Here's a copy of our current UNIX documentation.
The organization of the document is as follows:

A - Symbol Table (not done)
B - Memory Map
  B.1 - Core Memory Map
C - Routine Jump Network (who calls who)
D -
E - Listings
  E.1 - Uo
  E.1 - U1
  ...
F - General description of System
G - Description of data names
H - Routine descriptions
  H.0 - Uo Routine
  H.1 - U1 Routine
  ...

Hope it helps.

R. R. 463 - 4004

Jim Di Felice
140 Centennial Ave.
Rm G155

R. R. 463 - 4004

From Mr. L. C. H. 463 - 4004

0-463-6900
ID:  U0,2  /allocate tty buffers

FUNCTION:  Each DC-11 interface is assigned 140 bytes
of buffer space, the first 140-byte block beginning
at location "buffer." Also for each interface a
4-word block of control and status type information
is maintained. These 4-word blocks begin at
location "tty," the fourth word in each block is
a pointer to the beginning of the 140-byte buffer
assigned to that device. This section of code
loads these pointers into the proper places
in the tty blocks. The results are shown in the
diagrams on H.0 page

CALLING SEQUENCE

ARGUMENTS

INPUTS:  ntty (number of DC-11 interfaces)

OUTPUTS:  (see diagrams H.0 page 3)  r0, r1
FUNCTION - Block I/O devices (drum, disk, dectape) use blocks of size 256 words. Thus for each of "nbuf" block I/O buffers 256 words must be assigned. In addition to the 256 words for data each block has four additional words which represent an I/O queue entry. Thus each block contains 260 words. These blocks begin at location "buffer + 1260.". This segment of code loads pointers to these 260 word blocks in consecutive locations starting at "bufp." Thus "bufp" contains pointers to I/O queue entries since the first four words in each block represent the I/O queue entry for the block. Three additional I/O queue entries located at locations "sb0", "sb1", and "swp" also exist and pointers to them are also loaded into "bufp". Finally, the last 2 words of an I/O queue entry contain a word count and a bus address, these locations are initialized. The results are shown in the diagrams on I/O page 3.

CALLING SEQUENCE

ARGUMENTS

INPUTS  To (points to first block I/O buffer)

OUTPUTS (see diagrams I/O page 3), r1 (counter), r2 (internal pointer)
ID  wo;3 / free all character blocks

FUNCTION: This segment of code initializes the cf, cl and clist blocks in core to the following state:

<table>
<thead>
<tr>
<th>255</th>
<th>255</th>
<th>cf</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(cf+31)</td>
<td>cl</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>clist (cf+31)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>253</td>
<td>cclist + 500</td>
<td></td>
</tr>
<tr>
<td>254</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

CALLING SEQUENCE:
ARGUMENTS:
INPUTS:
OUTPUTS:
CALLED BY:
CALLS: PUT
ID - 4052 / set up drum swap addresses

FUNCTION - The drum is divided into 1024 blocks of 256 words. The highest 64 blocks are set aside for storing UNIX itself. Processes swapped to and from core are stored on the drum. The area in core beginning at location pdska contains a block number which is the number of the first block on the drum where the process is swapped to. There are 17 blocks on the drum assigned as swapping area for each process.

This segment of code initializes the pdska area in core by supplying the block numbers for each of "nproc" processes. The results appear as follows:

<table>
<thead>
<tr>
<th>643.</th>
<th>pdska</th>
</tr>
</thead>
<tbody>
<tr>
<td>926.</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>960. - nproc/7</td>
<td></td>
</tr>
<tr>
<td>pdska + 2*nproc - 2</td>
<td></td>
</tr>
</tbody>
</table>

CALLING SEQUENCE -

ARGUMENTS -
INPUTS -
OUTPUTS - pdska - [pdska + 2*nproc - 2] / 11, 12
FUNCTION - This portion of code is executed during 'cold' boot. (See UNIX Programmers Manual - Boot Procedures, VII.) It initializes the core image of the super block for the fixed head disk, Systm (which represents the number of bytes in the free storage map) is set to 128. Systm + 130. (which represents the number of bytes in the i-node map) is set to 64. ... (See Sec F, pp. 1, 2). Blocks 34, ... 687, on the drum are freed (the corresponding bits in the free storage map are set). These blocks are for user files.

CALLING SEQUENCE -

ARGUMENTS -

INPUTS - R1 contains the number of the highest block to be freed. (See inputs for 'free'; H.5, p. 2)

OUTPUTS - Systm, Systm + 6, Systm + 8, ... , Systm + 85, Systm + 130. (See outputs for 'free'; H.5, p. 2)

R1 used internally.
FUNCTION - This portion of code is executed during 'cold' boot. (See UNIX Programmers Manual - Boot Procedures (III).) It zeros blocks 1, ..., 33, on the drum. Block 1 is the 2nd block of the superblock for the drum. (Block 0 is the 1st block of the superblock. However since the in-core image of the superblock (see UNIX Implementation Manual - p3) is updated onto the RK03 whenever it is changed (can be changed by a call to 'free', updated by a call to 'sysre') it does not have to be zeroed.) Blocks 2, ..., 33, are used for i-nodes 1 thru 512. (See See F pp 1, 3, 4, 5)

CALLING SEQUENCE -

ARGUMENTS -

INPUTS - R1 Contains the number of the highest block to be zeroed + 1.

(See inputs for 'clear' H.3 p.1)

OUTPUTS - Blocks 2, ..., 33, on disk are zeroed.

(see outputs for 'clear' H.3 p.1)

R1 used internally
FUNCTION "sysclose", given a file descriptor in u.iro, closes the associated file. The file descriptor (index to the ufp list) is put in RI and "fclose" is called. (See "fclose" 1.2)

CALLING SEQUENCE: sysclose

ARGUMENTS:

INPUTS (u.iro) - file descriptor

OUTPUTS See fclose outputs
FUNCTION: "syscreat" is called with two arguments: name and mode. u.namep points to the name of the file and the mode is put on the stack. "name" is called to get the c-number of the file. If the file already exists, its mode and owner remain unchanged, but it is truncated to zero length. If the file did not exist, an c-node is created with the new mode via "maknod." Whether or not the file already existed, it is open for writing. The tsp table (See F page 8) is then searched for a free entry. When a free entry is found the proper data is placed in it (See outputs below), and the number of this entry is placed in the u.tfp list. The index to the u.tfp (also known as the file descriptor) is put in the users RO.

For more information see SYS CREAT in the users manual.

ARGUMENTS

name - name of file to be created
mode - mode "" "" ""

INPUTS

RI - c-number of file if found
sp - contains the mode argument
w, x, y, z - if file not found, contains c-number of new file
u.fsp - list of file descriptors
u.tsp - table of open file entries

OUTPUTS

if file not found - new c-node is created (See maknod)
R1 - contains c-number of new file
R2 - index into tsp table (file descriptor)
R3 - index into u.tfp list
R4 - index into tsp entry

1st word c-number of new file
2nd word device number
3rd word 0
4th word 0

u.tfp list - entry number of new tsp entry
(u.tfp) - index h u.tfp list (file descriptor of new file

ISSUE 1) DATE 2/18/72 ID UN1X SEC 11.1 PAGE 0.1
ID U1-1  System; unkni

FUNCTION; unkni or systent is the system entry from various traps. The trap type is determined and an indirect jump is made to the appropriate system call handler. If there is a trap inside the system a jump to passive is made. All user registers are saved and vsp points to the end of the users stack. The sys (trap) instruction is decoded to get the system code part. (See Trap instruction in the PDP-11 hand book) and from this the indirect jump address is calculated. If a bad system call is made, i.e., the limits of the jump table are exceeded, "badsys" is called. If the call is legitimate control passes to the appropriate system routine

CALLING SEQUENCE through a trap caused by any sys call outside the system

ARGUMENTS arguments of the particular system call

INPUTS  &syst+2, RO, SP, R1,R2, R3,R4, R5, M1, M2, SC

OUTPUTS clockp - contains &syst+2 v, ro - points to the location of the users RO on the stack RO - SC is saved on the stack v, vsp - points to the end of the users stack
FUNCTION sys exit

sys exit terminates a process. First each file that the process has opened is closed by close. The process status is then set to unused. The process ppid table is then searched to find children of the dying process. If any of the children are zombies (dead but not waited for) they are set free, the ppid table is then searched to find the dying process's parent. When the parent is found, it is checked to see if it is free or is a zombie. If it is, one of these the dying process just dies. It is waiting for a child to die, it is notified that it doesn't have to wait anymore by setting its status from 2 to 1 (waiting to active). It is then awakened and put on the run by "putdu". The dying process enters a zombie state in which it will never run again but stays around until a "wait" is completed by its parent process. If the parent is not found, the process just dies. This means, swap is called with_phylo = 0. What this does is that withswap is not called to write out the process and rswap reads a new process over the one that dies. i.e., the dying process is overwritten and destroyed.

CALLING SEQUENCE sys exit or conditional branch

ARGUMENTS:

INPUTS
v, uno - the process number of the dying process
ppilld - contains the name of the process (See Page 10)
ppid - contains the name of the parent process
pstat - the status of the process

OUTPUTS
uno - determines handling of interrupts - if it is set to 0 all open files of the process are closed
r3 - contains the dying process's name, on a number
r4 - contains its parent's name
r2 - is used to scan the process tables
children of the dying process are freed
r1 & r5 are used to hold the parent's process number if the parent of the dying process is waiting it is set to active and the dying process is made a zombie and its parent is put on the run's chand and the process is killed
FUNCTION: systork creates a new process. This process is referred to as the child process.

This new process's core image is a copy of that of the caller of "systork." The only distinction is the return location and the fact that (vuro) in the old process (parent) contains the process id (ppid) of the new process (child). This id is used by "syswait." systork works in the following manner:

1) The process status table (pstat) is searched to find a process number that is unused. If none are found an error occurs.
2) When one is found, it becomes the child process number and its status (pstat) is set to active.
3) If the parent had a control tty, the interrupt character is that tty buffer is cleared.
4) The child process is put on the lowest priority run queue via "putq".
5) A new process name is gotten from mpid. Actually its a unique number and is put in the child's unique identifier; the process id (ppid)
6) The process name of the parent is then obtained and placed in the unique identifier of the parent process of the child (ppid). The parent process name is then put in (vuro)
7) The child process is then written out on disk by "syswrite", i.e., the parent process is copied onto disk and the child is both
8) The parent process number is then restored & vuro
9) The child process name is put in (vuro)
10) The pc on the stack, sp +18, is incremented by 2 to create the return address for the parent process.
11) The ufp list is then searched to see what files the parent has opened. For each file the parent has opened, the corresponding tsp entry must be updated to indicate that the child process also has opened the file. A branch to systok is then made.

CALLING SEQUENCE
ARGUMENTS
INPUTS
p,stat status of a process active, dead, unused
p,mpd parent process number
v,tt,pp control tty buffer
ppid, pid process name generator
ufp - list of index into tsp table
fsp - list of open files
 outputs:

- p,stat - byte for child process is set to active.
- if control ity for parent exists, button +2 is cleared.
- child process number is put on pung +4.
- pipid - appropriate entry in the table contains the name of the child process.
- the child process is written on drum with wono being the child's process number and (wiro) containing the parent process number.
- (wiro) - contains the child's process name.
- sp+18 - gets 2 added to it to change the return address of the parent.
- tsp+6 - "number of processes that have opened the file" byte gets incremented in the particular tsp entry.
FUNCTION: "sysrdate" is given a file name. It gets the i-node of the file into core. The user is checked to see if he is the owner or the super user. If he is neither an error occurs. "selinux" is then called to set the i-node modification byte and the modification time, but the modification time is overwritten by whatever got put on the stack during a "sysrtime" call. (See sysrtime)
These calls are restricted to the super user.

CALLING SEQUENCE
sysrdate; name

ARGUMENTS
name - pointer to file name

INPUTS:
u.uid, users id
i.uid, owner id

OUTPUTS:
new modification time of the file
FUNCTION "sysctlty" gets the status of the tty in question. It stores in the three words addressed by its argument the status of the typewriter whose file descriptor is in (u,0).

CALLING SEQUENCE sysgtty ; arg

ARGUMENTS: arg - address of 3 word destination of status

INPUTS:
R1 - tty block offset
R2 - destination of status data

resr+R1 - reader control status
tcsr+R1 - printer control status register

tty+4+R1 - flag byte in tty block which contains the mode

OUTPUTS:
(R2) - contains the reader control status
(R2)+2 " " printer " "
(R2)+4 " " mode
ID UI-8 sysmkdir

FUNCTION: "sysmkdir" creates an empty directory whose name is pointed to by arg1. The mode of the directory is arg2. The special entries "." and ".." are not present.

Errors are indicated if the directory already exists or the user is not the super user.

CALLING SEQUENCE
sysmkdir ; name ; mode

ARGUMENTS
name - points to the name of the directory
mode - mode of the directory

INPUTS
uid - user id; if it's 0 the user is the super user
(sp) - contains the second argument "mode"

OUTPUTS
- makes an i-node for the directory via "makinode"
- sets up the flag in the directory i-node
- sets user id on executable directory
why not use \( t \) as subscripts 1, \ldots, 9 rather than 0, \ldots, 9 ?

\[ R_2 = \sigma, \ldots, 9 \]

the proper \( t_{5p+8} \) loaded with \( 21 \) (\#1)

\[
\begin{align*}
\tau_{5p+8} & = \sigma & \tau_{5p+10} & = 0 \\
\tau_{5p+12} & = \gamma &= 0 \\
\end{align*}
\]

\[ u \cdot t_p + \frac{X}{8} = \frac{\gamma}{8} + 1 \]

\[ (u \cdot r_0 = \frac{\gamma}{8} \Rightarrow) \]

\[ R_3 = \frac{\gamma}{8} + 1 \]
ID: UI-2  ERROR
FUNCTION "error": merely sets the error bit of the processor
status (c-bit) and then falls right into the sysreset, sysrek
return sequence.

CALLING SEQUENCE: conditional branch to error

ARGUMENTS:

INPUTS:

OUTPUTS: processor status - c bit is set (means error)
FUNCTION: "badsys" is called either because the user executed an illegal trap (e.g., instruction or because a 16-bit trap occurred). (The 16-bit trap is used to implement the quit function) "badsys" first turns on the bad system flag (wubsys) and then calls "printf" with uparg pointing to "core". The core image file is then opened for writing via "io open". If the file is not found, and i-node whose mode is 17 is found by "maknode", and the i-number for that node is put in RL, then RL is used to write out core aoe then set up and the core image is written into user's directory. Then the users area of core area written out, the file closed, sysexit is entered to terminate the process.

CALLING SEQUENCE
this badsys

ARGUMENTS

INPUTS:
RL - i-number of core image file's i-node

OUTPUTS:
wubsys - turn on, its the users bad system flag
wbase - holds address of "core" and user during write i-calls
wcount - users byte count to write out
wtop - contains file offset
woff - set to zero
RL - has i-number of core image file
FUNCTION `gtty` is called by `sysgetty` and `sysstty`.
It takes the first argument of the above calls and puts it in R2. This argument is either the source or destination of information about the tty in question. The file descriptor is put in R1 and the i-number of the file is obtained via `getf`. The number of the tty is gotten by (the i-number - 14). If no tty with this number exist and error occurs. 8 x (i-number-14) is the tty block offset. This is outputed in R1.

CALLING SEQUENCE: jsr ro, gtty

ARGUMENTS:

INPUTS: (UIO) - contains the file descriptor for the tty file
        R1 - i-number of file

OUTPUTS R1 - tty block offset
           R2 - source or destination of information
ID: U1; 7 for 2

FUNCTION - see 'error' routine
CALLING SEQUENCE -
ARGUMENTS -
INPUTS -
OUTPUTS -
FUNCTION — see 'error'

ARGUMENTS —

CALLING SEQUENCE —

INPUTS —

OUTPUTS —
**ID** UI-6  **sysopen**

**FUNCTION**

"sysopen" opens a file in the following manner:

1) The second argument in a sys open calls says whether to open the file to read (0) or write (1).
2) The c-node for the particular file is obtained via "namei".
3) The file is then opened by "iopen".
4) Next housekeeping is performed on the fsp table and the user's open file list - uifp.
   a) uifp and fsp are scanned for the next available slot.
   b) An entry for the file is created in the fsp table.
   c) The number of the entry is put on the uifp list.
   d) The index to the uifp list is pointed to by uiro.

**ARGUMENTS**

<table>
<thead>
<tr>
<th>Arguments</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>File name or path name</td>
</tr>
</tbody>
</table>
| mode      | 0 - open for reading  
|           | 1 - open for writing |

**INPUT**

R1 - contains an I-number (positive or negative depending on whether an open for read or open for write is desired.

**OUTPUT**

Entry in fsp table and uifp list
(u, ro) - index to uifp list (the file descriptor) is put into E0's location on the stack.
R2 - used as a counter through the uifp list.
R3 - used as a pointer to the beginning of an fsp entry.

**CALLING SEQUENCE**

sys open; name; mode
sysreth

sysreth first checks to see if the process is about to be terminated (in brk). It it is sysreth is called. If not the following happens:

1) The user's stack pointer is restored
2) RL = 0 and "get" is called to see if the last mentioned c-node has been modified. If it has, it is written out.
3) If the super block has been modified, it is written out via "ppoke"
4) If the dismountable file system's super block has been modified, it is written out to the specified device via "ppoke."
5) A check is made to see if the user's time quantum (uguard) ran out during his execution. If so, "eswap" is called to give another user a chance to run.
6) sysreth now goes into sysrele. (See sysrele for conclusion)

Calling sequence: jump table on br sysreth

Arguments:

Inputs:
- u, b, y, s, x, u, b, a, d, s, t, e, n, f, l, g, a, i, n, y, "get" call
- r, l - used internally
- set RL to 0 for "get" call
- smod - set if super block has been modified
- mmod - set if dismountable file system's super block has been modified
- u, g, a, r, d - users time quantum

Outputs:
- s, p - points to user's stack
- smod - cleared if it was set
- mmod - cleared if it was set
- "write", "bit", "is", "bit", "set", "during", "execution", "of", "sysreth"
ID - UI55  sysret1

FUNCTION - see 'sysret1'

CALLING SEQUENCE - 

ARGUMENTS - 

INPUTS - 

OUTPUTS - 
FUNCTION:
CALLING SEQUENCE:
ARGUMENTS:
INAIUS:
OUTPUTS:

see 'Sysnet' routine
ID UI-2  sysrele

FUNCTION "sysrele" first calls dswap if the time quantum for a user is zero (see sysret). It then restores the user's registers and turns off the system flag. It then checks to see if there is an interrupt from the user by calling "isintr". If there is the output gets flushed (see isintr) and interrupt action is taken by a branch to interrupt. If there is no interrupt from the user a relu is made.

CALLING SEQUENCE fall through a "one" in sysret ?

ARGUMENTS —

INPUTS  stack  (s, ch1gt+2) ?

OUTPUTS  sc, m, a, ac, r5, r4, r3, r2, r1, r0 restored
           sysflag turned off
           clock points to s, ch1gt+2
FUNCTION: ‘sysstty’ gets the status and mode of the typewriter whose file descriptor \l (uiro) first ‘\l getty’ is called to get the tty block and the source of the status information. ‘getty’ is called until the input list is flushed. The output character list is checked. If some characters are on it, the process is put to sleep and the input list is checked again. If there are no characters, the information in the source is put into the reader control status, printer control status registers and the tty flag byte in the tty block.

CALLING SEQUENCE: sysstty \l arg

ARGUMENTS: \l arg - address of three consecutive words that contain the source of the status data

INPUTS: R1 - offset to tty block
R2 - points to the source of the status information, see \l arg above
R1+tty+y+3 contains the cc offset
R3 - used to transfer the source information to the tty status registers and block

OUTPUTS: P5 - set to 5
PSR+R1 - contains new reader control status
TSSR+R1 - contains new printer control status
tty+y+R1 - contains new mode in the flag byte of the tty block
syswait waits for a process to die. It works in the following way:

1) From the parent process number, the parent's process name is found. The p.ppid table of the parent name is then searched for the process name. If a match occurs, R2 contains the child's process number. The child's status is checked to see if it's a zombie, i.e., dead but not waited for (p.stat=3). If it is the child process is freed and its name is put in R3. A return is then made via `sysret`. If the child is not a zombie, nothing happens and the search goes on through the p.ppid table until all processes are checked or a zombie is found.

2) If no zombies are found, a check is made to see if there are any children at all. If there are none and the return is made, if there are, the parent's status is set to 2 (waiting for child to die); the parent is swapped out and a branch to `syswait` is made to wait on the next process.

Calling sequence:

Arguments:

Inputs:

- Void: parent process number (process number of process in core)
- p.ppid: table of names of processes
- p.stat: contains status of process
  0: Free or Unused
  1: Active
  2: Waiting for process to die
  3: Zombie

Outputs:

- R2: used as index to p.ppid, p.pstat, and p.tables
- R3: used to keep track of the number of children
- If zombie found - status of process is freed (set to 0)
  - its name is put in R3 (waiting for child to die)
  - parent is swapped out
- If no zombies found - status of parent is set to 2 (waiting for child to die)
FUNCTION: sysread is given a buffer to read into and the number of characters to be read. It finds the file from the file descriptor located in *u.i.ro (ro). This file descriptor is returned from a successful open call. (See sysopen 111 page 1). The i-number of the file is obtained via "rw1" and the data is read into core via "readi".

ARGUMENTS:
- buffer - location of contiguous bytes where input will be placed.
- nchars - number of bytes or characters to be read

INPUTS
- RI - contains i-number of file to be read

OUTPUTS
- (u.i.ro) contains the number of bytes read

CALLING SEQUENCE
- sysread; buffer; nchars
ID UI 6 syswrite

FUNCTION: syswrite is given a buffer to write, an output file and the number of characters to write. It finds the file from the file descriptor located in *u.ro* (50). This file descriptor is returned from a successful open or creat call. (See sys open or syscreat). The c-number of the file is obtained via "rwi," and the buffer is written on the output file via "writei."

ARGUMENTS

buffer - location of contiguous bytes to be written

nchar - number of characters to be written

INPUTS

R1 - contains the c-number of the file to be written on.

OUTPUTS

(u.ro) - contains the number of bytes written

CALLING SEQUENCE

sys write; buffer; nchar
FUNCTION: "anyi" is called if a file has been deleted while open, "anyi" checks to see if someone else has opened this file. It searches the tsp table for an i-number contained in R1. If that i-number is found (if someone else opened the file) the "file deleted" flag in the upper byte of the 4th word of the tsp entry is incremented. (See F page 8). In other words the deleted flag is passed on to the other entry of this file in the tsp table.

Note: The same file may appear more than once in the tsp table.

If the i-number is not found in the tsp table (no one else has opened the file) the corresponding bit in the i-node map is cleared freeing that i-node and all blocks related to that i-node.

INPUTS

R1 - contains an i-number
R2 - start of table containing open files

OUTPUTS

"deleted" flag set in tsp entry of another occurrence of this file and R2 pts to 1st word of that tsp entry

if file not found - bit in i-node map is cleared
all blocks related to i-node are freed

CALLING SEQUENCE

jal ro, anyi - all flags in i-node are cleared
ARG extracts an argument for a routine whose call is of form:

\[
\text{SYS 'ROUTINE'; ARG1} \\
\text{or} \\
\text{SYS 'ROUTINE'; ARG1, ARG2} \\
\text{or} \\
\text{SYS 'ROUTINE'; ARG1; ...; ARGN} \quad (\text{System})
\]

CALLS SPECIFICS: JSR RO, ARG- J 'ADDRESS'

PRECONDITIONS: 'ADDRESS' - Address in which extracted argument is stored.

INPUTS: USP+18. - Contains a pointer to one of ARG1...ARGN. This pointer's value is actually the value of the updated RO at the time the TRAP to SYSENT(UNKNL) is made to process the SYS instruction.

RO - Contains the return address for the routine that called ARG. The data in the word pointed to by the return address is used as the address in which the extracted argument is stored.

OUTPUTS: 'ADDRESS' - Contains the extracted argument.

U8P+18. - Is incremented by 2.

R1 - Contains the extracted argument.

RO - Points to the next instruction to be executed in the calling routine.

CALLED BY: RV1, SYSERR, SYSILGINS, SYSMDATE, GTY, SYSUNLINK, SYSSYNT, SYSCACH, ARP2, SYSBREAK, SEEKTELL, SYSSINR, SYSSQUIT, SYSSAUTH
ID: u2, 7  arg2:

FUNCTION: Takes first arg in system call (pointer to name of file) and puts it in location U.namep; Takes second arg and puts it in U.off and on top of the stack.

CALLING SEQUENCE: for r0, arg2

ARGUMENTS:

INPUTS: u.sp, r0

OUTPUTS: U.namep
        U.off
        U.off ↓ sp
FUNCTION - see 'error' routine
CALLING SEQUENCE -
ARGUMENTS -
INPUTS -
OUTPUTS -
ID - 4231    error 4

FUNCTION - see 'error' routine
CALLING SEQUENCE -
ARGUMENTS -
INPUTS -
OUTPUTS -
INPUTS - user parameters (see sec G)
OUTPUTS - user parameters
FUNCTION: "sysfstat" is identical to "sysstat" except that it operates on open files instead of files given by "name". It puts the buffer address on the stack gets the i-numbers and checks to see if the file is open for reading or writing. If the file is open for writing (i-number is negative) the i-number is set positive and a branch into sysstat is made.

CALLING SEQUENCE: sysfstat (buf

ARGUMENT: buf - buffer address

INPUTS: (w|r|o) file descriptor

OUTPUTS: - buffer is loaded with file information. See UNIX PROGRAMMERS manual under SYSSTAT (4) for format of the buffer
sysgetuid returns the real user ID of the current process. The real user ID identifies the person who is logged in, in contradistinction to the effective user ID, which determines his access permission at each moment. It is thus useful to programs which operate using the "set user ID" mode, to find out who invoked them.

CALLING SEQUENCE: sysgetuid

ARGUMENTS:

INPUTS: u,ruid - real users id

OUTPUTS (u,ro) - contains the real users id
FUNCTION "sysintr" sets the interrupt handling value. It puts the argument of its call in v.intr. "sysintr" then branches into the "sysqintr" routine. v.tty is checked to see if a control tty exists. If one does not exist character in the tty buffer is cleared and sysqintr is called. If one does not exist sysqintr is just called.

CALLING SEQUENCE: sysintr; arg

ARGUMENT arg
- if 0, interrupts (ASCII DELETE) are ignored
- if 1, interrupts cause their normal result, i.e., force an exit.
- if arg is a location within the program, control is passed to that location when an interrupt occurs

INPUTS: v.tty - pointer to control tty buffer

OUTPUTS v.intr has value of arg
(R1)+6 (interrupt character tty buffer) is cleared if a control tty exists
FUNCTION: syslink is given two arguments named \texttt{name1} and \texttt{name2}. \texttt{name1} is a file that already exists. \texttt{name2} is the name given to the entry that will go in the current directory. \texttt{name2} will then be a link to the \texttt{name1} file. The c-number in the \texttt{name2} entry of the current directory is the same c-number for the \texttt{name1} file. At the end of a syslink call the following structure is constructed:

ARGUMENTS
- \texttt{name1} - file name to which link will be created
- \texttt{name2} - name of entry in current directory (the link to \texttt{name1})

INPUTS
- \texttt{u.namep} - points to the arguments above

OUTPUTS
- entry in the current directory with name \texttt{name2} containing c-number of \texttt{name1} on exit and c-number of current directory intermittently during subr.
- \texttt{cinks} - incremented by 1 to indicate another link added.
- \texttt{cored} - set by call to setmed.
ID- 0234  sysret3

FUNCTION  see 'sysret' routine

CALLING SEQUENCE

ARGUMENTS-

INPUTS-

OUTPUTS-
FUNCTION: see 'Sysret' routine

ARGUMENTS:

INPUTS:

OUTPUTS:
ID: 4251  Sys ret 9

FUNCTION:
see 'sys ret' routine

CALLING SEQUENCE:

ARGUMENTS:

INPUTS:

OUTPUTS:
sysseek

FUNCTION: sysseek changes the R/W pointer (3rd word in an FSP entry) of an open file whose file descriptor is in &v.roc.

The file descriptor refers to a file open for reading or writing. The read (or write) pointer for the file is set as follows:

if ptrname is 0, the pointer is set to offset.

if ptrname is 1, the pointer is set to its current location plus offset.

if ptrname is 2, the pointer is set to the size of the file plus offset.

The error bit (c-bit) is set for an undefined file descriptor.

ARGUMENTS

offset - number of bytes desired to move the R/W pointer by

ptrname - a switch indicated above

INPUTS

wbase - ? See seek tell

wcount - ?

OUTPUTS

wfp - pts thie R/W pointer in the FSP entry

The R/W pointer is changed according to offset and ptrname.

CALLING SEQUENCE

sysseek, offset, ptrname
FUNCTION `sys_setuid`

`sys_setuid` sets the user id `uid` of the current process to the process id (`uid`). Both the effective user `uid` and the real user `uid` are set to this. Only the super user and `make` have this call.

**Calling Sequence**

`sys_setuid`

**Arguments -**

**Inputs**

- `uid` - contains the process id
- `uid` - real user id
- `uid` - effective current user id

**Outputs**

- `uid` - set equal to the process id (`uid`)
- `uid` - " " " " " " " " 
FUNCTION

"sysstat" gets the status of a file. Its arguments are the name of the file and a buffer address. The buffer is 34 bytes long and information about the file is placed in it. `sysstat` calls "namei" to get the i-number of the file. Then "get" is called to get the i-node in core. The buffer is then loaded and the results are given in the UNIX programmer's manual, `SYS STAT (II)`. CALLING SEQUENCE `sysstat; name; out`

ARGUMENTS: name - points to the name of the file out - address of a 34-byte buffer

INPUTS: sp - contains the address of the buffer

OUTPUTS - buffer is loaded with file information
FUNCTION "SYSSTIME" sets the time. Only the super user can use this call.

ARGUMENTS:

INPUTS - sp+2, sp+4  time system is to be set to

OUTPUTS - s, time, s.time+2  new time system read to
FUNCTION "sys_time" gets the time of the year. The present time is put on the stack.

CALLING SEQUENCE

ARGUMENTS:

INPUTS: s_time, s_time + 2 - present time

OUTPUTS: sp+2, sp+4 - present time
10 U2 -8  sys quit

FUNCTION: sys quit turns off the quit signal. If no, the argument
of the call in u.quit, utty is checked to see if a control
byte exists. If one does, the interrupt character in
the tty buffer is cleared and syret is called. If one
does not exist, syret is just called.

CALLING SEQUENCE
sys quit

ARGUMENT:
arg: - if 0, this call disables quit signals from
the typewriter (ASCII FS)
- if 1, quit is re-enabled and cause execution
to cease and a core image to be produced.
- if an address in the program, a quit causes
control to be sent to that location.

INPUTS
utty - pointer to control tty buffer

OUTPUTS
u.quit - has value of arg
(R1) + 6 - (interrupt character in buffer) is cleared
if a control byte exists
FUNCTION: sysunlink

removes the entry for the file pointed to by name from its directory. If this entry was the last link to the file, the contents of the file are freed and the file is destroyed. If, however, the file was open in any process, the actual destruction is delayed until it is closed, even though the directory entry has disappeared.

The error bit (c-bit) is set to indicate that the file does not exist or that its directory cannot be written. Write permission is not required on the file itself. It is also illegal to unlink a directory (except for the super-user).

ARGUMENTS

name - name of directory entry to be removed

INPUTS

unamet - points to name

RI - i-number associated with name

OUTPUTS

links - number of links to file gets decremented

woff - gets moved back 1 directory entry

mod - gets set by call to setmod

if name was last link contents of file freed; file destroyed

entry "name" in directory is free (its first

word that usually contains and i-number is zeroed.

CALLING SEQUENCE

sys link; name
FUNCTION  

**wdir**: write a directory entry into the current directory whose i-number is **ii**.

**ARGUMENTS**

**INPUTS**

**ii** - contains the current directories i-number

**OUTPUTS**

an entry in the current directory

**v.base** points to **wdir** but

**v.count = 10.

**pi** - contains the current directory's i-number

**CALLING SEQUENCE**

for ro, **wdir** - in syslink

follows mkdir directly
FUNCTION: Given the file descriptor (index to the u.fp list) "fclose" first gets the
l-number of the file via "getf." If the l-node is active (i-number ≠ 0) the entry in the u.fp list is cleared.
If all the processes that opened that file close it, then the tsp entry is freed and the file is closed. If not, a return is
posted. If the file has been deleted while open (see "deleted flag," F page 8) "anyi" is called to see if anyone
else has it open, i.e., see if it appears in another
entry in the tsp table. (See "anyi" for details. H.2, page 0)
Upon return from "anyi" a check is made to see if the
file is special.

INPUTS
R1 - contains the file descriptor (value = 0, 1, 2, … 9)

v.fp - list of entries in the tsp table

fsp - table of entries (4 words/entry) of open files See F page 8

OUTPUTS
R1 - contains the same file descriptor it entered with

if all processes that open file close it, the tsp entry is freed and the
file is closed

if "anyi" is called the outputs in "anyi" occur (H.2, p. 0)
- the "number of processes" byte in the tsp entry is decremented
  (See F page 8
- R2 contains l-number

ARGUMENTS —

CALLING SEQUENCE for R0, fclose
FUNCTION - see 'error' routine
CALLING SEQUENCE - 
ARGUMENTS - 
INPUTS - 
OUTPUTS -
ID U2-3 "isdir"

FUNCTION: "isdir" checks to see if the i-node whose i-number is in R1, is a directory. If it is an error occurs, because "isdir" is called by syslink and sysunlink to make sure directories are not linked. If the user is the super user (u uid = 0), "isdir" doesn't bother checking. The current i-node is not disturbed.

CALLING SEQUENCE:
JSR R0, isdir

ARGUMENTS:
R1 - contains the i-number whose i-node is being checked
u uid - user i d
ii - current i-node number
i flgs - flag in i-node (This is tested to see if the i-node is a directory i-node)

OUTPUTS:
R1 - contains current i-number upon exit
current i-node back in core
FUNCTION: "isown" is given a file name. It finds the c-number of that file via "namei" then gets the i-node into core via "iget". It then tests to see if the user is the super user. If not, it checks to see if the user is the owner of the file. If he isn't an error occurs. If user is the owner "setmod" is called to indicate the i-node has been modified and the 2nd argument of the call is put in R2.

ARGUMENTS:

INPUTS arguments of syschmod or syschown calls

OUTPUTS

C01D - set to 1

linod - set to 1

R2 - contains second argument of the system call

CALLING SEQUENCE jsr r0, isown
maknoded

maknoded creates an i-node and makes a directory entry for this i-node in the current directory. It gets the mode of the i-node in R1, the name is used in mkdir for the directory entry. (See mkdir H.2). The i-node is made in the following manner. First the allocate flag is set in the mode. A scan of i-nodes above 40 begins. The i-node map is checked to see if that i-node is active. If it is the next i-node in the bit map is checked until a free one is found. If one is found a check is made to see if it is already allocated. If it is the search continues. If not the i-number is put in u.dir but and a directory entry is made via mkdir. Then the new i-node is fetched into core and its parameters are set (see outputs).

ARGUMENTS  

INPUTS  

R1 - contains mode
ii - current i-number - should be of the current directory
mg, R2 - bit position & byte address in i-node map

OUTPUTS  

u.dir but - contains i-number of free i-node
i. flags - flag in new i-node
i.uid - filled with void
i.uns - 1 is put in the number of links
i.climz - creation time
i.climz2 - modification time
i.mod - set via call to setmod

CALLING SEQUENCE for go, maknoded
FUNCTION "mkdir" makes a directory entry from the name pointed to by u.namep
in to the current directory.
It first clears the locations u.dirbut+2 - u.dirbut+10.
"mkdir" then moves a character at a time into u.dirbut+2 -
u.dirbut+10, checking each time to see if the character is a ".". If it
is an error occurs, because "/" should not appear in
a directory name.

A pointer to an empty directory slot is then put in u.off
the current directory (inode is brought into core and an
entry is written into the directory.

ARGUMENTS

INPUTS

R2, u.namep - points to a file name that is about to become
a directory entry

R3 - points to u.dirbut - locations

II - current directory's c-number

OUTPUTS

u.dirbut+2 - u.dirbut+10 - contains file name

u.off - points to entry to be filled in the current directory

u.base points to start of u.dirbut

R1 - contains c-number of current directory

See w.dir for others
FUNCTION:

getf first checks to see that the user has not exceeded the maximum number of open files (16). If he has an error occurs. If not, the index into the fsp table is calculated from the u.ftplist. u.ftofp contains the address of the 3rd word in that fsp entry. cdev and ri contains the device and i-number of the file.

ARGUMENTS

INPUTS

OUTPUTS

RI - contains index into u.ftplist

u.ftofp - contains address of 3rd word in that fsp entry

cdev - contains files device number

RI - contains files i-number

CALLING SEQUENCE: jsr 80, getf
FUNCTION: seektell

seektell puts the arguments from a sys seek and
systell call in u.base and u.count. It then gets the
1-number of the file from the file descriptor in u.ro
and by calling getfil. The 1-node is brought into core
and then u.count is checked to see if it is a 0,1022.
If it is 0 - u.count stays the same
1 - u.count = offset (u.fofp)
2 - u.count = u.size size of file

ARGUMENTS

INPUTS

u.base - puts offset from sysseek or systell call
u.count - put ptrname " " " " " " (u.ro) - contains file descriptor (index to u.fp list
u.size - size of file in bytes
(u.fofp) - points to 3rd. word of fsp entry

OUTPUTS

- an 1-node in core via "get"
11 - 1-number of file in question
u.count - see function above

CALLING SEQUENCE: jsr ro, seektell
FUNCTION: "sysbreak" sets the programs break point. It checks the current break point (vbreak) to see if it is between "core" and the stack (sp). If it is, it is made an even address (if it was odd) and the area between vbreak and the stack is cleared. The new break point is then put in vbreak and control is passed to "sysret."

CALLING SEQUENCE: sysbreak ; addr

ARGUMENTS: addr - address of the new break point

INPUT: v.break - the current break point

OUTPUT: v.break - contains new break point
area between old vbreak and stack is cleared if vbreak is between "core" and the stack "sp"
FUNCTION namei takes a file path name (address of string in u.nameip) and searches the current directory or the root directory (if the first character in the string pointed to by u.nameip is a '/') and returns the i-number for the file in R1. namei operates in the following manner:

- A file may be referenced in one of two ways, either relative to the users directory or relative to the rootdir directory; in the second case the file path name must begin with the char '/'. Whenever a '/' is encountered in a path name, it indicates that the characters preceding it represent the path name of a directory and the file name following the '/' is stored in the directory.

- Directories contain 10 byte entries, the first 2 bytes contain an inumber, the last 8 bytes a file name associated with the i-number.

- namei scans the file path name until it reaches a '</' or a '<\0>', it reads the current directory until it finds a file name which matches the scanned portion of the file path name. When a match is found the i-number is taken from the matched directory entry. If namei has scanned to a '<\0>' then the i-number is that for the file specified by the file path name. If namei scanned to a '</' then the i-number is that of the next directory in the path. namei scans the file path name until it reaches a '</' or a '<\0>', etc. If no file is found return to nofile; otherwise normal.

ARGUMENTS -

INPUTS

- u.nameip (points to a file path name)
- u.edirm (i-number of users directory)
- u.cdev (device number on which user directory resides)
- R1 - contains the i-number of the current directory (u.edirm)

OUTPUTS

- R1 (i-number of file referenced by file path name)
- cdev
- R2, R3, R4 (internal)
- vidirp - points to the directory entry where a match occurs and search for file path name, if no match vidirp points to the end of the directory and

R1 - i-number of the current directory

Calling sequence jsr R0, namei's nofile ; normal ;
ID 02-4  syschdir

**FUNCTION:**  syschdir makes the directory specified in its argument the current working directory.

**ARGUMENTS**
- name - address of the path name of a directory terminated by a 0 byte

**INPUTS**
- cflags - c-node flag
- cdev - contains c-number
- cdev - contains device number of c-node

**OUTPUTS**
- c - contains c-number
- u_cdir - c-number of user's current directory (same as c)
- u_cdev - device number of current directory

**CALLING SEQUENCE**  syschdir; name
FUNCTION

"syschmod" changes the mode of a file.
It calls "iswn" to get the file's c-node into core
and checks its owner against the user. A check
then then made to see if the file is a directory.
If it is, the "set user id" and "executable" modes
are cleared because a directory cannot be executed.
Then the remaining mode used in the lfltgs.
If the file is not a directory, lfltgs is set.

ARGUMENTS:

name - name of file whose mode will be changed
mode - see syschmod UNIX programmers manual

INPUTS

lfltgs - c-node flag
r2 - mode

OUTPUTS

lfltgs - contains new mode

CALLING SEQUENCE

syschmod name, mod
FUNCTION: "syschown" changes the owner of a file.
"isown" is called to get the c-node of the name of the file. Then syschown checks to see if the "set user id on execution" flag is on. If it is, an error occurs, because one could create false files able to misuse other files.
If not, the new owners id is put into the c-node.

ARGUMENTS
- name: file name
- owner: id of new owner

INPUTS
- u.uid: user id
- u.flgs: c-node flag

OUTPUTS
- u.uid: the c-node user id now contains id of new owner

CALLING SEQUENCE: syschown (name; owner)
FUNCTION

sysexec initiates execution of a file whose path name is pointed to by "name" in the sysexec call. sysexec performs the following operations:

1. obtains i-number of file to be executed via "namei".
2. obtains i-node of file to be executed via "iget".
3. sets trap vectors to system routines.
4. loads arguments to be passed to executing file into highest locations of user's core.
5. puts pointers to arguments in locations immediately following arguments.
6. save number of arguments in next location.
7. initializes user's stack area so that all registers will be zeroed and the PS cleared and the PC set to core when sysret restores registers and does an rti.
8. initializes u.ro and u.sp.
9. zeroes user's core down to u.ro.
10. reads in executable file from storage device into core starting at location "core".
11. sets u.break to point to end of user's code with data area appended.
12. calls "sysret" which returns control at location "core" via rti instruction.
The layout of core when sysexec calls sysret is:

```
user prog
    ...
    (u.break)
zeros
    ...
    (u.sp) = (sp)
    ...
    (u.ro); (u.sp)+10
    core
    ...
    core
```

**CALLING SEQUENCE**

```
sys exec ; namep ; argp
```

**ARGUMENTS**

- `namep` (points to file path name of file to be executed)
- `argp` (address of table of argument pointers)
- `argp1`, ..., `argpn` (table of argument pointers)
- `argp1 : <...\o>, argp2 : <...\o>, ..., argpn : <...\o>` (argument strings)
FUNCTION:
"clear" zero's out a block (whose block number is in Ri) on the current device. "clear" does this in the following manner:
1) 'wslot' is called, which obtains a free I/O buffer (see 'poke' H. 8 pages 5) via 'bufalloc'.

Bits 9:15 of the 1st word of the I/O queue entry are set to set up the buffer for writing.

2) The buffer is zeroed and written out on the current device for the block(indicated by R1) via 'askw'.

ARGUMENTS:

INPUTS
R1 - Contains block number of block to be zeroed
cdev - current_device_number
See inputs for: 'bufget', 'wslot', 'askw'.

OUTPUTS
a zeroed I/O buffer onto the current device
R5 - points to last entry in the I/O buffer
R3 has 0 in it. It counts from 256 - 0. It is used as a word counter in the block.

CALLING SEQUENCE: 

    JSR  50, clear
FUNCTION - clears core from arg1 to arg2.

CALLING SEQUENCE - jsr ro, copyz; arg1; arg2

ARGUMENTS - arg1 - address of lowest location in core to be cleared,
arg2 - address of highest location in core to be cleared,
arg1 < arg2

INPUTS - ro - return address for the routine calling copyz. It is used to access arg1, then arg2 and, finally, set to the actual return address of the calling routine.

OUTPUTS - ro - points to the next instruction to be executed in the calling routine.
ID - U3-3

**FUNCTION**

`idle` saves the present processor status word on the stack, then clears the processor status word. `clockp` is saved on the stack. It points to one of the clock cells in the superblock. `clockp` is then made to point to another set of clock cells specified as an argument in its call.

When an interrupt occurs, `clockp` and the processor status word are popped off the stack thus being reset to their values before the call took place.

**CALLING SEQUENCE**

`jsr r0, idle`

**ARGUMENTS**

`-5, wait + 2`

**INPUTS**

`ps - process status`

**OUTPUTS**

`clockp, clock pointer`

`ps - restored to original value`
FUNCTION

"putk" is called with a process number in R1 and a pointer to the lowest priority Q (rung + 4) in R2. A link is created from the last process on the queue to the process in R1 by putting the process number in R1 into the last processes link. (The last processes number slot in p_link). The process number in R1 is then put in the last process position on the queue.

If the last process on the queue was "L" and the process number in R1 was "N" then upon return from putk the following would have occurred:

```
   | n       | rung + 4 | n        |
   +----------+----------+----------+
   | previously held "L"    |
   +----------+----------+----------+
   | p_link + L - 1  |
   +----------+----------+----------+
```

ARGUMENTS

INPUTS

R1 user process number
R2 points to lowest priority Queue

OUTPUTS

R3 process number of last process on the queue upon entering putk
p.link - 1 + (R3) - process number in R1
rung + 5 - process number in R1
R2 points to lowest priority queue
FUNCTION: swap is the routine that controls the swapping of processes in and out of core. It works in the following manner:

1) The processor priority is set to 6.

2) The rung table is searched for the highest priority process. If none are found, idle is called to wait for an interrupt to put something on the queue. Upon returning after an interrupt, the queues are searched again.

3) The highest priority process number is put in R1. If it is the only process on that queue the queue entry is zeroed. If there are more processes on this queue the next one in line is put in the queue from plink. (See F page 9)

4) The processor priority is set to 0.

5) If the new process is the same as the process presently in core, nothing happens. If it isn't, the process presently in core is written out onto its corresponding disk block and the new process is read in. "wswap" writes out the old process. "rswap" reads in the new one. For more information see "wswap", "rswap", "upack", and pll of Implementation Manual.

6) The new process' stack pointer is restored. The address where the process left off before it was swapped out is put in R9. So when "rts RO" is executed the new process will continue where it left off.

INPUTS

- rung table - contains processes to be run. See F page 9
- plink - contains next process in line to be run. See F page 9
- pmono - process number of process in core
- stack - swap stack used as an internal stack for swapping

ARGUMENTS

- present process to its disk block
- new process into core
- uquant = 30 (Time quantum for a process)
- d.pri - points to highest priority run q
- R1 - points to the run queue
- R2 - contains new process number
- ps - processor status = 0
- RO - points to place in routine or process that called Swap

OUTPUTS

- all user parameters
FUNCTION

"tswap" is the time out swap. "Tswap is called
when a user times out. The user is put on the
low priority queue. This is done by making
a link from the last user on the low priority queue
to him via a call to "putlu." Then he is
swapped out.

CALLING SEQUENCE -

ARGUMENTS -

jsr 10, tswap

INPUTS u.uno - users process number
    rung + 4 - lowest priority queue

OUTPUTS R1 - users process number
   R2 lowest priority queue address
FUNCTION: "unpack" unpacks the user's stack after swapping and puts the stack in its normal place. Immediately after a process is swapped in its stack is next to the program break. "unpack" moves the stack to the end of core.

If v.break is less than "core" or greater than v.esp nothing happens. If v.break is in between these locations, the stack is moved from next to v.break to its normal location at the end of core.

CALLING SEQUENCE

JST R0, unpack

ARGUMENTS:

INPUTS: v.break: users break point (end of users program)

OUTPUTS - stack gets moved if proper conditions stated above are met.
FUNCTION: "rswap" reads a process, whose number is in R1, from disk into core. 2*(the process number) is used as an index into p.break and p.dska. The word count in the p.break table is put in the 3rd word of the swap I/O queue entry. The disk address in the p.dska table is put in the second word. The first word of the swap I/O queue entry is set up to read (bit 10 set to a 1) and "p poke" is called to read the process into core.

ARGUMENTS:

INPUTS
R1 - contains process number of process to be read in
p.break - table containing the negative of the word count for the process
p.dska - table containing the disk address of the process
v.empt - determines handling of empty
v.illegal - determines handling of illegal instructions

OUTPUTS
10 = (v.illegal)
30 = (v.empt)

swp - bit 10 is set to indicate a read (bit 15 = 0 when reading is done)
swp+2 - disk block address
swp+4 - negitive word count

CALLING SEQUENCE: JR R0, RSWAP
FUNCTION: "wswap" writes out the process that is in core onto its appropriate disk area. The process stack area is copied down to the top of the program area to speed up I/O. The word count is calculated and put in "swp+4." The disk address (block number) is put in "swp+2." "swp" is setup to write by setting bit 9 and "ppoke" is called to initiate the writing. The area from $USER to the end of the stack is written out. The I/O queue entry "swp" looks like Fig 1 below just before the process is written out by p poke.

<table>
<thead>
<tr>
<th>bit 9 among others is set</th>
<th>swp</th>
<th>swp+2</th>
<th>swp+4</th>
<th>swp+6</th>
</tr>
</thead>
<tbody>
<tr>
<td>disk block address</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>neg. word count</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>user (address to start)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

ARGUMENTS

- $break - points to end of program
- $wsp - stack pointer at beginning of swap
- core - beginning of process program
- ecore - end of core
- user - start of user parameter area
- $up - user process number
- $disk - holds block number of process

OUTPUTS

- $swp I/O queue (see above)
- $break - negative word count of process
- $l - processes disk address
- $r2 - negative word count
FUNCTION: "clock" handles the interrupt for the 60 cycle clock. It increments the time of day, increments the appropriate time category and decrements the users time quantum. It then searches through the touts table and does the following:

1) If the processor priority is high (24) and the time in the touts entry is not out (#2), the time in the entry is decremented. If it turns 0 when decremented, it is incremented so that it will turn 0 next time when the priority might be low (see 2 below).

2) If the processor priority is low and (1) the user is not timed out or (2) we are presently inside the system and a touts entry gets decremented to 0, the corresponding routine in the touts table is called. If the touts entry was 0 before decrementing, nothing happens. If the user is timed out (and we are outside the system), the users RD is restored to him and "sys_rele" is called to swap him out and bring in another process.

CALLING SEQUENCE: interrupt vector

ARGUMENTS:

INPUTS:
- lks - clock status register
- stime+2 - time of day
- clock - points to one of the clock cells in the super block
- ugrad - users time quantum
- systla - system touts table
- touts - table of entry points of substrates

OUTPUTS:
- stime+2 - incremented
- clock - incremented
- ugrad - decremented
- touts - entries decremented
- RO - contains users RD if conditions of (2) above are met
ID 04-4  isintr

FUNCTION
"isintr" checks to see if an interrupt or quit from a tty belongs to the current user. If so, it won't skip on return; if not it will skip. When the interrupt does belong the output list in clist is erased via calls to getc. This prevents output coming out on the current tty key is hit.

Case I
Nothing happens except the return is skipped when
1) tty, tty buffer pointer, etc. buffer = 0
2) interrupt character in buffer = 0
3) char = "FS" and q.quit = 0
4) no tty block is found that matches tty

Case II
The return is not skipped and the output gets flushed if
1) interrupt character = "FS", q.quit = 0 and the tty block in control is found
2) interrupt character = "delete," and qintr = 0 and the tty block in control is found.

CALLING SEQUENCE
just go, isintr

ARGUMENTS

INPUTS
u.quitp - pointer to buffer of tty in control of the current process
q.quit - determines handling of interrupts if 0 - nothing happens

OUTPUTS
Case I - nothing except return is skipped
Case II - Processor priority = 5
getc erases the output character list
ID - U4; 3

PPTI - PAPER TAPE INPUT INTERRUPT ROUTINE

FUNCTION - PPTI does one of following depending on value of "PPTIFLG".

1. If "PPTIFLG" indicates file not open (=0), nothing is done.

2. If "PPTIFLG" indicates file just opened (=2), a check is made to determine if
   the error bit in PRS is set. If it is "PPTITO" is called to place 10 in
   the Tout entry for PPT input. If the error bit is not set, "PPTIFLG" is
   changed to indicate "NORMAL" operation (set to 4). A new "WAKEUP" is
   called to wake up process identified in WLIST for PPT input. Also, the
   character in the PRB buffer is placed in CLIST if there is room. If there
   is no room, the character is lost. Finally, a check is made to determine
   if the character count in the PPT input area of CLIST has 500
   characters. If it does, the reader enable bit is set.

3. If "PPTIFLG" indicates file normal (=4), the process in the PPT input entry of
   WLIST will be woken up (via WAKEUP). A check is then made to determine
   if the error bit in PRS is set. If it is, the "PPTIFLG" is set equal to 6.
   If it is not the contents of PRB are placed in the CLIST with "PUTC". If
   CLIST full, the character is lost. In addition, if the character
   count for PPT input in the CLIST is less than 50, the reader enable
   bit is set.

4. If "PPTIFLG" indicates the file not closed (=6), this is an indication that
   the error bit was set when PPTIFLG equalled four and therefore normal is
   done.

CALLING SEQUENCE - PPTI is the PAPER TAPE INPUT INTERRUPT ROUTINE

INPUTS - PPTIFLG - Flag which indicates function to be performed
PRS - Paper tape read status bits
CC+2 - Character count for PPT input in CLIST
PRB - Input character

OUTPUT - PPTIFLG - (see above)
ID_UH4 PPTITO - PAPER TAPE INPUT ROUTINE

**Function**: If "PPTIFLG" indicates the file has just been opened, PPTITO

1. Places 10 in the TOU TT entry for PPT input

2. Checks error bit in PRS and sets reader enable bit if error bit not set.

   For all other values of "PPTIFLG", PPTITO does nothing.

**Calling sequence**: JSR 100, PPTITO

**Inputs**: PPTIFLG - Value of this parameter indicates to PPTITO the function it is to perform

PRS - Status of PPT reader

**Outputs**: TOU TT+1 - Contains the count for PPT input.

PRS - Reader enable bit
ID-U4,3 PPTO - PAPER TAPE OUTPUT INTERRUPT ROUTINE

FUNCTION: CALLS STARPT TO OUTPUT NEXT CHARACTER IN CLIST PPT OUTPUT

CALLING SEQUENCE: INTERRUPT ROUTINE

INPUTS: SEE INPUTS FOR "STARPT"

OUTPUTS: SEE OUTPUTS FOR "STARPT"
FUNCTION \texttt{retisp} pops the stack and restores the values of \texttt{RO}, \texttt{RI}, \texttt{R2}, \texttt{R3} and \texttt{CLOCKP} to what they were before the interrupt occurred. \texttt{retisp} then executes \texttt{RTI} and returns.

CALLING SEQUENCE
\begin{verbatim}
   JMP retisp
\end{verbatim}

ARGUMENTS
- 

INPUTS
- 

OUTPUTS
\texttt{RO}, \texttt{RI}, \texttt{R2}, \texttt{R3}, \texttt{CLOCKP}
FUNCTION: Sleep puts the process whose process number is in u.uno on the wait list (wlist) and swaps it out of core. It works in the following way:

1) A wait channel number is given as an argument to sleep. The process number occupying that channel is saved on the stack. The process number that is getting put to sleep (u.uno) is put in that wait channel.

2) A call is made to "isintr" to see if that user has any interrupts or gofs. If he does a return to him via "sysret" is made. If he doesn't swap is called to swap out the process so it can sleep.

3) A check is made on the new user (the one who got swapped in) to see if he has any interrupts or gofs. If not, a link is created of the old process number that first occupied the wait channel by a call to "putlru" a normal return is then made.

CALLING SEQUENCE: jsr 50, sleep

ARGUMENTS arg - wait channel number

INPUTS u.uno - process number that gets put to sleep
wlist - wait channel list
rung+1 - lowest priority run

OUTPUTS sleeping process number onto disk

ID - U4;5    STARPPT

FUNCTION - "STARPPT" CHECKS THE CHARACTER COUNT FOR PPT OUTPUT IN THE CLIST. IF IT IS ≤10, "STARPPT" USES "WAKEUP" TO WAKEUP PROCESS IDENTIFIED IN "WLIST" ENTRY FOR PPT OUTPUT. "STARPPT" THEN CHECKS THE READY BIT IN THE PUNCH STATUS WORD. IF IT IS SET, "STARPPT" USES GETC TO FETCH THE NEXT CHARACTER IN THE CLIST AND THEN PLACES IT IN PRB.

CALLING SEQUENCE - JSR 100, STARPPT

INPUTS - CC+3 - CHARACTER COUNT FOR PPT OUTPUT IN CLIST
PPS - CONTAINS READY BIT

OUTPUTS - SEE OUTPUTS FOR "GETC" AND "WAKEUP"
PPB - PPT OUTPUT BUFFER
FUNCTION: "setisp" stores $r1, r2, r3 and clockp on the stack
puts $s, syst+2 in clockp and returns via a jump
without popping the stack

CALLING SEQUENCE   \texttt{jsr \ Ro, setisp}

ARGUMENTS
INPUTS \ $r1, r2, r3, clockp are saved on the stack
OUTPUTS \ clockp points to $s, syst+2
FUNCTION

startty prepares the system to output a character on the console tty.

It performs the following operations:

1. **S**OME **E**ELEING **W**ITH **W**AKEUP?
2. tests console output status register ready bit, if bit is clear, return.
3. if bit is set, check timeout byte for console (toutt); if non-zero, return.
4. if toutt is zero, put char to be output in r1
5. load character in console data buffer register
6. if char = LF, make next char to be output a CR or CR
7. if char = HT, set time out to 15 clock cycles

ARGUMENTS

**INPUTS**  tty, ch (character to be output), toutt

**OUTPUTS**  tpb (loads a character in tty output data buffer register), r1 (character output), toutt.

**CALLING SEQUENCE**  

    .je r10, startty
FUNCTION ttyo

ttyo is the console typewriter output interrupt routine. It calls setup to save registers during the interrupt then calls startty to put the character in the tty output buffer and then restores the registers and returns from the interrupt.

CALLING SEQUENCE interrupt routine called via trap

ARGUMENTS -

INPUTS character in ttyo.ch

OUTPUTS see startty
FUNCTION

"wakeall" wakes up all the processes on the wait list by making consecutive calls to wakeup going through all the wait channels. The processes are linked together on the lowest priority queue (runq + 4), used to identify the world when a quit or interrupt happens. (radio, typewriter)

CALLING SEQUENCE

for i, wakeall

ARGUMENTS

INPUTS

OUTPUTS

all sleeping processes are put on the lowest priority queue
**FUNCTION:**

TTYI - console tty interrupt handler (process priority = 5)

TTYI puts a character from the tty reader buffer (lkb) in RI

sets the enable bit of the tty status register, and

strips the character to 7 bits. Depending on what the

character is the following things may occur:

1) If the character is a letter (A-Z)

   console tty input list in

   It is changed to lower case and put on the list via "pute". It is then put on the tty output buffer

   via "startty" ( Thus echoed ). If the number of characters on that

   list (cc) exceeds 15 a call to "wakeup" is made
to clear that list. If less than 0 nothing else happens

2) If the character is a "3" or a "del"

   If also, the console tty blocks buffer pointer is zero

   wakeall is called (and all sleeping processes are put on the low

priority queue). If the console tty blocks buffer pointer to the char (3 or del) is

   put in the 7th byte of the buffer and wakeall is

   called.

3) If the char is an "07" or "NL"

   cc is not checked and wakeup is called

**CALLING SEQUENCE**

**ARGUMENTS**

**INPUTS**

lkb   tty reader buffer

tks   tty status register

cc    number of characters on the character list

**OUTPUTS**

RI is used to contain the character

TTYCH - has the character

sec  function for other outputs depending on what the character is

R2 - points to the console tty's buffer

(R2)+6 - the interrupt character byte is filled with the appropriate char if above conditions met
FUNCTION: wakeup is called with two arguments: arg1 is one of the run queues and arg2 is a wait channel number.

wakeup wakes the process sleeping in the specified wait channel by creating a link to it from the last user process on the run queue specified by arg1. This is done by a call to "putl".

If there is no process to wake up, (wait channel contains 0) nothing happens.

CALLING SEQUENCE: jsr 45n_wakeup; arg1, arg2

ARGUMENTS:

arg1 - points to one of the three run queues

arg2 - is the number of the wait channel of the process to be woken

INPUTS:

wlist  - wait channel table

u, pri - users process priority (expressed as a pointer to one of the run queues)

OUTPUTS:

if u, pri > arg1, then ugrant = 0

wlist (R2) = 0, i.e., entry in wait channel = 0

R2 is used to point to one of the run queues

R3 contains the number of the wait channel
ID: u633 access:

FUNCTION reads in section of core beginning at location "inode" the i-node for file with i-number n. Checks whether user is owner and other stuff to be described.

ARGUMENTS: argo (user, owner flagmask)

INPUTS: r1 (i-number of file), u.uid, i.uid

OUTPUTS: inode, r2 (internal)

CALLING SEQUENCE: jsr r0, access 3 argo
FUNCTION — Given a block number for a block structured I/O device, 'free' calculates the byte address and bit position of its associated bit in the free storage map of the in-core image of the superblock for the device (not fixed head disk or mountable device superblock). It then declares the specified block free by setting this bit. Then a flag is set to indicate that:

1. the superblock for the RF-fixed head disk has been modified \( (smod = smod + 1) \),

or

2. the superblock for a mountable device has been modified \( (mmod = mmod + 1) \).

CALLING SEQUENCE — \textit{jsr 10, free}

ARGUMENTS —

INPUTS — byte mask table : Mask for bit 1

\begin{center}
| 2 & 0 \\
| 10 & 2 \\
| 40 & 0 \\
| 200 & 100 |
\end{center}

\begin{center}
mask for bit 0

\begin{center}
| 4 & 6 \\
| 9 & 7 |
\end{center}

R1: block number for a block structured device
cdev: current device ; 0 = drum, non zero = mountable device

OUTPUTS — MOUNT-$SYSM + (R2)$ word in free storage map portion of the in-core image of the superblock for a mountable device. If the device is mountable, the appropriate bit is set to free the block. If the device is not mountable, the bit remains unchanged.

$SYSM + 2 + (R2)$ same as above, but for drum with the superblock for the fixed head disk.
ID_45; 2 free

Outputs: MMOD is incremented if the superblock for the mountable device was modified.
SYSTEM is incremented if the superblock for the drum was modified.

R2 - saved on stack and restored on return.
R3 -
FUNCTION - "alloc" scans the free storage map of the super block of a specified device. When it finds a free block it saves the physical block number in r1; it then sets the corresponding bit in the free storage map and sets the super block modified byte (smod, mmod).

CALLING SEQUENCE — jsr ro, alloc

ARGUMENTS —

INPUTS — cdev (current device), r2, r3

OUTPUTS — r1 (physical block number of block assigned), smod, mmod, systm (drum superblock), mount (dismountable super block), r2 (internal), r3 (internal), stack (values of r2, r3 on input)
calc calculates the physical block number from the i-number of an i-node. It then reads in that block and calculates the offset in the block for the i-node with the particular i-number, then depending on whether the argument in the calc call is a 0 or a 1, it reads the inode in the data buffer in core starting at location "inode"(argument =0) or it will take the inode information currently stored at location "inode" and write it out on the device. (argument =1)

The physical block number for an inode is calculated as follows:

let \( N = \text{i-number} \), \( PBN = \text{physical block number} \), \( BO = \text{byte offset} \)

then \( PBN = \frac{N + 31}{16} \)

and \( BO = 32 \times ((N + 31) \mod 16) \) (see secF for general discussion of inodes)

ARGUMENTS
\begin{align*}
\text{arg} = 0 & \text{ read inode} \\
\text{arg} = 1 & \text{ write inode}
\end{align*}

INPUTS
\begin{align*}
\text{inode} : i1 (\text{i-number})
\end{align*}

OUTPUTS
\begin{align*}
\text{inode} : i1 (\text{internal}), i5 (\text{internal}), i3 (\text{internal})
\end{align*}

CALLING SEQUENCE
\begin{align*}
\text{jsr } n0, \text{calc ; arg}
\end{align*}
FUNCTION - "iget" get a new i-node whose i-number is in ri and whose device is in cdev. If the new i-number and its device are the same as the current i-number and its device (ri=ii and cdev=idev) no action is taken. If they do not agree, "iget" checks to see if the current i-node has been changed (iMOD=1). If it has been changed, the current inode is written out to its device. Then if the current device is the drum, the new i-node i-number is checked to see if it is the i-number of the cross device file, if it is the current device becomes the mounted device and the i-number is set to 41. (thus the root directory for the mounted device is referenced). Then the new inode is read into the "inode" block in core via "icale".

ARGUMENTS

INPUTS

ii (current i-number) , rootdir
cdev (new i-node device)
idev (current i-node device)
imod (current i-node modified flag)
mnti (cross device file i-number)
ri (i-number of new i-node)
mntd (mountable device number)

OUTPUTS
cdev, idev, imod, ii, ri

CALLING SEQUENCE
dir ro, iget
FUNCTION: "itrun" gets and cnodes via cget. It increments through the c.dskp (list of contents or indirect blocks in the cnodes) table and frees the blocks specified there. If the file is small, the block numbers in the c.dskp list are freed. If the file is large, c.dskp contains pointers to indirect blocks. The block numbers in these indirect blocks are then freed and the indirect blocks are freed.

ARGUMENTS

INPUTS

r1 - contains J-number for use by "cget."
c.dskp - pointer to "contents or indirect blocks" in cnode
c.flags - contains flag for large file. SEE SEC F PAGES 5
c.size - size of file

OUTPUTS:
c.flags - "large file" flag is cleared
c.size - set to 0.
c.dskp - i.dskp+16 - the entire list is cleared
setlmod - set to indicate c-node has been modified
r1 - contains J-number on return from this subr.
r3 - used in subroutine

Calling Sequence

jsr ro, itrun
FUNCTION

imap

imap finds the byte in core containing the allocation bit for an i-node whose number is in RI. This core area is a copy of the super block and happens to be the i-node map. The byte address is calculated as follows:

\[ \text{byte address} = \text{addr of start of map} + \frac{(i\text{-number} - 41)}{8} \]

The bit position = \((i\text{-number} - 41) \mod 8\)

ARGUMENTS

INPUTS

R1 - contains i-number of i-node in question

R2 - has byte address of byte with the allocation bit

OUTPUTS

R3 - has a mask to locate the bit position.

R4 - is in the calculated bit position

R5 - used internally

CALLING SEQUENCE

J5 R5 for imap
FUNCTION - sets byte at location "imod" to a 1, thus indicating that the i-node has been modified. Also puts the time of modification into the i-node.

CALLING SEQUENCE - jsr ro, setimod

ARGUMENTS -

INPUTS - s.time, s.time+2 (current time)

OUTPUTS - imod, i.mtim, i.mtim+2
FUNCTION: "dskr" gets an inode into core via "iget"

It then sets u.count according to the following rules:
If the number of bytes left to read in a file is
greater than the number of bytes he wants to read
u.count is unchanged.
If the number of bytes left to read in the file is less than u.count, u.count gets
set to that number.
If the user offset u.offset is greater than the file length,
there is nothing left to read so dskr returns.
Once u.count is established a block address for the
file is calculated via iget, the file is read into
system buffers and the data is transferred to
user buffers in core. If u.count is not to
the process is repeated until u.count is 0. Processor
status is then cleared.

ARGUMENTS

INPUTS:  R1 - contains T-number
1. size - file size in bytes
u.count - byte count desired
u.offset - offset in file, telling how many bytes have been read

OUTPUTS:  R2 - internal register
PS = 0
R3 - internal register

CALLING SEQUENCE:  JMP dskr

ISSUE 0  DATE 3/13/72  ID UNIX  SEC 4.6  PAGE 8
FUNCTION - 'dskw' writes user specified data into a file on the drum, as follows:

dskw' obtains an i-node number from the stack. If the i-node currently residing in the i-node area of core has been modified, this i-node is written out onto the drum in its appropriate position in the i-list. In any event, the i-node specified in the stack by the caller is read into the i-node area of core. A file is composed of blocks. The caller can modify several blocks in several passes thru a single call to dskw'. The no. of the block to be modified next is calculated by dskw' from the file offset (relative to the start of the file in bytes) specified by the caller in (u/oftp). The caller specifies the number of bytes to be modified in u/count. If the number of bytes the user specifies plus the offset into the file is greater than the present size of the file in bytes, i.size, then the size of the file is increased to incorporate the data overflow by changing the file size field in the i-node for the file (which is currently in the i-node area of core). The time that this file size change occurs is also inserted into the i-node and the i-node modification flag, (smod) is set. Dskw' then uses (u/oftp) to calculate an offset (relative to the start of the block) which specifies the 1st location within the block at which the callers data is to be written. Note that the offset determines the minimum number of bytes of user data that can be written on the file during this pass thru dskw'. If the number of data bytes the caller specifies is less than a block, the block is read from drum into a system buffer, then the appropriate bytes are overwritten. If the number of data bytes is less than a block, but exceeds 512, the offset, only 512 bytes are overwritten. Succeeding passes thru 'dskw' are necessary to write out the rest of the data. After each pass, the modified file block (in the system buffer) is written out on drum. When all required blocks are written, counters and pointers are returned to the caller.

CALLING SEQUENCE - jsr ro, dskw

ARGUMENTS -

INPUTS - sp - i-node number (u/oftp) - file offset
u/count - no. of bytes of data the caller desires to write
i.size - size (in bytes) of file to be altered (This parameter appears in the i-node whose no. is in sp)

see inputs for 'iget', 'setimod', 'inget', 'dskrd', 'wslot', 'siorg'

f1 - pointer to callers data area
(l1), (r1), ... , (r1) + [u/count - 1] - the callers data

Drum file
FUNCTION: epass gets the next character from the user into RI. A non-local return takes place (to the caller of "write") when the user's count (u.count) becomes zero.

ARGUMENTS -

INPUTS
- u.count - user's character count
- u.base - points to user's character buffer

OUTPUTS -
- if u.count > 0
  - u.count gets decremented
  - RI contains the next character
- unread gets incremented
- u.base - gets incremented -6 points to next character
- if u.count = 0
  - RO = return address to program that called "write"
  - RI = i-number of file under consideration

CALLING SEQUENCE: JSR RO, epass
FUNCTION: 'pasc' moves a byte of information specified in the lower half of $R1$ to
the byte address specified by $u$.base. It then increments $u$.base to
point to the next byte address, increments $u$.unread, the number of bytes passed;
and decrements $u$.count, the number of bytes yet to be moved. If there
are no more bytes to be moved, a non-local return to the caller of 'read'
(through which control was eventually passed to 'pasc') is taken. The current
$u$.number is popped off the stack into $R1$. If there are more bytes
to be transferred, the processor status is cleared and control is returned
to the caller.

CALLING SEQUENCE: 
\texttt{\textbf{J}Sr \textbf{R}0, pasc} 

ARGUMENTS:

INPUTS: $R1$ - contains a data byte in the lower half
$u$.base - contains a pointer to the user area of core to which the data byte
is to be transferred.
$u$.unread - the no. of bytes transferred
$u$.count - the no. of bytes to be read
$sp$ - the non-local return address
$sp+2$ - the value of $R1$ prior to calling 'pasc'

OUTPUTS: $(u$.base) - $0, \ldots, (u$.base) - $[u$.count - 1] contain the transferred information
$u$.base - points to the last byte transferred
$u$.unread - contains the number of bytes transferred + original value of $u$.unread
$u$.count - contains the number of bytes that still must be read
$sp$ - if non-local return popped twice
$ps$ - cleared
FUNCTION    see 'error' routine
CALLING SEQUENCE
ARGUMENTS
INPUTS
OUTPUTS
IO - ubju dskw

outputs - i: size - file size (may have been modified by dskw)
See outputs for 'i-get', 'setiMOD', 'mget', 'dskrd', 'uslot', 'sioreg'
R1 - points to the location succeeding the last caller data byte transferred
R2 - points to the location (in the system buffer) succeeding the last system buffer
byte over written.
R3 -
U:count - 0
Modified drum file
FUNCTION: 'ret' is a special subroutine return, used by the following subroutines:
1. ret
2. rept
3. dskr
4. passc
5. dskw
6. bread
7. bwrite
8. revt

In place of the standard return. In addition to performing standard return functions, 'ret' pops the stack and puts its value in R1. It also clears the program status word. 'ret' can be used simply to clear the program status word by entering via its 2nd entry point.

ARGUMENTS:

CALLING SEQUENCE: control should be passed to this routine by either a conditional or non-conditional transfer, to 'ret!' (the 1st entry point), or to '1', the secondary entry point.

INPUTS:
A. for primary entry: (SP)
B. for secondary entry: __________

OUTPUTS:
A. for primary entry: R1, PS
B. for secondary entry: PS
RPPT - READ PAPER TAPE

Function: RPPT uses "PPTIC" to get a character in PPT input section of CLIST and to set reader enable bit in PRS. If the PPT input section is empty and PPTIFLG = 6 (indication that the error bit was set during "NORMAL OPERATION") return is made to "RPPT" to instruction "OR RET" which eventually causes a return to the caller of "READ". If a character is available in CLIST, return is made to "RPPT" at "JSE, NO, PASSC".

Upon return from "PPTIC", "RPPT" uses "PASSC" to place the character fetched by "PPTIC" into the user's buffer area, if the number of characters that were specified by the user to be read in has been read in, return from "PASSC" is made to the caller of "READ".

It is appropriate at this point to describe how all the PPTA routines and subroutines are tied together to READ PPT. First of all the PPT file must be open. To do this a "SYS OPEN" is FOR READING OF THE PPT FILE IS USED. THIS ROUTINE OPENS THE FILE VIA "IDOPEN" WHICH SETS THE "PPTIFLG" INDICATING FILE OPEN. It also sets the "READER Interrupt Enable" bit in the PRS and empties the PPT input portion of CLIST.

Once the file is open, a "SYS READ" of the PPT file is made. A pointer to the location where the characters are to be placed along with the number of characters to be read are passed as arguments to "SYS READ". "SYS READ" then uses "JMP" to set "VBASE" to the location where the characters are to be placed. READ 1 is then called which jumps to "RPPT" which is described above. It should be noted that when "PPTIC" is called to obtain a character from CLIST, the process will be put to sleep if NO characters are in CLIST (with PPTIFLG = 6) and all characters to be read in have not been read. Also, the reader enable bit is set. Upon completion of the input of the next character (READ 1 is set) the PPT input interrupt routine (PPTI) is started which USES "WAKEUP" to wake up the process previously put to sleep.

Calling Sequence: JMP RPPT
Inputs: SEE INPUTS FOR "PPTIC", "PASSC"
Outputs: SEE OUTPUTS "PPTIC" AND "PASSC"
FUNCTION - Essentially, 'rtty' transfers characters from the console tty buffer, into a
user area of core, starting at byte address (u.base). If there are no
characters in the console tty buffer, 'rtty' calls 'canon', which gets
a line (120 characters) from the console tty clist and puts it in the
console tty buffer. The caller specifies the number of characters to
be transferred in u.count. If the number specified is greater than
the number actually in the console tty buffer, a synthetic return
is taken to the caller, after the characters in the buffer have been
transferred. If the number specified is less than or equal to the number
actually in the console tty buffer, a non-localized return to the caller
of 'readl' (which is the routine via which control was actually transferred
to 'rtty') is made when all the characters have been transferred
to the users core area (via pass0).

CALLING SEQUENCE - [conditional or unconditional branch, or jmp] rtty

ARGUMENTS -

INPUTS - tty + 70. - contains pointer to the header of the console tty buffer

2(tty+70.) - 2nd word of console tty buffer header; contains a count of
characters in the buffer

4(tty+70.) - contains a pointer to the next character in the buffer; 4
pointer values can include (tty+70.)+?, (tty+70.)+?, ..., (tty+70.)+?

see inputs for 'canon', 'passo', 'retl'

OUTPUTS - R1, RS used internally by 'rtty', original values destroyed.

RS - points to header of console tty buffer

see outputs for 'canon', 'passo', 'retl'
FUNCTION: 
"readi" reads from and i-node whose number is in RI. If the file in i-node is special a transfer is made to the appropriate routine. If not "dskr" is called and the file is read into user core. See "dskr" for details.

ARGUMENTS

INPUTS
RI - contains and i-number
u.count - byte count user desires
u.base - pts to user buffer
u.toff - pts to word with current file offset

OUTPUTS
u.nread - accumulates total bytes passed back

CALLING SEQUENCE: jsr $00, readi
WPPT—WRITE PAPER TAPE

Function—WPPT uses "CPASS" to get a character from the user's buffer area and "PPTOC" to output the character on the punch.

It is appropriate at this point to describe how all the PPT output routines and subroutines are tied together to output data on the PPT punch. First, the PPT file must be open. This is done via a "SYS OPEN" for writing. This places entries in the FSP table and the user's FP area.

Once the file is open a "SYS WRITE" of the PPT file is made. A pointer to the location where the characters are stored along with the number of characters to be punched are passed as arguments to "SYS WRITE". Then uses "RUI" to set "U.COUNT" equal to the number of characters to be punched and "U.CASE" equal to the location of the characters. "WRITE I" is then called which jumps to "WPPT".

"WPPT" as mentioned above uses "CPASS" to get a character from the user's buffer area. If the number of characters as specified in "SYS WRITE" call has been read, control is passed back to "SYS WRITE". If not, "PPTOC" is called. "PPTOC" first checks to see if character count for PPT output in the CLIST is 250. If it is then the process is put to sleep. If it isn't the character is placed in the CLIST and "STARPPT" is called.

"STARPPT" uses "GETRI" to get a character from CLIST and inserts it into the PPB. If the ready bit is set, if it isn't, control is passed back to "PPTOC".

Upon completion of output of the character in PPB (READY BIT SET) THE PAPER TAPE OUTPUT INTERRUPT ROUTINE (PPTO) IS STARTED VIA AN INTERRUPT. This routine (CALLS "STARPPT") which performs the following function ON AN INTERRUPT IN ADDITION TO THOSE DESCRIBED IN THE PREVIOUS PARAGRAPH. IT CHECKS TO SEE IF THE CHARACTER COUNT FOR PPT OUTPUT IS 250. IF IT IS IT WILL WAKE UP THE PROCESS IN THE CLIST ENTRY FOR PPT OUTPUT.

As seen from above a process puts itself to sleep when it has 250 characters in CLIST and is "woken up" by the paper tape output interrupt routine (PPTO) when the count becomes < 10.

Calling sequence—JMP PPT

Inputs—See inputs for "CPASS" and "PPTOC"

Outputs—See outputs for "CPASS" and "PPTOC"
Function 1. Calculates the first byte location (in the I/O buffer assigned to the caller) into which the caller's data is to be written.
2. Calculates the number of user data bytes to be transferred into this I/O buffer.
3. Performs bookkeeping functions, supplying the caller with information pertinent to the data transfer.

Calling sequence: jsr ro, sio reg

Arguments:

Io.fsp - specifies the byte in a file (relative to the start of the file) at which the user wants to start writing data.
Rs - address of data area of I/O buffer assigned to the user
u.base - address of 1st byte of user data
u.count - number of bytes of data to be transferred from user data area to I/O buffer
unread - number of bytes of data written out on the file for this user previously

Outs - (u.fsp) - specifies the byte immediately following the last byte of the file area in which the u.count bytes of user data is to be written
R1 - address of 1st byte of user data
u.base - specifies the byte immediately following the last byte of user data to be transferred to the I/O buffer
u.count - specifies the number of bytes of user data left to be transferred after the preceding set is transferred
unread - updated to include the count of bytes to be transferred by this
R2 - specifies the byte in the I/O buffer assigned to the caller at which the transfer of users' data is to start.
R3 - number of bytes of user data to be transferred to users I/O buffer
**FUNCTION**  
`writei` checks to see if there is any data to be written (on any device).  
If not, it does nothing more than return to the routine which called it.  
If there is data to be written, `writei` saves the i-node number of the file  
to be written on the stack, so it can be used by the appropriate output  
routine. Then `writei` checks to see if the output is to a special file (those  
files associated with i-nodes 1, ... 40) or to a non-special file. Writes for non-  
special files are routed to the `dskw` routine. Writes for special files are  
routed to appropriate routines, as follows:

<table>
<thead>
<tr>
<th>Special File</th>
<th>Write Routine</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASR-33 : console tty</td>
<td><code>wtty</code></td>
</tr>
<tr>
<td>PC11 : paper tape punch</td>
<td><code>wppt</code></td>
</tr>
<tr>
<td>core</td>
<td><code>wmem</code></td>
</tr>
<tr>
<td>RF11/RS11 : fixed head disk (drum)</td>
<td><code>wrfs</code></td>
</tr>
<tr>
<td>RR11/RM11 : moveable head disk</td>
<td><code>wrk1</code></td>
</tr>
<tr>
<td>711/711 : deck tape unit</td>
<td><code>wtap</code></td>
</tr>
<tr>
<td>711/711</td>
<td><code>xmtt</code></td>
</tr>
</tbody>
</table>

**CALLING SEQUENCE** - `jsr ro, writei`  

**ARGUMENTS**  

**INPUTS** -  
- `u, count` contains a count of the number of bytes to be written  
- `rl` contains the number of the i-node for the output file

**OUTPUTS** -  
- `A` to the calling routine if return is made to it by `writei`  
  - `u, unread` is cleared  
- `B` to the write routine for non-special files  
  - `u, unread` is cleared  
  - `(sp)` contains the i-node number
0: to the write routine for special files
unread = cleared
(SR) - contains the inode number
RI - contains the index into the special file routine jump table
Function — "WTTY" uses 'CROSS' to obtain the next character in the user buffer area. It then uses 'PUTC' to determine if there is an entry available in "FREELIST" portion of "CLIST". If there is, 'PUTC' places the character there and assigns the location to the console TTY portion of "CLIST". If there is no place available in the "FREELIST" portion of "CLIST", the process is put to "SLEEP". If there was a vacant location, "STARTTY" is used to attempt to output the character on the TTY. Upon return from "STARTTY", the next character is obtained from user buffer. If the buffer is empty, control is passed via "CROSS" back to "SYNWRITE". When the process is awakened by "WAKEUP", it again tries to find a location available in "FREELIST" and the character count for the console TTY < 20 so it can output the character.

Calling sequence — JMP, WTTY

Arguments — inputs:

CC+1 — contains character count for console TTY input.

See inputs for 'CROSS', 'PUTC', 'STARTTY', 'SLEEP'

Outputs —

R:1 (character from user buffer)

PS — processor priority set to 5

See outputs for 'CROSS', 'PUTC', 'STARTTY', 'SLEEP'
**FUNCTION**
canon handles the erase/kill processing on the typewriter (console tty).

It points to the start of the tty buffer. The argument following the call is where the characters are obtained. "canon" returns only when a full line has been gathered. (2) A new line has been received. (3) EOT (004) has been received. (4) 120 characters (the length of the buffer) have been received.

Cannon works in the following way:

1. The address of the start of the characters is put in buffer + 4 (4(RS))
2. buffer + 2 (2(RS)) is cleared. This is the character count.
3. A character is gotten off the queue. If it is a kill character (6), (100) a return to the beginning is made. Actually one starts over.
4. If the character is an erase (#), next char will overwrite the previous one and thereby erase it.
5. If the character is an EOT (004) the byte pointer is reset to the first character and a return is made.
6. If char is none of the above, it is put in the buffer when the character pointer tells it "go to 4(RS)"
7. The character count 2(RS) and the character pointer 4(RS) are then incremented.
8. If the char is a newline (\n) the char pointer is reset and a return is made.
9. If the buffer is full (byte count > 120) the char pointer is reset and a return is made.
10. If the buffer isn't full, the next character off the queue is put through the above tests.

Note: Canon should only be called when the number of already treated characters is zero, i.e., when the char count = 0; 2(RS) = 0. If the char count is > 0, the character pointer 4(RS) points to the first character not yet picked up.
CALLING SEQUENCE

ARGUMENTS

ARGUMENT arg - where characters are to be obtained from

INPUTS

RS points to tty buffer address

10(RS) start of character buffer

2(RS) character count

4(RS) points to next character position in data area

OUTPUTS

a full buffer, or a full line

R1 pointers to buffer+10

4(RS) character pointer reset to start of data area buffer+10

tty buffer

<table>
<thead>
<tr>
<th>number of char in buffer</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>char pointer (buffer+10)</td>
<td>+4</td>
</tr>
<tr>
<td>to start</td>
<td>+6</td>
</tr>
<tr>
<td></td>
<td>+8</td>
</tr>
<tr>
<td></td>
<td>+10</td>
</tr>
</tbody>
</table>

character storage area
FUNCTION: "cesc" is called by canon to check for an erase (#) or kill (@) character. R1 contains the character being tested. If the character is not an erase or kill the return is skipped. If the char is an erase or kill the character count and character pointer are decremented. If the previous character was a "\" the # or @ are taken literally and the the return is not skipped.

CALLING SEQUENCE: isr ro cesc; arg

ARGUMENT: arg 100 - @ means kill the last
           43 - # means erase last character

INPUTS: R1 - character to be tested
         2(R5) - character count
         (RS)+ - contains address of previous character

OUTPUTS: skip return if test character not erase or kill
         if character was erase or kill
         2(R5) - character count gets decremented
         4(R5) - character pointer ""
ID - 07; 7

CPPT - CLOSE PAPER TAPE FILE

FUNCTION - CPPT ASSIGN ALL PPT INPUT LOCATIONS IN CLIST TO FREE LIST, AND SETS "PPTFLG" TO INDICATE FILE CLOSED (=0)

CALLING SEQUENCE - JMP CPPT

INPUTS - NONE

OUTPUTS - SEE OUTPUTS FOR "GETC"

PS - PROCESSOR PRIORITY SET TO 5

PPTFLG - SET TO "0" TO INDICATE FILE CLOSED.
FUNCTION: ctty

ctty closes the console tty. All it does is decrement the number of processes that have opened the console tty file. The first byte of the console tty buffer is the number of processes that have opened this tty file.

See F page 11. A return is made via "reset."

CALLING SEQUENCE

JMP table in i-close

ARGUMENTS

INPUTS:

OUTPUTS

RS - points to console tty's buffer
(R5) - first byte of buffer gets decremented
ID: 4718
ERROR

FUNCTION: see 'error' routine

CALLING SEQUENCE:

ARGUMENTS:

INPUTS:

OUTPUTS:
FUNCTION  ttych gets characters from the queue of characters inputted to the console tty. If there are none, sleep is called. ttych works in the following manner:
   1) the processor priority is set to 5
   2) a character is gotten off the queue via "getc"
      if the list is empty, sleep is called
   3) if not the process status is cleared and a return is made

CALLING SEQUENCE  isr  ro  &*(ro)
                  in the call to "canon"

ttych was an argument

ARGUMENTS
  *ro

INPUTS
  PS = 0

OUTPUTS
  R1 - Character on top of list
  See getc #7 page 2 for others
FUNCTION

Removes a clist entry from list identified by r1, makes the second entry the first, puts the clist offset of entry removed from list in r2. Return to "normal".
If the list identified by r1 is empty, r2 is returned equal to zero, and return made to "empty".
If the list has just one entry, the entry is removed and the first and last character pointers for the list are zeroed.

ARGUMENTS

r1 (list identifier), clh(r1) (see sec 6 for general description of tty ip handling)

OUTPUTS

r2 (offset into clist of entry just removed from list r1), clh(r1), clt(r1), clist(r2)

CALLING SEQUENCE

jst r0, get j empty: ; normal:
ID 07-8  getspl

FUNCTION:  getspl  gets a device number from a special file name.  "v.namep" points to the name.  "namei" is called
to get the l-number.  (l-number -1) is the device number.  If it is
less than or equal to zero or if
it is greater than 9 an error occurs.  If not
the device number is returned in RI.

CALLING SEQUENCE:  jsr ro, getspl

ARGUMENTS:

INPUTS  v.namep  points to the name of the special file

OUTPUTS:  RI  device number of the special file
FUNCTION: "close" checks to see if the file, whose i-number is in RI, is special. If it is, a transfer is made to the appropriate routine. If it isn't a return is made.

ARGUMENTS

INPUTS

RI - contains i-number of file being closed

OUTPUTS

- If special file, RI is put on the stack, i.e., the i-number is put on the stack.

CALLING SEQUENCE

JSR 40, close
getc - the first
getc. removes a clist entry from a list identified by arg, via call to getc;
  decrements character count for list; puts the clist entry removed onto
  the free list; puts the character in the entry into r1 and takes "normal" return.
  If list is empty take "empty" return.

ARGUMENTS
  arg - list identifier

INPUTS
  r2 (clist offset from put)

OUTPUTS
  r1 (character on top of list), ecc(arg), ecc(c), clist(r2)

CALLING SEQUENCE
  jsr ro, getc ; arg ; empty: ; normal:
ID - U7j 5

OPT = OPEN 

OPT TAPe FILE FOR READ OR WRITE

Function — OPPT PERFORMS THE FOLLOWING FUNCTIONS

1. SETS THE READER ENABLE BIT IN PRS
2. ASSIGNs ALL PPT INPUT LOCATIONS IN "CLIST" TO FREE LIST
3. SETS PPTIFLG "TO INDICATE FILE JUST OPENED (=2)"
   AND PLACES 10 IN TOUTT ENTRY FOR PPT INPUT

Calling Sequence — JMP OPPT

Inputs — PPTIFLG - USED TO DETERMINE IF FILE ALREADY OPEN

Outputs — PPTIFLG - SET BY OPPT TO INDICATE FILE JUST OPENED
PRS - PROCESSOR PRIORITY SET TO 5
PRS - CONTAINS READER ENABLE BIT
TOUTT + 1 - CONTAINS COUNT FOR PPT INPUT
SEE OUTPUTS FOR "GETC"

ISSUE D DATE 3/2/72 IDUNIX SEC N.7 PAGE 3.0
Function: PPTIC performs the following functions for PPT input:

1. If the error, busy and done bits are not set in the PRS and the character count for PPT input in the CLIST is less than 30, PPTIC sets the reader enable bit.

2. Uses GETC to get character from paper tape input area of CLIST. If this area of "CLIST" is empty, a ciferk is made to see if 'PPTIFLG' is set equal to six (indication that error flag in PRS is set during normal operation). If it is, return is made to the calling routine which in turn returns to its calling routine. If 'PPTIFLG' does not equal six, the process is put to sleep.

Calling sequence: JSR, NO, PPTIC

Inputs:
- C + 2: Contains character count for PPT input
- PRS: Contains status bits for PPT reader
- PPTIFLG: Indicates condition of PPT file

Outputs:
- PRS: Contains reader enable bit
- See outputs for "GETC" and "CIFERK"
- PS: Processor priority set to 5 and then to 0
FUNCTION - "PPTOC" first checks to see if the character count for PPT output in theclist is > 50. If it is, the process is put to sleep. If it isn't "PUTC" is used to place the character which is in RI, in theclist. If the clist is full, the process is put to sleep. If the character is placed in clist, "STARPPT" is called to output the next entry in the PPT output section of clist.

CALLING SEQUENCE - JSR 140, PPTOC

INPUTS - CC+3 - character count for PPT output in clist

OUTPUTS - PS - process priority set equal to five
          see outputs for "STARPPT" and "SLEEP" and "PUTC"
ID 07-4  lopen

FUNCTION: lopen opens the file whose i-number is in RI. If the file is to be opened for reading, "access" is called and the the i-number is checked to see if the file is special. If it is special, a jump table of transfer addresses takes care of transferring control to the correct special file routine. If not special, file a return is made. If the file is to be opened for writing, "access" is called and a check is made to see if the file is a directory. If it is, an error occurs, because users cannot write into directories. Special files are handled in the same manner as above.

CALLING SEQUENCE  jsr ro, lopen

ARGUMENTS

INPUTS  RI - contains i-number of the file to be opened

OUTPUTS  - files i-node is in core
            RI - if i-number was negative upon entry it is positive on exit
FUNCTION: Takes a clist entry pointed to by r2, and makes it the last entry in the list identified by r1. If this is the first entry in a currently empty list then the first char pointer in cf is also updated.

ARGUMENTS:
INPUTS: r1 (clist identifier) r2 (clist offset)

OUTPUTS: cl+1(r1), clist+1(r2), cf+1(r1)

CALLING SEQUENCE: for ro, put
FUNCTION

CALLING SEQUENCE: jsr ro; put

ARGUMENTS:

INPUTS:  r2 = 2, 4, 6, ..., 510.
         r1 = minus 1
         cl = zero, or non zero

OUTPUTS: if cl = 0
         cf = r2/2
         cl = r2/2
         cf+1 = 0, if r2/2 < 128,
         cf+1 = 256, if r2/2 ≥ 128.

         if cl ≠ 0
         cl = r2/2
         clist + [r2+1] = r2/2

         if cl ≠ 0
         cf = r2/2
         cl = r2/2
         cf+1 = 0, if r2/2 < 128,
         cf+1 = 256, if r2/2 ≥ 128.
FUNCTION: sysmount

sysmount announces to the system that a removable file system has been mounted on a special file. The device number of the special file is obtained via a call to "getdp", and is put in the I/O queue entry for the dismountable file system (sbl) and the I/O queue entry is set up to read. (bit 10 is set), "ppoke" is then called to read the file system into core. This call is super user restricted.

CALLING SEQUENCE: sysmount; special; name

ARGUMENTS: special - pointer to name of special file (device)

name - pointer to the name of the root directory of the newly mounted file system. "name" should always be a directory

INPUTS: moti - records c-number of unique cross file device

sp - contains the name of the file

sbl - I/O queue entry for the dismountable file system

OUTPUTS: moti - c-number of special file

motd - device number of special file

sbl - has device number in lower byte

cdev - has device number in lower byte

file system is read into core now.

ppoke
FUNCTION - see 'sysrel' routine
CALLING SEQUENCE -
ARGUMENTS -
INPUTS -
OUTPUTS -
FUNCTION: systsumount announces to this system that the special file, indicated as an argument, is no longer to contain a removable file system. "getsp1" gets the device number of the special file. If no file system was mounted on that device an error occurs, mntd and mnti are cleared and control is passed to sysret.

CALLING SEQUENCE: systsumount special

ARGUMENTS special - special file to dismount (device)

INPUTS mntd - device number of mounted device
sb1 - I/O queue entry for the dismountable file system

OUTPUTS mntd - zeroed
mnti - zeroed
FUNCTION: puts a character at the end of a list identified by the argument in the putc call.

In detail, it takes a clist entry from the free list via call to "get". Appends the entry to the list identified by arg via call to "put". Then fills in the new entry with a character passed in ri.

ARGUMENTS: arg - list identifier (see discussion in G on my device I O)

INPUTS: ri - character from device buffer,

OUTPUTS: r2 - clist offset where character stored, cc(arg), clist-1(r2)

CALLING SEQUENCE: jsr ro, putc ; arg
FUNCTION: "bread" reads a block from a block structured device (disk, tape).

It operates in the following way:

1) If "cold" = 1 (cold boot), the block specified in RI is read into an
   I/O buffer via "preread." If its a warm boot (cold = 0), the block
   in RI and the next consecutive block are read into I/O buffers
   via "preread." The reason two blocks are read in is to speed up the
   overall reading process. On a cold boot however, only two I/O buffers
   are available, so only one buffer is used.
2) The block number is always checked to see if the maximum block
   number allowed on the device has been exceeded. (See argument.) If the
   block number does exceed the maximum an error occurs.
3) "preread" is called again on the first block. Since the first block is
   already in an I/O buffer, all preread will do is reverse the priority
   (see bufalloc, page 78) so that the first block is of
   higher priority than the second.
4) Bit 14 of the first block's I/O buffer is set.
5) Bits 10 and 13 (the read bits) of the I/O buffer are now checked
   if they are set (reading in progress) and the device is disk
   or drum, or the device is tape, and "uquant" = 0, "idle" is called.
   If the device is tape and uquant = 0, "sleep" is called.
   If bits 10 and 13 are 0 (read done), bit 14 of the I/O buffer
   is cleared and the data is moved from the I/O buffer to
   the users area. "dioreg" does the bookkeeping on the transfer.
6) If uucount = 0 the reading is finished. If not a branch
   back to the start is taken and the above steps are repeated.
7) A return is taken to the routine that called "readi.

CALLING SEQUENCE jsr ro, bread; arg
ARGUMENTS arg ro: maximum block number allowed on device
       R2 points to the users data area; R3 has the byte count
       (u, tofp) is the block number
       @dev is the device
       u.base - base of users data area
       ucount - number of bytes to read in
       RI is used internally as the block number.
       cold - 0 warm boot or 1 cold boot.
       @e - points to the beginning of the I/O buffer or the data area.
       uquant - byte quantum allowed for each process.
       E3 points to the next consecutive block to be read.
       R3 = 0 (used internally)

INPUTS

OUTPUTS -
FUNCTION: "write" writes on a block structured device (RF, RK, tape)
It operates in the following way:
1) The block number is placed in R1
2) If the block number exceeds the maximum allowable block
    number of the device an error occurs
3) (u->ofp) is incremented to point to the next block in sequence
4) "wslot is called to get an I/O buffer to write into
5) "dioreg" is called to set up the bookkeeping for the transfer.
6) The data is then transferred from the user's area to the I/O buffer
7) "dskwr" is called to write it out to the device
8) If u->count is 0 the procedure is repeated. If it is a
    return to the routine that called "write" is made

CALLING SEQUENCE: jsr 50, pwrite; arg

ARGUMENTS: arg - is the maximum allowable block number for the device

INPUTS: (u->ofp) is the block number
        u->dev is the device
        R1 is used internally to hold the block number
        R2 points to the I/O data buffer
        R2 points to the users data area, initially its u->base
        u->count - number of bytes user desires to write

OUTPUTS: (u->ofp) is the next block to be written into
        R3 = 0 (used internally)
FUNCTION: "dioreg" does the bookkeeping on block transfers of data. It first checks to see if there are more than 512 bytes to transfer. If so, it just takes 512. If not, it takes u.count.

ARGUMENTS:

INPUTS
- u.count: number of bytes user wants transferred
- u.base: start of user's data area

OUTPUTS
- R3: used internally to hold the count
- unread: updated by adding R3
- u.base: """" subtracting ""
- R2: has value of u.base before it gets updated.
FUNCTION

"bufaloc" scans the I/O buffers for block-structured devices, looking for an active buffer (bits 9...15 of the 1st word in the I/O queue entry for the buffer gate set), which has already been assigned to the block number and device currently under consideration, or for a free buffer (bits 9...15 set) which has been previously assigned to this device and block number. If there is no such buffer, the vacant buffer with the highest core address is assigned. If no free buffer is found, "bufaloc" calls "idle". Eventually, a buffer is located.

The routine "picky" which actually performs the I/O operations scans the "bufp" area of core from the highest to the lowest address, thus the priority of an I/O queue entry is established by where a pointer to the I/O queue entry appears in bufp.

The newly assigned buffer I/O queue entry pointer is placed in "bufp" thus making it the lowest priority I/O operation in the queue. The other entries in "bufp" are moved into higher addresses to accommodate the newly assigned buffer's I/O queue entry pointer at location bufp.

Once the buffer has been assigned the device number is put into the lower half of word 1 of the corresponding I/O queue entry and the block number is put into word 2 of the I/O queue entry.

CALLING SEQUENCE

```
jsr ra, bufaloc
```
FUNCTION  
dskrd acquires an I/O buffer, puts in the proper I/O queue entries (via buffetra) then
reads a block (number specified in fl) into the acquired buffer. If the device
is busy at the time dskrd is called, dskrd calls idle. Once the I/O operation is completed
rs is set to point to the first word in the buffer.

ARGUMENTS

INPUTS

OUTPUTS  
rs — pointer to first word in data block ; (rs) ; ps

CALLING SEQUENCE  
j sr ro, dskrd
FUNCTION dskwr writes a block out on disk, via p poke.
the only thing dskwr does is set bit 15 in the
first word of the I/O queue entry pointed to by
"bufp", "wslot" which must have been called previously.
has supplied all the information required in the
I/O queue entry.

ARGUMENTS

INPUTS

OUTPUTS (bufp)

CALLING SEQUENCE jsr ro, dskwr
FUNCTION

"drum" is the interrupt handling routine for the drum. It is called after the transfer of data to or from the drum is complete, i.e., when the ready bit in the dos (drum control register) is set. (See interface manual page 73-74.) R1, R2, R3 and clock p are saved on the stack. (See setisp) calls "trapt" to check for stray interrupt or error. If neither, it clears bit 12+13 in 1st word of transaction buffer, checks for more disk buffers to read into or write; return returns from interrupt by calling retisp.

CALLING SEQUENCE

called by interrupt vector at location 204 after data transmission has taken place, i.e., ready bit of dos set.

INPUTS

same as setisp, trapt and retisp

OUTPUTS

"""""

CALLED BY

interrupt vector

CALLS

setisp, trapt
ID - 483  error 10
FUNCTION - see 'error' routine
CALLING SEQUENCE -
ARGUMENTS -
INPUTS -
OUTPUTS -
ID 18, 4 poke

**FUNCTION**

Poke performs the basic I/O functions for all block structured devices. In order to understand the functioning of poke, the general handling of block structured I/O must be described.

I/O on block structured devices is handled via a collection of data buffers beginning at location "buffer." Each buffer consists of a four word I/O queue entry followed by a 256 word data buffer.

An I/O queue entry has the following form:

```
+-----------------+-----------------+-----------------+-----------------+
| write bit       | read bit        | waiting to write bit | waiting to read bit |
+-----------------+-----------------+-----------------+-----------------+
| inhibit bit     | device id       | physical block number | word count (-256) |
+-----------------+-----------------+-----------------+-----------------+
| bus address     |                 |                 |                 |
```

---

TFE/15 D. DATE 1/07/77 TD 14:09 sec 4.8 PAGE 5
byte 0 - device id codes are
0 = drum
1 = disk
other = dec tape

byte 1 - write bit - when set indicates
write the data in the buffer out
onto the device identified in byte 0.
read bit - when set indicates read data off of
the indicated device into the data
buffer
waiting to write bit - if set indicates that a write
operation has been requested but not
yet completed.
waiting to read bit - if set indicates that a read
operation has been requested but not
yet completed.
inhibit bit - when set will delay request for
operation indicated by write bit or read bit
until cleared.

byte 2-3 - physical block number (see sec 6, discussion of file system)
byte 4-5 - word count - number of words in buffer, loaded into
word count register for device
byte 4-7 - bus address - address of first word of data buffer.
In addition to the general I/O queue entries there are three special entries at locations sbo, sbi, and swp. These are the I/O queue entries for the super block for drum (sbo), the super block for the mounted device (sbi) and the core image being swapped in or out (swp) — these entries are initialized in the "allocate disk buffers" segment of code in uo.

An area in core starting at location "bufp" and extending nbnf + 3 words, contains pointers to the I/O queue entries. This table of pointers represents the priority of I/O requests, since poke scans these pointers starting at the highest address in "bufp", examining the control bits in byte 1 of each I/O queue entry pointed to by the bufp pointers. If either bit 9 or 10 is set and neither of bits 15, 13 or 12 is set then poke will attempt to honor the I/O request.

To honor an I/O request, poke checks "active" to see if the bit associated with the device is clear. If it is clear poke initiates the I/O operations by loading the appropriate device registers. In all I/O operations the interrupt is enabled and thus when completed an appropriate routine is called via the interrupt. When poke initiates a I/O operation it clears bit 9 or 10 and sets bit 11 or 12. The routine called upon completion of the I/O operation will clear bit 11 or 12 thus freeing I/O queue entry.
"poke" calculates a physical disk address, which is loaded into register RKDA, from the physical block number in the following way:

Let \( N = \) physical block number

then

\[
\text{sector number} = \text{remainder} \left( \frac{N}{12} \right)
\]

\[
\text{surface} = \begin{cases} 
0 & \text{quotient} \left( \frac{N}{12} \right) \text{ even} \\
1 & \text{quotient} \left( \frac{N}{12} \right) \text{ odd}
\end{cases}
\]

\[
\text{cylinder} = \text{quotient} \left[ \left( \text{quotient} \left( \frac{N}{12} \right) \right) / 2 \right]
\]

"poke" calculates a physical disk address for the drum from the physical block number in the following way:

the drum address is given in the dae and dar registers.

\[
\begin{array}{c}
daepx
\hline 10
\hline
15
\hline
\hline
\text{track}
\end{array}
\]

\[
\begin{array}{c}
darpv
\hline 10
\hline
0
\hline
\hline
\text{word}
\end{array}
\]

The physical block number is essentially multiplied by 256 (by shifting the low order byte into the high order byte of the dar), and shifting the high order byte into the low order byte of the dae.
ARGUMENTS

INPUTS — bufp, ..., bufp+22, deverr; active
          (bufp), ..., (bufp)+6, (bufp+2), ..., (bufp+22)+6 (I/O queue entry)

OUTPUTS — sets bits 12, 13 on I/O queue entries where I/O operation is initiated, active, rkap, rkap
           dae, dar, wc, cma, dc5, rkcs, rkw, rkba, rhda

CALLING SEQUENCE — jst ro, poke
FUNCTION

"preread" is called by "bread" to read in a disk block on device "cdev." The block number is in Ri. "preread" gets a free I/O buffer via "bufalloc." It sets bit 10 of the first word of the I/O buffer and then reads the specified block into the I/O buffer via "poke." If the I/O buffer already contains the specified block bit 10 is not set and the call to "poke" is skipped. The processor status is then cleared.

CALLING SEQUENCE

lsl r0, preread

ARGUMENTS:

INPUTS

R1 - block number to read
RS - points to first word of I/O buffer

OUTPUTS

- specified block into an I/O buffer
PS = 0

RS points to first word of the I/O buffer
FUNCTION: "rtap" is the read routine for dec tape. The device number is \[(l\text{-number}/2) - 4\]. The l-number is in R1 upon entry. "bread" is called to read the proper block in blocks.

CALLING SEQUENCE from jump table in readi

ARGUMENTS:
- INPUTS
  - R1 is the l-number of the special file
- OUTPUTS
  - cdev is the device number
  - See outputs for "bread"
FUNCTION

"tape" handles the dec tape interrupts.
"setisp" is first called to set registers and the
clock. The state of the dec tape (testate)
(i.e., reading, writing, idle etc.) is passed.
"trapt" is then called to check for data transmis-
sion errors. If none occur control passes to
the appropriate dec tape routine depending
on what the state is. Control is passed by
putting R3 in the PC. If an error occurs a
CALLING SEQUENCE: Interrupt vector

ARGUMENTS:

INPUTS
testate - the state of the dec tape (read, write etc.

OUTPUTS - control passes to appropriate dec tape routine

PC - set to address of above routine

R3 is used to hold the address of above
routine
FUNCTION: "tstdev" checks to see whether a permanent error has occurred on special file I/O. (It only works for tape, however). If there is an error, the error is cleared and the user is notified.

CALLING SEQUENCE: jsr 70, tstdev

ARGUMENTS:

INPUTS:
- cdev - the device in question
- (R1) + deverr - the device's in question error indicator

OUTPUTS:
- R1 = cdev - the device number
- If no error, nothing else happens
- If error, (R1) + deverr, get cleared and user notified via error 10.
FUNCTION: "trap" is part of the drum, disk, or dec tape interrupt handler. The ready bit of the device control register is checked. If the ready bit is not set the device is still active so a return through "retisp" is made. It then checks to see if a stray interrupt has occurred. If not "trap" checks to see if an error in the data transmission has occurred. If so, the return is skipped. If not, the return is not skipped. The return is via a JMP.

Calling sequence: jsr ro, trap 3 dv 5 but 5 act

br normal
br error

Arguments dv - device control status register. (For dec tape it is the command and register) but - contains address of disk buffer being read into or written act - tested against the bits in "active" to see if the device was busy

Inputs: active - contains bits that tell which devices are busy

Outputs R1 - points to the disk buffer
R2 points to the device control and status register or command register depending on the argument
FUNCTION - \textup{wslot} calls "bufaloc" and obtains as a result a pointer to the I/O queue of an I/O buffer for a block-structured device. "Bufaloc" has inserted into this I/O queue the device number and block number which \textup{wslot} passes from its caller to "bufaloc". It then checks the first word of the I/O queue entry. If bits 10 and/or 13 (read bit, waiting to read bit - see H.8 p.5) are set, \textup{wslot} calls "idle".

When "idle" returns, or if bits 10 and/or 13 are not set, \textup{wslot} sets bits 9 and 15 of the first word of the I/O queue entry (write bit, inhibit bit), sets the processor priority to zero, and sets up a pointer to the first data word in the I/O buffer associated with the I/O queue.

CALLING SEQUENCE - \texttt{jsr \, ro, wslot}

ARGUMENTS -

INPUTS - See inputs for "bufaloc" - H.8 p.1

OUTPUTS - \begin{itemize}
  \item \texttt{bufp} - bits 9 and 15 are set, the remainder of the word is left unchanged
  \item \texttt{rs5} - points to first data word in I/O buffer
\end{itemize}

See outputs for "bufaloc" - H.8 p.1. Note that outputs given above take precedence over outputs from "bufaloc".
ID - U9; 6  RCVCH - RECEIVE CHARACTER

FUNCTION - "RCVCH" USES "GETC" TO READ A CHARACTER FROM THE TTY'S READ SECTION OF THE CLIST. IF IT IS EMPTY, THE PROCESS IS PUT TO SLEEP. WHEN THE PROCESS IS AWAKEN, RCVCH AGAIN TRIES TO OBTAIN A CHARACTER FROM CLIST.

CALLING SEQUENCE  JSR  RO, RCVCH

INPUTS  X2 - CONTAINS 8X TTY NO.
NCER + 8XTTYN - CARRIER DETECT & CLEAR DATA TERMINAL BITS
SEE INPUTS FOR "GETC" AND "SLEEP"

OUTPUTS  PS - SET PROCESSOR STATUS TO 5
SEE OUTPUTS FOR "SLEEP" AND "GETC"
RCUT - READ TTY

**FUNCTION**

"RCUT" places TTY characters in the user buffer area. If the "raw" flag in the TTY area is set, a character is obtained from the TTY's input area of CLIST. If the flag is not set, "canon" is used to process a line of TTY characters and place them in the user's buffer area.

**CALLING SEQUENCE**

JMP RCUT

**INPUTS**

RCR + EXTTYNO - Carrier detect and clear data term bits
TTY + EXTYYNOT6 - Pointer to TTY buffer
TTY + EXTYYNOT4 - Raw data flag

See inputs for 'canon', 'pasc', 'getc', and 'rcuch'.

**OUTPUTS**

PS - Set processor priority to 5

See "canon", "pasc", "getc", "rcuch", and "sleep" outputs.
ID-U933  STARRYMT

**Function**  STARRYMT does the following:

1. Checks to see if the output character count for the TTY in Clist is ≤ 10. If it is, "STARRYMT" uses "WAKEUP" to wake up the process identified in the "WLIST" entry for the TTY's output channel.

2. Checks to see if the TOUT entry for the TTY output is equal to zero. If it is NOT, control is passed back to the calling routine.

3. Checks to see if the ready bit in the TTY's TCSR register is set. If it is NOT, control is passed back to the calling routine.

4. Checks the byte of TTY's "TTY" area (contains the character left over after LF for a null character. If the byte contains a non null entry, the entry is used as the next character to be output. If the entry is null, the next character to be output is obtained from the Clist via "GETC".

5. A00s 200s to ASCII code of character to be output. If digit 2 (for left digit) of entry in "PARTAG" table for character is a "2",

6. Checks TTY's RCSR buffer to determine if carrier is present. If it is not the character is "dropped" and a new character is obtained by returning to the beginning of the subroutine. If the carrier is present, a check is made to determine if the character to be output is "HT". If it is a check is made to see if the "TAG TO SPACE" flag (bit 1 of 5th byte in "TTY" area) is set. If it is the character to be output is changed to a space (ASCII 40h)

7. Places character to be output in TTY's "TCBR" buffer. STARRYMT then does one of the following dependent on the character to be output (digits 0 and 1 of the characters "PARTAG" entry are used as offsets into jump table)

   a. For ASCII codes 40-176, increments column pointer which is in Byte 2 of TTY area
   b. For ASCII codes 0-9, 16-37 and 177, does nothing
   c. For ASCII 01 (CS), decrements column pointer
   d. For ASCII 01 (LF), checks for setting of CR flag (bit 4 of 4th byte in "TTY" area). If it is ASCII 01 (CR) is placed in Byte 2 of TTY area (character left over after line feed). STARRYMT then determines value for the TTY's output entry in the output table. This value is dependent on whether "LF" is to be output or
a. For ASCII 012 (HT), does some foiling around with column count and 3rd byte of TTY area (character left over after LF) dependent on value of "tab to space" flag in 5th byte of TTY area. It then determines value for the TTY's output entry in the Tout Table.

b. For ASCII 013 (VT), determines value for the TTY's output entry in Tout Table.

c. For ASCII 015 (CR), determines value for the TTY's output entry in Tout Table and sets column pointer = 0.

calling sequence — JSR 100, STXRTM

inputs — (SP) — contains &XTTY NUMBER
TTY+3 + &XTTY NUMBER — contains CC, CP, and TL lists for TTY
CC + (TTY + 3 + &XTTY NUMBER) + 1 — contains character count for TTY output in CLIST
TTY + 1 + &XTTY NUMBER — contains column pointer for TTY
TTY + 2 + &XTTY NUMBER — contains character left over after LF for TTY
TTY + 4 + &XTTY NUMBER — contains flags for TTY

see outputs from 'GETC'

RSR + &XTTY NUMBER — contains ! carrier present flag for TTY
TSR + &XTTY NUMBER — contains ready flag for TTY

outputs — see inputs to 'GETC'

TC + (TTY + 3 + &XTTY NUMBER)
TTY + 1 + &XTTY NUMBER — see inputs above
TTY + 2 + &XTTY NUMBER
TCR + &XTTY NUMBER — contains character to be output on TTY
TOUT + 3 + &XTTY NUMBER — contains Tout entry for TTY
FUNCTION — "XMTT" uses "CPASS" to obtain the next character in the user's buffer area. If the character count for the TTY (identified by node number of TTY's special file in stack) is ≥ 50, the process is put to sleep. If not, "XMTT" uses "PUTC" to determine if there is an entry available in "freelist" portion of "CLIST". If there is, "PUTC" places the character there and assigns the location to the TTY portion of "CLIST". If there is no location available in freelist portion of "CLIST", the process is put to sleep. If there is a vacant location, "STARKMT" is used to attempt to output the character on the TTY. Upon return from "STARKMT", the next character is obtained from the user's buffer area. If the buffer is empty, control is passed back to the calling routine via "CPASS". When the process is awakened by "AWAKE", it tries again to find a location available in freelist and a character count for the TTY output < 50 so it can output characters.

CALLING SEQUENCE — JMP XMTT

INPUTS — See inputs for "CPASS".

(SP) — contains a number of TTY's special file

NI — contains character to be placed in CLIST upon return from 'CPASS'

OUTPUTS — See inputs for "STARKMT" and "PUTC"

Processor priority set to five
FUNCTION:
ru1 is called by sysread and syswrite. It puts the buffer pointer (butler) into u.base and the number of characters (nchars) into u.count. It then finds the c-number of the file to be read by getting the file descriptor in *u.ro and calling "getf." The c-number so returned in RI.

ARGUMENTS
- buffer - buffer pointer
- nchars - number of characters
- (u.ro) - file descriptor

INPUTS

OUTPUTS
u.base - buffer pointer
u.count - number of characters
RI - contains the c-number of the file to be read

CALLING SEQUENCE
for c, ru1
FUNCTION: "intract" checks to see if the process was a quit or interrupt from the typewriter. If it was a quit, the quit flag is cleared and the 7 bit (trace trap) of the processor status is set. If the interrupt character is a "del" (177), v.intr is checked to see if it is equal to the address "core." If it is, control is transferred to "core." If not, sys.exit is taken.

CALLING SEQUENCE
br intract

ARGUMENTS

INPUTS
(s,p) - contains the instruction RO is pointing to

v.itty - pointer to buffer of tty in control of the process (R1)+6 - interrupt character in the control tty's buffer

v.intr - determines handling of interrupts (See sys.intr in the UNIX programmers manual)

OUTPUTS

clock pointer is popped.

If the interrupt char is a quit character
(R1)+6, the interrupt character in the control tty's buffer, is cleared.
8 bit of ps is set

If the interrupt char is a "del" (interrupt)
(R1)+6 is cleared
control is transferred to "core" if (v.intr)= "core"
mget; 

**FUNCTION**
mget takes the byte number of a byte to be read/written in a file and obtains the physical block number of the block in which it occurs. The file offset for the byte (ie the byte number) is passed by passing a pointer to the offset in u.tofp. The block number for the byte is returned in r1.

Along the way several things can happen:

1. The file is small (less than 8*256 words) and the byte number extends beyond the current size of the file but does not exceed 8*512. In this case mget assigns a new block from the free area of the file device and updates the i-node for the file by adding the physical block number of the new block. And modifying the free storage map.

2. The file is small and the byte number exceeds 8*512. In the case the status of the file changes from small to large. mget sets the large file bit in i-flgs of the i-node. Next an indirect block is assigned to the file. The block pointers in i-node are moved into the new indirect block and a pointer to the indirect block is put in the i-node. Next a new data block is assigned via the large file handling logic, described below.

3. The file is large and the byte number exceeds the current size of the file, but does not exceed the capacity of the highest indirect block. mget assigns a new file block and adds a new entry to the indirect block.

4. The file is large and the byte number exceeds the current size of the file, and also exceeds the limit of the highest indirect block. An indirect block is assigned from free storage and a pointer to it put in the i-node. Then a new file block is assigned and a pointer to it stored in the new indirect block.

See File Structure write-up in the UNIX programmer's manual.

**CALLING SEQUENCE**
```
jsr ro, mget
```

**ARGUMENTS**
```
INPUTS — u.tofp (file offset pointer), iinode, iu.off (file offset) 
OUTPUTS — r1 (physical block number), r2 (internal), r3 (internal), r5 (internal)
```
FUNCTION: "otty" opens the console tty for reading or writing. The interrupt enable bits are set in the TKS and the TPS. If the console is the first tty opened in this process assign its buffer address to utttyp return through "sret."

CALLING SEQUENCE: [conditional or unconditional branch, or jmp] otty

ARGUMENTS:

INARGS:

see inputs for 'sret'

utttyp - points to the buffer header for the process control typewriter.
(tty + 70.) - lower byte of 1st word of header contains the no. of processes that opened the buffer.
(tty + 70. - 1) contains pointer to the header of the console tty buffer.

OUTPUTS:

utttyp - points to the console tty buffer header if it was the 1st tty opened by the process. Otherwise points to?
(R5) - points to header of console tty buffer
(R5) - lower byte (no. of processes that opened the buffer) incremented by one.

tk5 - reader status register interrupt enable bit set, rest of bits zeroed.
tps - punch status register "

see outputs for 'sret'