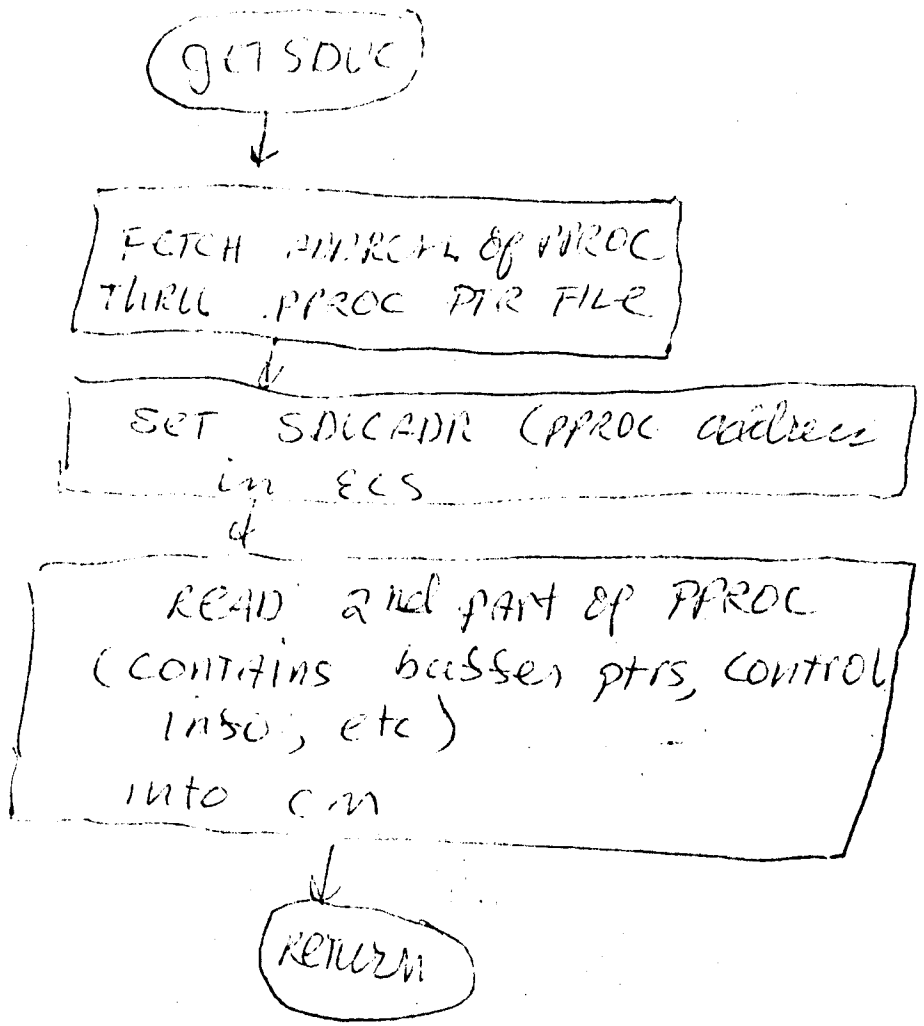

SCANPPS 1
CALL PUTSIXL
SCANPPS 2
ZERO THE DPU REQUEST WORDS FOR THE CURRENT BUFFER
SCANDSV
FETCH ADDRESS OF NEXT BUFFER; UPDATE CURRENT BUFFER POINTER
CURRENT BUFFER PTR = 0
YES
RETURN
NO
(LOOP)
SCANPPSA

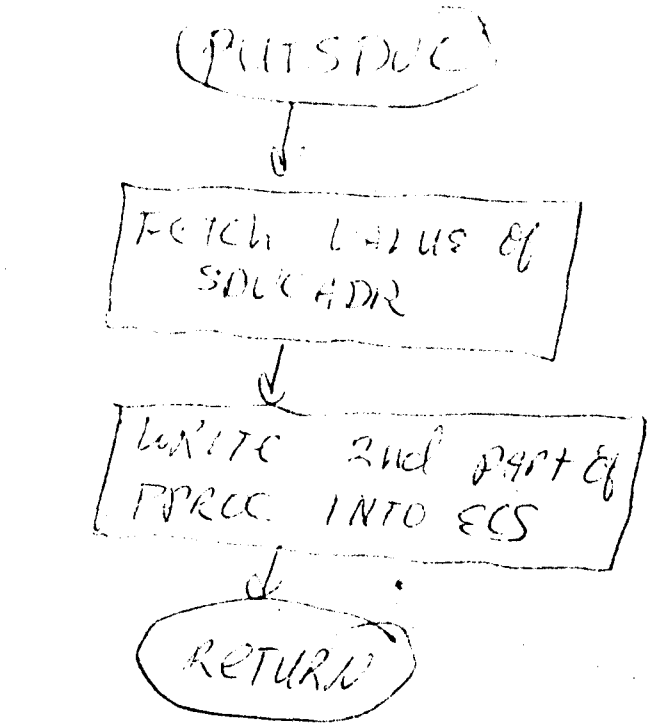

CALL DINTQ
HAVE ANY IPROCS RECD EVENTS
NO
RETURN
YES
READ CAUSE PROC INTO CM
SAVE ADR ← ECS ADDR of CURRENT PROC
Fetch 2 WD CURNT WD and EFLAG FROM PPROC
COMPARE EVNTWD2, EFLAG: BAD error RECOVERY BIT
YES
GETEVNT2
refuse request
ACTION REQUEST = 0
YES
GETEVNT2
NEXT PAGE

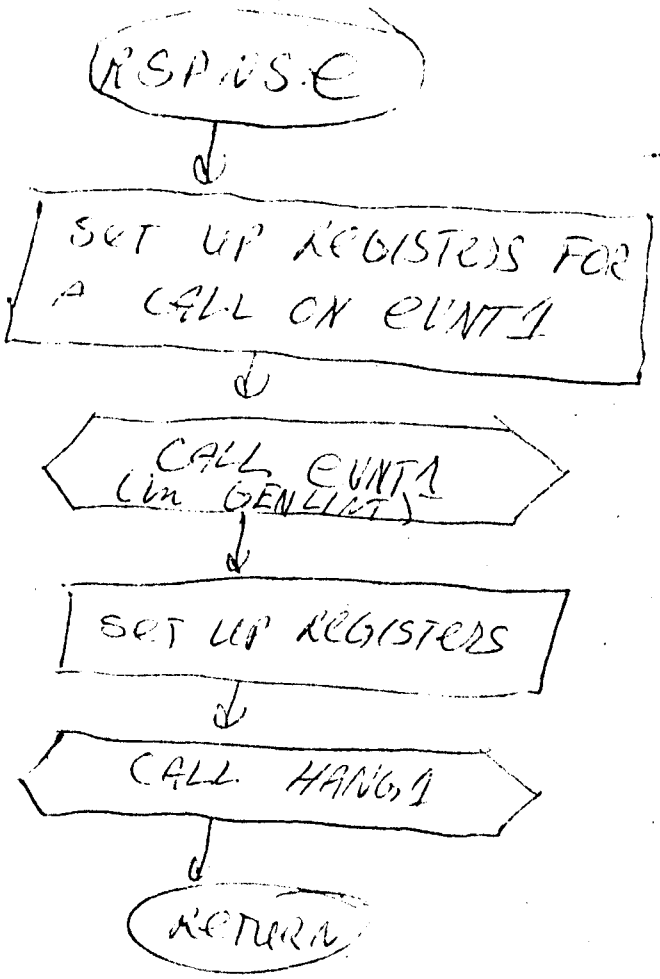

SPLIT the EVENTCODE into parts
ACTION
BUFPOS
BUFSIZ
COUNT
PUT INTO DPICC
COUNT = 0
yes
COUNT ← COUNT + 1
(ASSUME 1)
GETEVNT3
REQLEFT (in PPROC) ← MAX(100B, COUNT)
ZERO REQDONE
BUFPOS > FILSIZ
yes
GETEVNT2
refuse request
GETEVNT4
1. FETCH ACTION & DVCNUMB from SPROC
2. PUT ACTION into the DVCNUMB th position in SOUCREQ TABLE
CALL PUTSXC
RETURN lower HALF of PPICC TO ECS
GETEVNT
(LOOP)



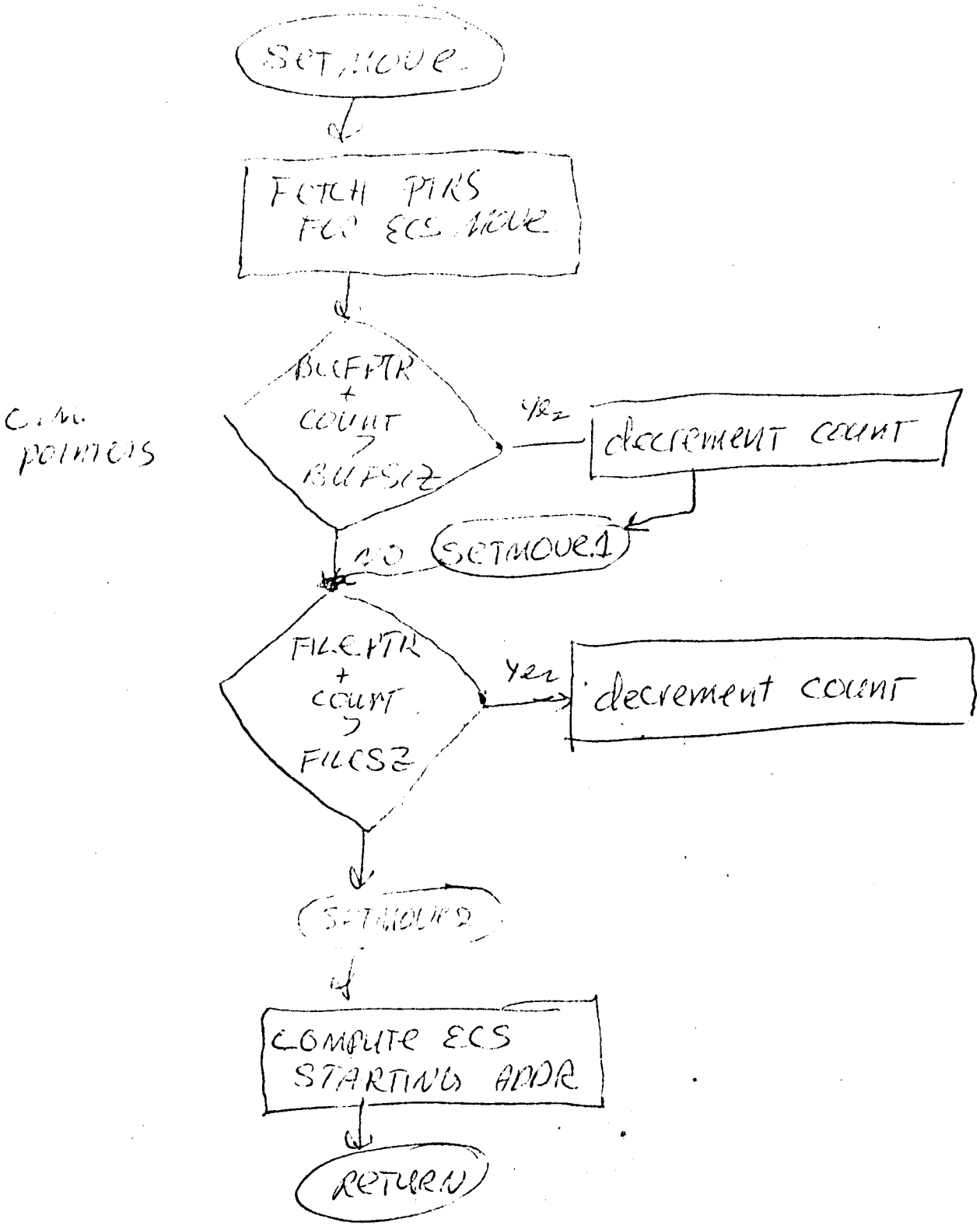
5 device interrupt subroutines







hang the PPROC


SETMOVE
FETCH PTRS FOR ECS MOVE
BUFPTR + COUNT > BUFSIZ
yes
decrement count
no
SETMOVE1
FILEPTR + COUNT > FILESZ
yes
decrement count
SETMOVE2
COMPUTE ECS STARTING ADDR
RETURN
C.M. POINTERS