Annotated TSS internals

UNPUNCHED
BULLETINS
1959 - 3

Allocation of ECS

Capabilities & Capability Lists

Event Channels

Files

The Line Collector

Maps

Operations

Processes

Subprocesses

System Entry/Exit

Teletype IO Functions

TS Interrupt System

TSS Text Standard

p 58	-2 → -1 etc	4Feb70
p 34	the → a	"
p 46,47	param passing stuff ← stuff+1	9June70
p 37	redescribe interrupt, fixed that flag	15Nov7

CAL-TSS Internals Manual

November 1969

Computer Center
University of California
Berkeley

Table of Contents

System entry/exit - Operation Interaction

Control passes from the user to the entry point (USERCAL) of the system entry/exit routines when the user executes a CEJ instruction. Control returns to the user (at S.RETU) at the end of the system entry/exit routines, again by a CEJ instruction. Thus the system runs in monitor mode, while the user runs in user mode. The function of these routines is to determine the reason for the user's call upon the system (i.e., the operation he wishes to perform), to collect and check the parameters needed for the operation, to transfer control to the proper system action routine specified by the operation, and to handle the return to the user after the system action is completed.

Operations are ECS system objects which control the calling of system actions or subprocess call actions, and provide the mechanism to facilitate "layers" in the system. Internally, an operation consists of one or more orders, each in the same format, prefixed by a fixed sized header. Each order specifies an action and consists of an action number, which is an index in a jump table of ECS action addresses; parameter information, which is used by the system entry/exit routines as explained below; and those parameters for the action(s) of the operation which are fixed for each call (see Figure 1).

The parameter information consists of a variable-length sequence of bytes, one for each parameter to indicate its type. The parameter specification types and their descriptions follow:

PS.UCAP user-supplied capability (which must match the type and option bits stored in the operation)

PS.UDAT user-supplied 60-bit datum (no checks are performed)

PS.FCAP fixed capability (both words of a capability are stored in the operation, and no corresponding information is taken from the user's input parameter list)

PS.FDAT fixed datum (a 60-bit data word is stored in the operation, and always passed unchanged)

PS.ACAP any capability (a capability is expected from the user, but no type or option checking is to be performed on it)

PS.NONE none (when an operation is created, all parameter specifications are initialized to "none", but they must be fixed with the various actions supplied before the operation may be used).

A bit in the operation is set if the action involved is parameterless; in this case there are no parameter specification bytes. When present, the 3-bit parameter specification bytes are packed 19 to a word, with special bytes to indicate "end of current byte-word" and "end of all bytes". Following the byte-words come the parameter specification data-words: a word containing option bits and a capability type for PS.UCAP parameter specifications, and the fixed parameter (a two word capability for PS.FCAP bytes, a 60-bit datum for PD.FDAT bytes), but nothing for any other type of parameter specifications. If the action specified by the operation is a "subprocess call" or "jump-call", a flag bit in the operation indicates the presence of a class code (name of the subprocess to be called), and the class code will appear after the data words in the operation.

In addition to the above mentioned information, each order of an operation contains several fields used to "traverse" the operation when it is being interpreted or changed. There is a total-number-of-parameters field, which is cumulative for all orders from the first through the current order, as well as a cumulative-total-number-of-capability-parameters field. Also, each order contains the length and origin (relative to the beginning of the operation) of the next order.

At system initialization time, operations for all available ECS system actions are created. In addition, operations may also be created for particular subprocess calls or jumps. The user of the TSS may call upon any of the system action operations for which he has been given a capability. The system actions created at initialization allow the user to create and manipulate ECS objects, e.g., create, read and write files, and create subprocess call or jump operations etc. (See individual documents and below for details.)

On entry to the system entry/exit routines (at USERCAL) the origin of the process descriptor (see Processes) has been picked up in B1 by the exchange jump. The origin of the process descriptor will remain in B1 through all system actions. First, the system and user clocks are updated. The difference between S.OLDTM, which contains the value of S.CHARG from the last time it was updated, and S.CHARG, which runs whenever the interrupt system is not running, is added to the system total user time (S.URSTM) in system core and to the user's total user time (P.USRTM) in the process descriptor.

The CEJ instruction which caused the transfer of control is then examined to find the address of an input parameter list (see Figure 2). It is expected that the CEJ which the user executed was in the upper two parcels of the instruction word. The low order 18 bits of the 30 bit CEJ instruction are extracted and interpreted to locate an input parameter list. If the 18 bit field is negative, the complement of the low order 4 bits specify which register in the user's exchange package contains the input parameter list (IP list) pointer (e.g., -3 → B3; -10 → X2). Otherwise, the 18 bit field is taken to be the IP list pointer. This pointer is checked for legality (i.e., must be positive and less than user FL) and an error is generated if necessary. Finally, the IP list pointer is saved in the process descriptor at P.IPLIST in case it is needed to form a stack entry for a subprocess call. Also the stack manipulation flag (P.OLDP), which controls the updating of the old stack entry in case of a subprocess call, is reset.

Next, the first word of the IP list (called IP0) is expected to be, and is interpreted as, a full C-list index of the operation for the desired action. The corresponding capability is fetched by calling GETCAP (note that a negative or overly large C-list index will cause an error to be generated). This capability is checked to insure that it is a capability for an operation; if it is not, an error is generated. The first order of the operation is read from ECS by reading the fixed-sized header and then reading in the first order.

The parameter specifications of the first (and possible only) order of the operation are used by OPINTER to form an actual parameter (AP) list in the process descriptor starting at P.PARAM. This list consists of the two actual words of each capability parameter and one word for each datum parameter. Parameters which are fixed in the operation are copied directly to the actual parameter list. User supplied parameters are drawn from the IP list, which is expected to contain in successive words, C-list indices and data parameters. C-list indices are checked to assure that they fall in the range of the full C-list and are used to fetch the actual two words of user-supplied capabilities. User-supplied capabilities are checked for the correct type and required options unless the parameter specification is "any capability". User-supplied datum parameters are transferred directly from the IP list to the AP list without any checking. If a "none" parameter specification is encountered, an error is generated and parameter processing is terminated.

For operations which are flagged as being parameterless, the interpretation of parameter specifications is omitted. After the completion of the actual parameter list (AP list), the operation is checked to see if it requires a subprocess name and parameter type bit masks (i.e., it is a subprocess call for why? operation). If so, the subprocess name is copied from the operation to P.PARAMC in the process descriptor, the number of parameters is stored in P.PARAMC-1. In addition, the input-parameter list address is stored in P.IPLIST and the F-return count, which is now zero, is stored in the top stack entry.

Finally, the ECS action number is extracted from the operation; it is used as an index to jump into the ECS action jump table starting at ACTIONL where there will be a jump to the proper entry point for the desired action.

Upon successful completion of an ECS action, the ECS action routine normally returns to the system entry/exit routine to return control to the user. The only exceptions to this are the case in which the user process has hung on an event channel or exceeded its quantum, in which case the event channel system ~~routine~~ exits to the swapper, and the case in which an F-return has been made. An F-return results when a situation arises which is not serious enough to cause an error but does not permit the action to be carried out normally.

There are three points in the system return sequence to which an ECS action may return: SYSRET, TOUSER and S.RETU. The normal return begins at SYSRET. This return updates the user's P-counter in accordance with the user supplied P-counter offset which is stored in the low order 18 bits of the CEJ instruction word originally used to call the system. The legitimacy of the new P-counter (old P-counter + P-counter offset) is checked and an error may be generated. Falling through to TOUSER the normal return updates the system time clocks at S.SYSTM in system core and P.SYSTIM in the process data area and checks to see if the user's quantum has run out. If S.QUANT is positive (quantum has run out) the swapper is entered at SWAPOUT. Otherwise, an exchange jump (CEJ) is executed at S.RETU to return control to the user.

If an F-return results, either from a subprocess call action or an ECS action, control transfers to SYSFRET, and the IP list whose address is the top entry of the stack or P.IPLIST is consulted. The count of the number of orders in the operation, kept in the header word, is checked and if remaining orders exist, the F-return count in the stack is incremented, and the next order of the operation is interpreted. This proceeds as previously described, except that the parameter specification of all orders up to and including the current one are used to form the actual parameter list. If any one of the subsequent orders terminates normally, the return is through SYSRET as described above. If the F-return count reaches the number of orders in the operation, then the return is to TOUSER and behaves the same as the return to SYSRET except that the user's P-counter is not modified. (This return is used by the subprocess calling, subprocess return and process interrupt action routines.)

If the action resulted in an error, control transfers to E.ERROR where error processing, which involves calling a subprocess in the user's process to handle the error, is initiated. (See Subprocesses.)

The entry S.RETU is used by the swapper after a process has been swapped in to transfer control to the user; it consists only of the CEJ instruction.

Creating an Operation

When an operation is created, each of its parameter specifications is initialized to "none", and the operation thus may not be invoked (unless it is parameterless). There are two actions for creating new operations. The first creates an operation of order 1 to call or jump-call a designated subprocess. The second action creates an operation of order N. It is supplied with an operation of order N-1, which is copied with a new order appended, again to call or jump-call a named subprocess.

ECS system actions are also available to copy an existing operation and then to modify the parameter specifications as well as to destroy an operation.

In order to specify the parameter specifications in an order of an operation created in any of the ways just described, a set of actions is provided. Each takes as parameters an operation, a parameter specification index (for which purpose the order-boundaries of the operation are ignored), and further information in certain cases. The actions are

ACAP change a PS.NONE to a PS.ACAP specification (no further parameters need be supplied)

UDAT change a PS.NONE to a PS.UDAT specification (no further parameters)

UCAP change a PS.NONE to a PS.UCAP specification (two additional parameters are required, a type and an option bit mask)

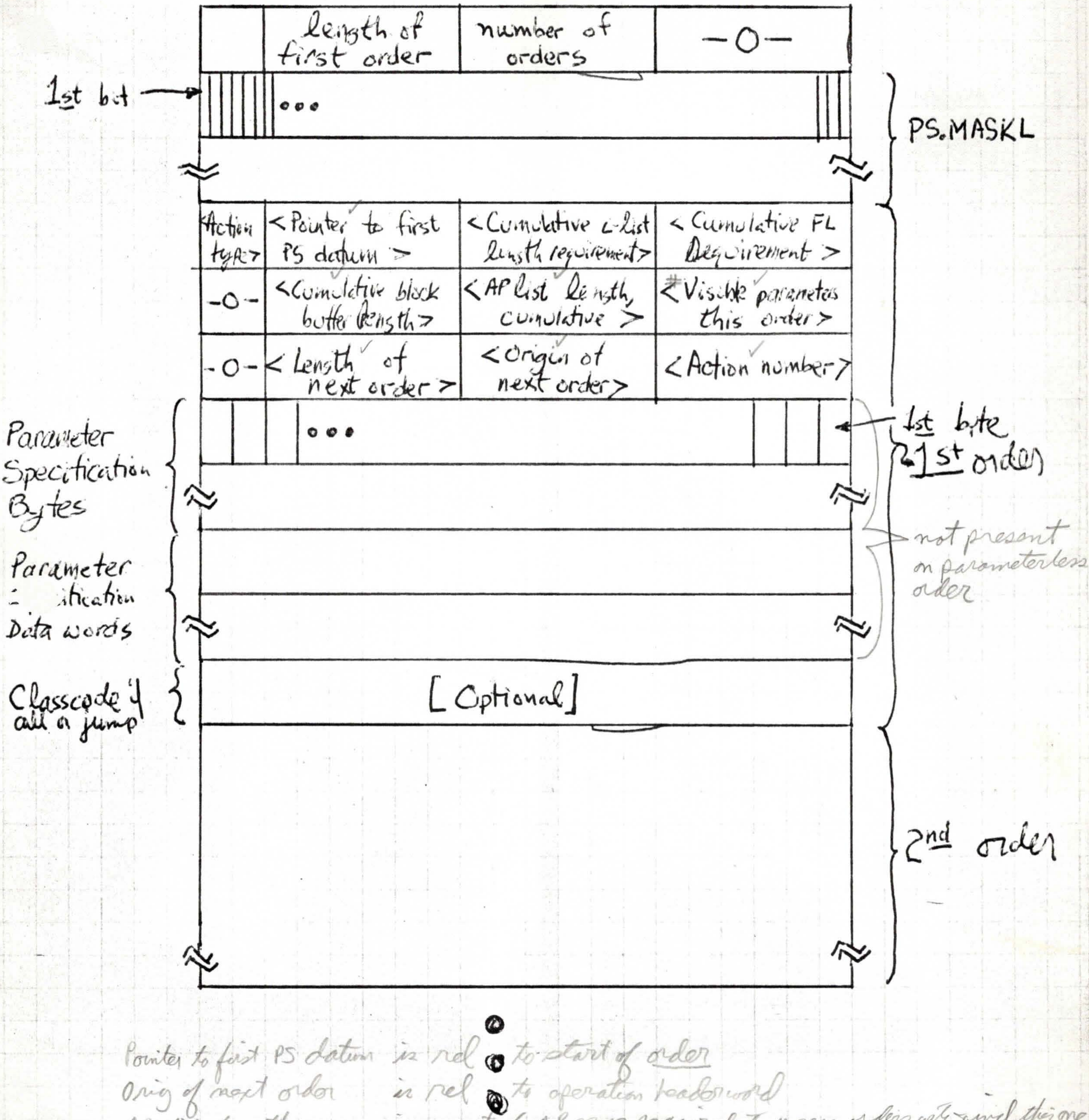
FDAT change a PS.UDAT to a PS.FDAT specification (an additional parameter, a 60-bit datum, is required)

FCAP change a PS.UCAP to a PS.FCAP specification (an additional parameter, a capability index, is required)

Note in the last two cases that "fixing" a parameter specification requires two steps, changing the specification first to a user-supplied type and then to the corresponding fixed type.

The UCAP, FDAT, and FCAP actions involve reallocating the operation in ECS, since in each case one extra word is added to an order of the operation. Also, the cumulative-total-of-capability-parameters field must be updated in the affected order, and all fields after it, in the case of the UCAP and FCAP actions.

Operations



	length of first order	number of orders	-0-
<Action type>	<Pointer to first PS datum>	<Cumulative C-list length requirement>	<Cumulative FL Requirement>
-0-	<Cumulative block buffer length>	<AP list length, cumulative>	<#Visible parameters this order>
-0-	<Length of next order>	<Origin of next order>	<Action number>

1st bit
•••
PS.MASKL
•••
1st byte
1st order
Parameter Specification Bytes
Parameter Specification Data words
not present on parameterless order
Classcode if call a jump
[Optional]
2nd order

Pointer to first PS datum is rel to start of order
Orig of next order is rel to operation headerword
AP list length, cum is amount of AP space required to process orders up to and this one
block buffer length, cum is space reqd to hold block data case thru this order
Cum C-list length reqmt is # slots reqd in C-list of called SP thru this order
" FL reqmnt is memory cells reqd in " " " " "

Action type = 40 if parameterless
20 subprocess call or jump

Parameter-type bit mask: 0=datum, 1= capability or block (which also uses two words in the AP list)

Parameter Specification Byte Value:
0 -(end of specs)
1 - (go to next word)
2 - PS.NONE (unspecified)
3 - PS.UDAT (user-supplied datum
4. - PS.FDAT (fixed datum)
5 - PS.FCAP (fixed capability)
6 - PS.BLK (block ~~[illegible]~~ specification)
7 - PS.UCAP (user-supplied capability)

Parameter Specification Data word formats:

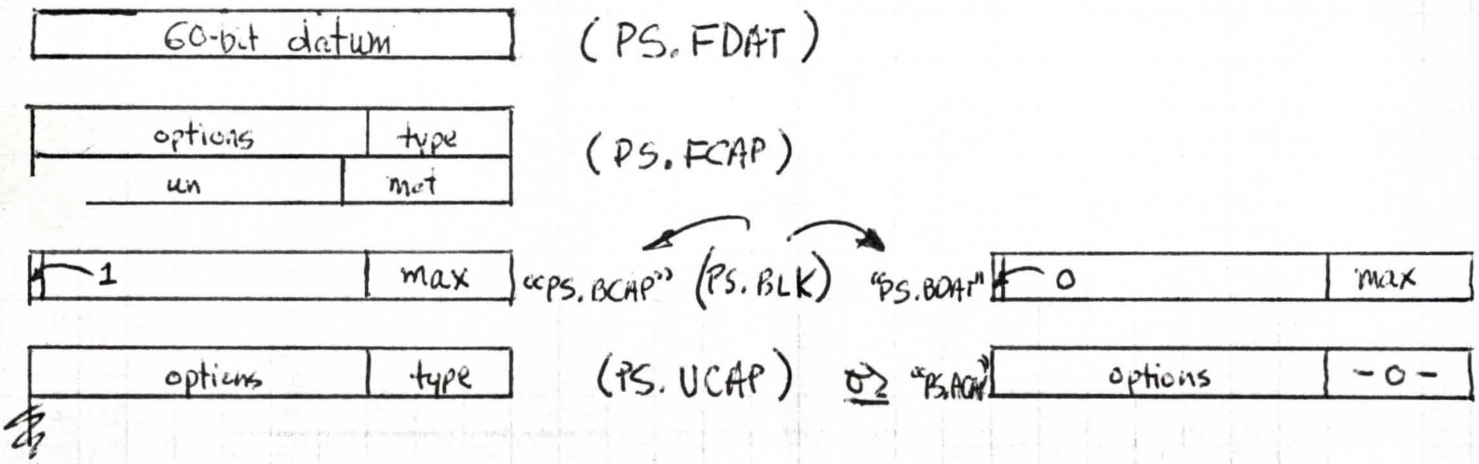


Figure 1

Operation

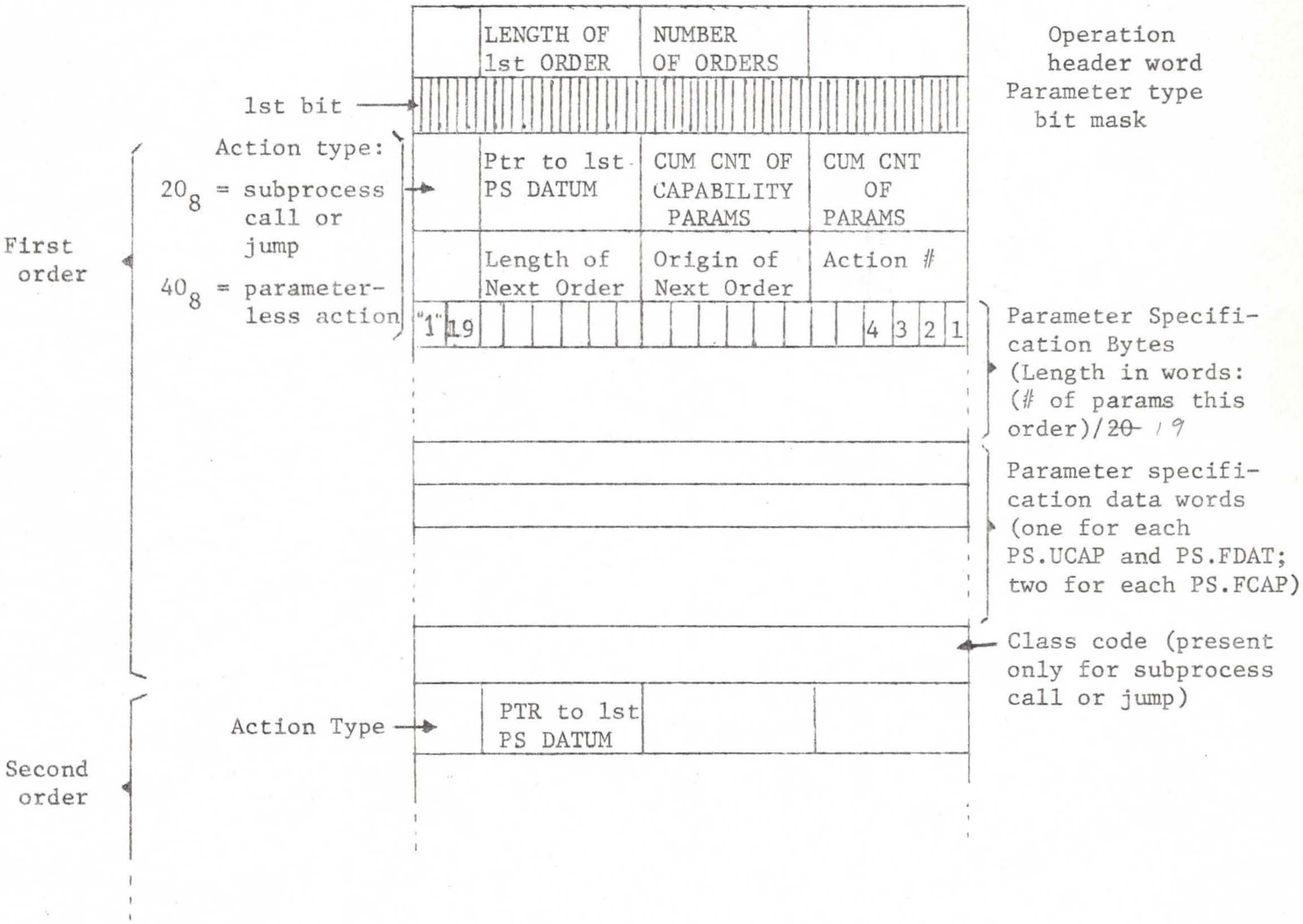

LENGTH OF 1st ORDER
NUMBER OF ORDERS
Operation header word
1st bit
Parameter type bit mask
Action type:
20_8 = subprocess call or jump
40_8 = parameter-less action
Ptr to 1st PS DATUM
CUM CNT OF CAPABILITY PARAMS
CUM CNT OF PARAMS
Length of Next Order
Origin of Next Order
Action #
First order
"1"
19
4
3
2
1
Parameter Specification Bytes (Length in words: (# of params this order)/~~20~~ 19
Parameter specification data words (one for each PS.UCAP and PS.FDAT; two for each PS.FCAP)
Class code (present only for subprocess call or jump)
Action Type
PTR to 1st PS DATUM
Second order

Figure 1 - Operations (con't)

Parameter type bit mask: 1 = capability; 0 = datum

Parameter Specification Byte Values

0 - end

1 - last byte this word

2 - PS.NONE (unspecified)

3 - PS.UDAT (user-supplied datum)

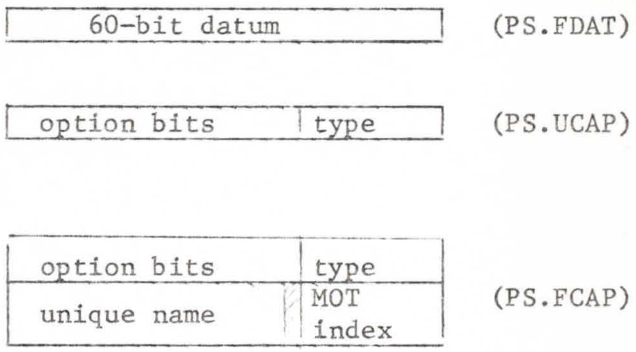
4 - PS.FDAT (fixed datum)

5 - PS.FCAP (fixed capability)

6 - PS.ACAP (any capability)

7 - PS.UCAP (user-supplied capability)

Parameter Specification Data Word Formats



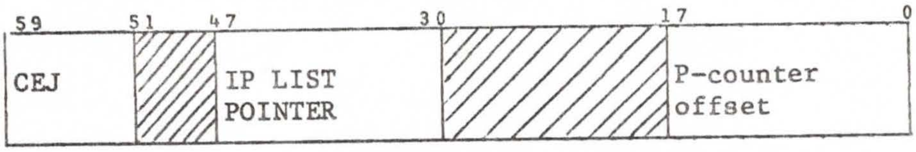
<PTR to 1st PS DATUM>::= pointer relative to start of that order

<CUMUL CNT OF PARAMS> ::= total number of parameters through that order

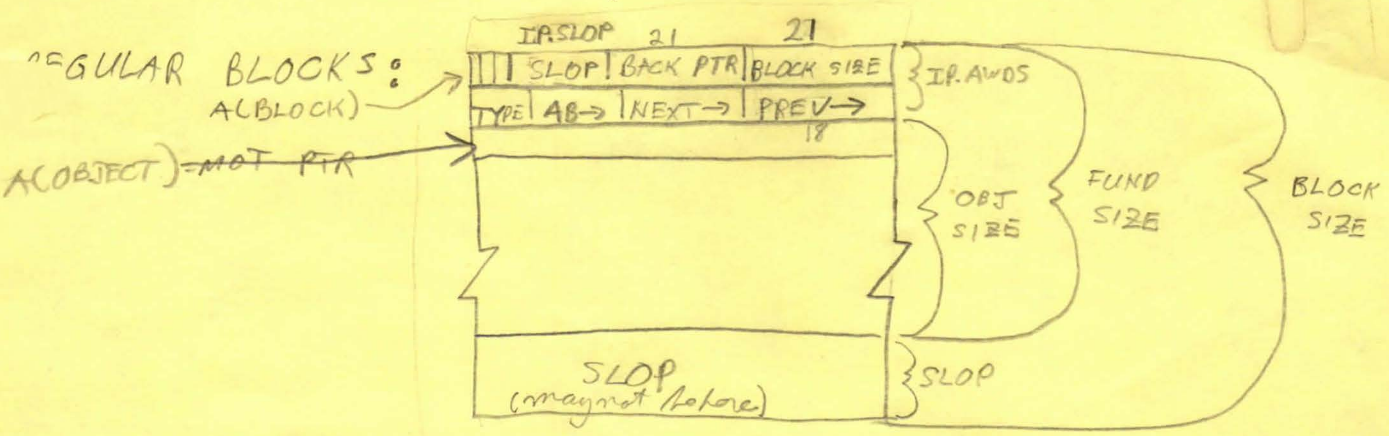
<ORIGIN OF NEXT ORDER> ::= origin relative to operation header word

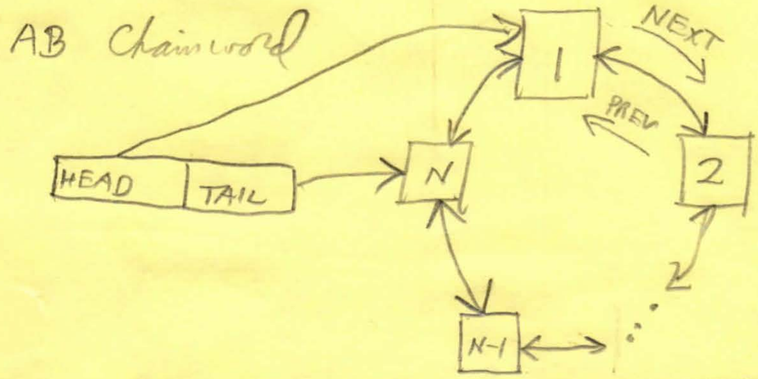
Figure 2

System Call



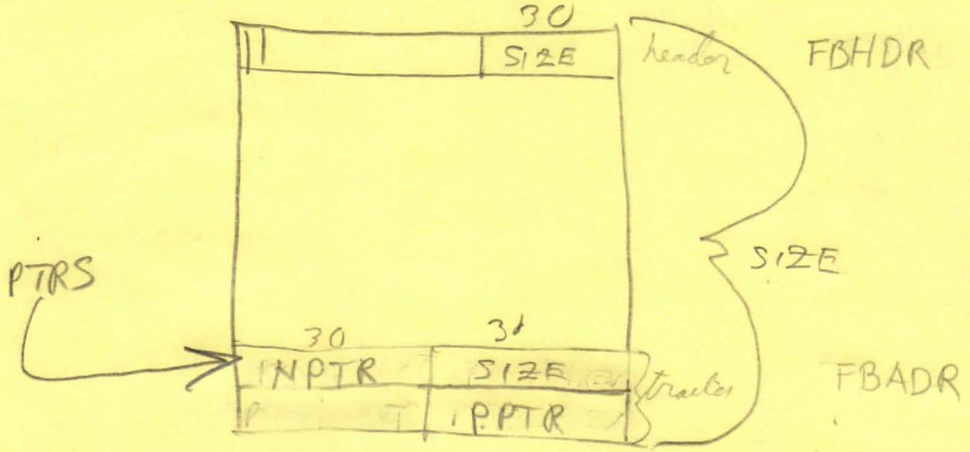
REGULAR BLOCKS:





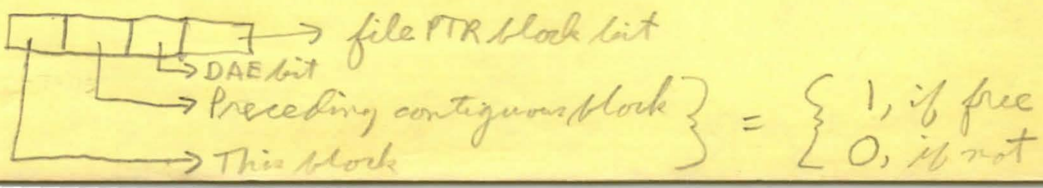
(these pointers are MOT indices)

FREE BLOCKS



NPTR is the pointer to the next FB
PPTR " " " " " prev FB

BLOCK BITS



General Considerations wrt the operation of the allocator routines

1) ~~2~~ >1 CPUs with more than 1 CPU ~~using ECS,~~ provision must be made to prevent from accessing certain info at the same time; a test & set register is talked about, but meanwhile dummy RESERVE & RELEASE macros have been used in the code to indicate where certain things are locked & unlocked.

These currently include

ALLOCBLK - locks up EC.ABPCK, ~~the MOT,~~ & prevents modification of any allocation block

ALLOC - reserves EC.APACK & the freespace chains

2) The MOT is stuffed into low core so that 18-bit arithmetic may be used in manipulating MOT indices. The PTRs in the MOT are 21 bits & have to be handled with 60-bit arithmetic

3) The amount of space in the free chain (EC.APACK+2 in ECS) is kindly maintained in central at cell 3 for display, etc.

4) The size field (which is 1 more than the usable size of an object or file block) is restricted to $2^{17}-1$ so as to be suitable for 18 bit arith

COMPACTI~~FICATION~~NG, INCREMENTAL

Basically, the compact~~ifier~~or (herinafter refered to as "C") is just supposed to push all the objects to one end or the other of memory so that the free space all merges into one nice contiguous usable piece. Even if C were allowed to proceed from start to finish without fear of interruption, the task thus stated is impossible because "nudged" blocks make it virtually impossible to eliminate all gaps from the block structure. Thus, even after C has run to completion, the free space will, in general, consist of some number of free blocks smaller than about 100_8 and one big free block containing what's left. Schemes to avoid the unpleasant situation where someone has made a request for which there is ~~suffi~~ sufficient free space but it can't be coagulated into a large enough block will be discussed later.

C's problem is vastly clouded by demands from various people that they be allowed to do something while C is trying to fix up ECS. There seem to be four types of code clamoring for special treatment:

1) The interrupt code. We don't want to lock up interrupts for the second or so that complete compactification is likely to require. Fortunately, this is easily handled because only event channles and process descriptors need be in shape for the interrupt code. Thus, the interrupt code can set I.WAIT and C can pause briefly to let the interrupte code run in the usual manner, with no great special manipulations.

2) Another CPU. Without going into detail, access to ~~x~~ most objects should be easy to engineer. ~~XXXXXXX~~ Creations and deletions seem difficult.

3) A special process (SPEED FREAK). A process with a demand for fast response might be allowed to run while C was dormant and ECS wasn't really in good shape, if the process didn't make certain demands on ECS structure, in particular, creations or deletions. C could then be reawakened to finish its work.

4) Any old process. This fragments into two situations:
   a) We let C accumulate at least enough free space to satisfy the demand of the process ~~of~~ which is waiting for space, but we don't necessarily let C compact all of ECS. Thus, the demanding process is gotten off ~~b)~~ our backs in the normal manner, while other processes are perhaps spared annoying delays.
   b) We just let C do some fixed amount of collecting. If there isn't enough space to satisfy the process which precipitated the compacting, it is put in some abnormal state and other processes are allowed to run, thus guaranteeing that other processes ~~xx~~ (which don't make any demands on the allocator) are spared annoying delays.

Another use of the compactor would be to collect a little each time through the idle loop.

Of these, only 1 has received much thought, and fortunately it is easy to make it work. No special facilities for implementing the restricted access refered to in 2 has been provided (or even thought about). 4a presents no special problems to the system, as there are no funny processes to worry about; one just has to provide a C which knows how to quit before it's finished and worry about whether or not it's sensible/efficient/practical/good/bad/disastorous to do partial compactification. (C has been provided with the appropriate handle, the "N-CONTROLLER", which ~~it~~ is not used, currently).

From the point of view of C, 4b ~~is easier than~~ could be handled as easily as the less powerful 3, because C could simply stop compacting and clean things up and quit, leaving someone else the task of doing the right thing for any processes left in an awkward state.* Future calls of C would just start form scratch. 3 represents a half-way house, intended to minimize the pain of the awkward state processing. The process P, which precipitated compacting, is suspended. The special process SF is run under restrictions that assure that SF won't make untoward demands on ECS (these restrictions and their implementation are discussed in the SPEED FREAK document, soon available). Then P is restored, C is fired up and completes, P is in a normal state again and the system doesn't have to worry about it. Much.

Let me try to say it again: 3 is to make it feasible to guarantee fast response to certain special processes; 4b is to avoid delays to normal processes (which don't make demands on the allocator).

The "M-CONTROLLER", described below, makes 4b processing possible without changes to C or the structure of ECS. All that is needed is someone with the energy to engineer the "awkward state" process stuff.

3 could also be implemented using the M-CONTORLLER, but in the interests of efficiendy/speed, the I.COOL mechanism has been provided - whenever C finds the cell I.COOL ≠ 0, it cleans up ECS and takes a special exit as soon as possible.

* I must note that many of the facilities envisioned for running processes in mid-compactification are invalidated by problems in the file block allocation mechanism which are still under discussion/design.

COMPACTING STRATEGY

1) A suitable starting address is found (usually the bottom, see the descriptions of the N and M controllers).

1.5 I.LOCK is set.

2) The first free space above the starting address is located and objects above that are slid down one at a time. A new free chain and new ECS statistcs are accumulated and the old ones are revised.

   a) "Nudged" blocks. These are located at the next available address whach is a mult of 100_8. Space will in general exist between the beginning of the nudged block and the end of the xxx preceding block. It is incorporated in the new free chain.

   b) "Small" blocks (less than I.FIT). An attempt is made to put these into slots in the new free chain. If no fit, handled as in c.

   c) Other blocks. These are just slid down.

3) After each object, I.WAIT is examined. If it's non-0, I.LOCK is cleared and C waits for the interrupt as usual. After the interrupt, I.LOCK is set again and compactification continues.

4) Termination. If N or M is satisfied, or if I.COOL gets set, C patches the old and new free chains together and puts the new ECS free space statistics back. It leaves the free chain pointer pointing at the free block that it has constructed, ignoring the small ones caused by nudged blocks. If I.COOL caused termination, a special exit is taken (currently to DISASTER); otherwise exit is to the caller.

Cases

FREE A1 neither adj block free
A2 prec " " "
A3 foll " " "
A4 both " " "

A5 free a block with slop

A6 free a block when the free chain is null

make 10 contig objects, 1, 2, 3, ..., 10
10* Doper C.CCLIST, K.MALLOC, I
K.CLIST1

A1- free 3
A2- free 4
A3 free 2
free 6
A4 free 5

Doper C.CCL , ,10

A5 free it

ECSFULL (an oper whereon parame
can be fixed to add 1 wd
at a time)

A6 free an obj

ALLOC A1 Block survives

A2 doesn't survive

A2.1 0 slop

A2.2 max slop

A3 GC type (screen for now)

100 wd obj; create & free

A1 allocate 50

A2.1 " "

free 50

A2.2 allocate 50-1P.MNBLK

COMPACT C1 1st avail space in FB

C2 1st " " " slop

compact

free object

C1 compact

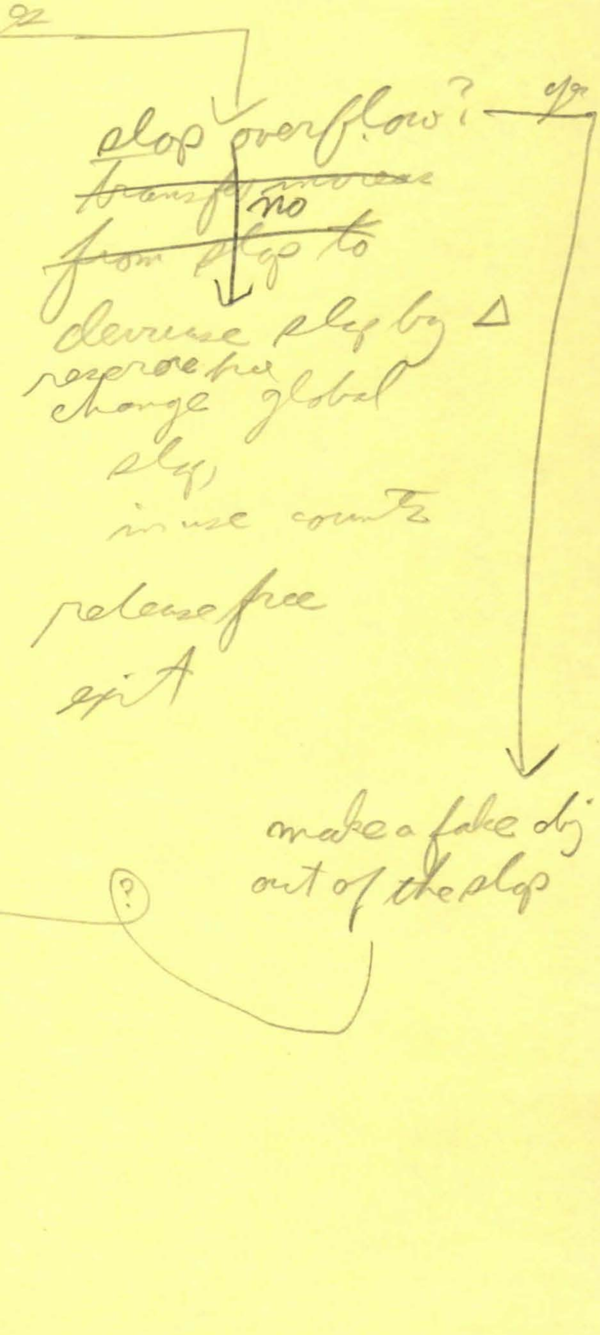
free object

allocate 1 wd smaller

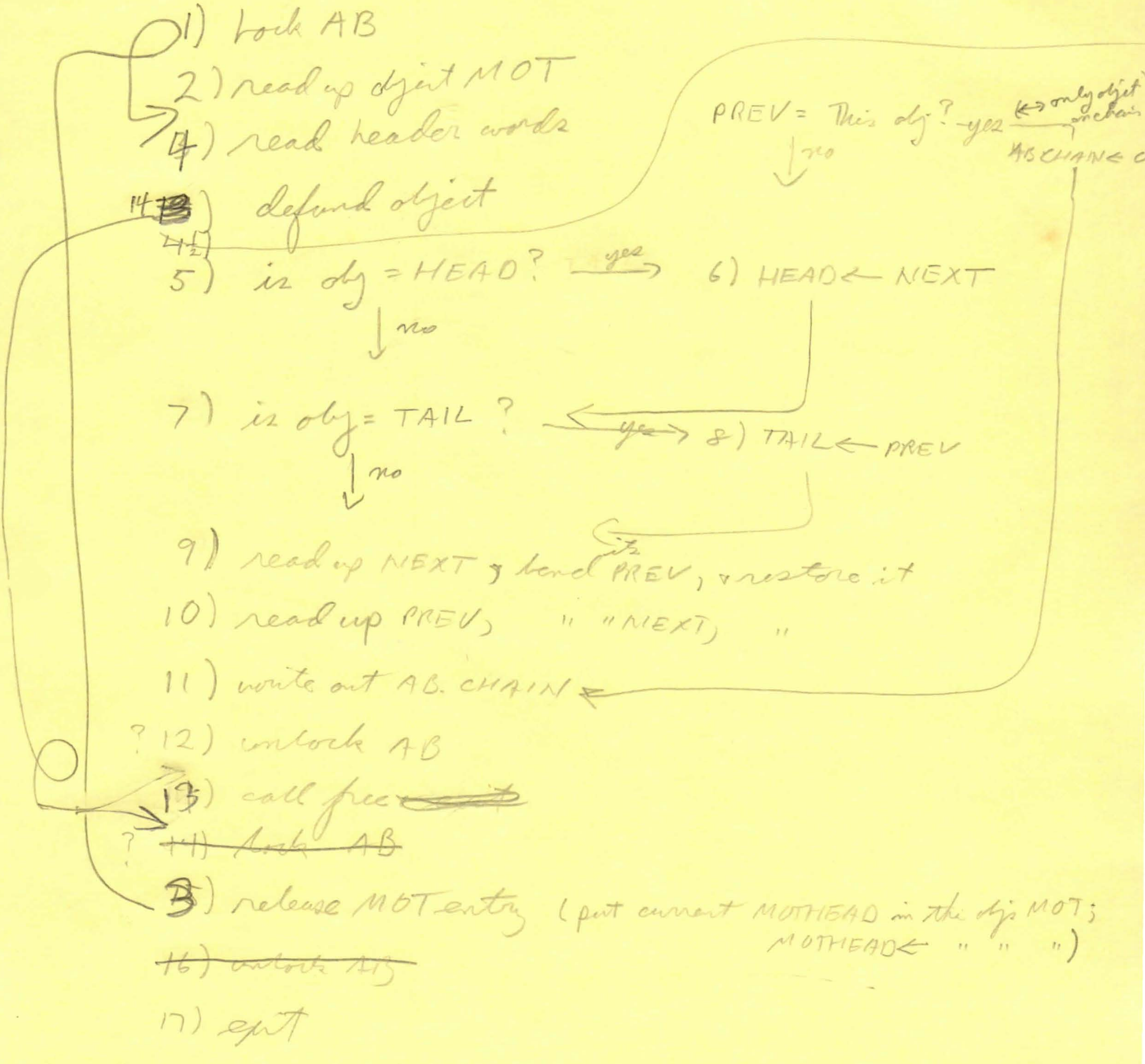
C2 compact

REALLOC ①

```
0  resorve, find increase Δ, ~~release~~
1  read PTR; read object header
2  enuf slop          yes
       ↓ no
3  Δ ≤ IP.PAD
   yes   ↓ no
4  ~~reserve~~, fund Δ-IP.PAD, ~~release~~
   (preserve old PTR)
5  call ALLOC to find a spot
6  copy the INFO
7  free the ~~old~~ block
8  Δ ≤ IP.PAD
   yes   ↓ no
9  ~~reserve, defend~~ —, release
10 exit
```



DELOBJ ○



1) Lock AB

2) read up object MOT

4) read header words

14 ~~13~~) defund object

4½)

PREV = This obj? — yes → only object on chain → AB CHAIN ← 0

no

5) is obj = HEAD? — yes → 6) HEAD ← NEXT

no

7) is obj = TAIL? — yes → 8) TAIL ← PREV

no

9) read up NEXT & bend its PREV, & restore it

10) read up PREV, " " NEXT, "

11) write out AB. CHAIN ←

? 12) unlock AB

13) call free ~~it~~

? ~~14) lock AB~~

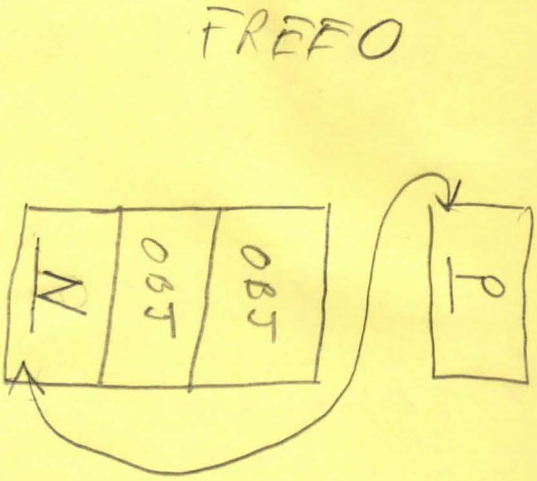
3) release MOT entry (put current MOTHEAD in the obj's MOT; MOTHEAD ← " " ")

~~16) unlock AB~~

17) exit

FREE

PTR



1 freeze FSINFO & read it in

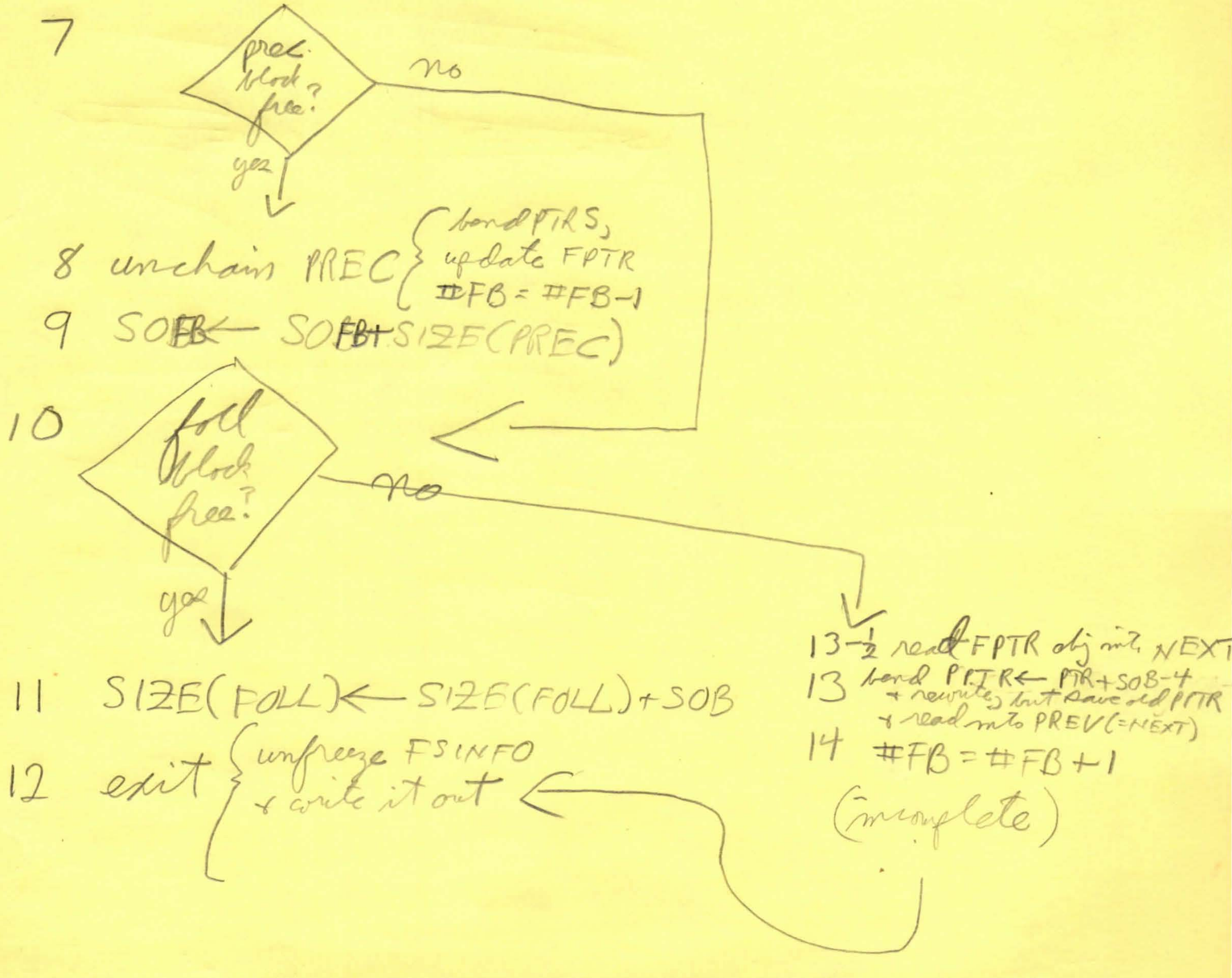
2 #obj ← #obj-1

3 FS ← FS + SOB

4 SLOP ← SLOP + SOB - FNDSZ

5 SOFB ← SOB

~~6 WHERE ← PTR + SOB - 4~~



8 unchain PREC { bend PTRS, update FPTR, #FB = #FB-1

9 SO~~FB~~ ← SO~~FB+~~ SIZE(PREC)

11 SIZE(FOLL) ← SIZE(FOLL) + SOB

12 exit { unfreeze FSINFO & write it out

13-½ read FPTR obj into NEXT

13 bend PPTR ← PTR + SOB - 4 & rewrite, but save old PPTR & read into PREV (=NEXT)

14 #FB = #FB + 1

(incomplete)

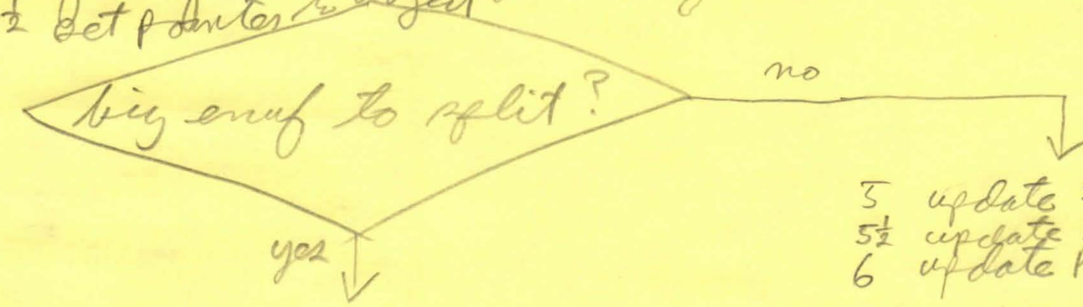
ALLOC

1 freeze the free space info & read it in

2 update # objects
2½ " free space

fix to indicate compact → 3 find first FB large enough to accomodate new object
3½ set pointer to object

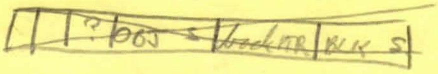
4



5 update #FB
5½ update slop, free space
6 update PTRs in prev & following FB
6½ set prv block not free in next bl
~~7 set block size = FB size~~ set slop
7½ set initial FB to next one.

8 change size of FB
9 rewrite FB leader
~~10 set block size =~~ set slop = 0
11 set initial FB to this one

12 write out free space info & thaw it
13 write header for object



14 clear remainder of object
15 exit

MAKEOBJ

0

OOD

1) lock allocator info ~~& read it in~~

2) size ← size + IP.AWROS
FUND
3) read EC.ABPCK &
get MOT slot & unique name
(if EC.ABPCK = 0, MOT desaster)
(if " + 1 ≥ 2^{36}, UN OV ")

4) write MOT entry UN | 0 , X5 = MOT entry

5) write EC.ABPCK &
unlock ALLOCBLK

? 5½) form capability in CAPAB+0, +1

6) ALLOC

7) lock allocator info ~~& read it~~ to diddle the AB

8) read the AB

9) objects already on chains (PTR wrd ≠ 0) —no→ 12) set HEAD = ~~TAIL~~ = MOT of new obj
yes ↓ (yes)

10) bend TAIL PTR to new object ←?

11)

TYPE	MOT	HEAD	TAIL

13 exit via putcap

Allocation of ECS

What is ECS?

The ECS space available for use by the system is defined by two ~~system parameters~~ numbers assembled into ~~the~~ system as S.ECSRA + S.ECSFL. These are, by definition, the first usable cell + the total number of usable cells respectively. Due to hardware limitations, the must both be multiples of 100_8*.

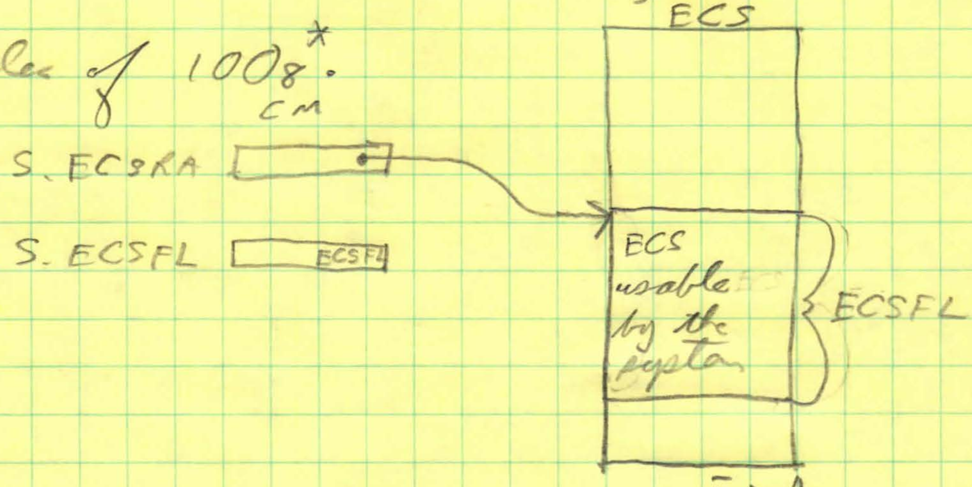


Fig 1

These quantities are usually invisible since hardware registers supervise actual ECS addressing; we are only concerned with them when setting up the ECSRA + ECSFL fields of exchange jump packages. Thus we ~~prefer~~ describe ECS below as though it runs from address 0 to address ECSFL-1 inclusive.

* ~~if the assembled values I believe that~~ The low 6 bits are ignored, in case values not mults of 100_8 are assembled.

How is ECS allocated?

ECS space is organized as follows:

A) Dedicated system space (in low ECS) (including the MOT, free space pointers, non-resident system code, etc.)

B) Any space left over for user files, ~~code~~ processes & other details incidental to letting people use the machine), organized ~~into blocks~~ roughly:

1) Space in use, accounted for ~~on some ABs~~ in ABs, ~~(space actually currently occupied by something)~~ containing object blocks of one ~~of the~~ or another of the following types:

a) allocation blocks
b) capability lists
c) processes
d) files
   i) pointer blocks (these are not strictly
   ii) data blocks objects; ~~as explained~~ see below)
e) event channels
f) operations

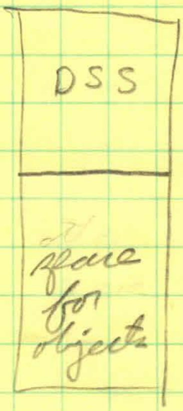


fig 2

2) Free blocks, where newly created objects can be put. ~~This is in the form of a circular chain of free blocks.~~

3) Slop, which is basically space which its difficult to account for, as explained in the description of the ALLOC routines

The objects listed in 1) are the real denizens of ECS & their creation, access, & destruction is the business of the ECS system code. To facilitate access control & relocation, access to objects is thru the Master object Table (MOT).

Now P9, as modified etc

ALLOCATION OF ECS

The lower portion of ECS contains system code and certain other specialized system cells. The remainder of ECS is divided into blocks of three varieties: objects, free blocks, and file blocks.

MOT

~~The Master Object Table (MOT) is located in low ECS and~~ The MOT contains an entry for each object in ECS (see Figure 3). Each entry occupies one cell and contains a pointer to the object as well as the "unique name" associated with the object. Except for the special case of "~~direct access~~ DAE" all references to an object are made through the MOT entry. The unique name must be checked against a "unique name" provided by the user in his capability before allowing access to the object. This insures protection even after an object has been deleted and the MOT entry has been reassigned. The pointer in the MOT enables the relocation of objects during garbage collection. compacting of ECS.

~~how big MOT? where insert empties?~~

The unused entries in the MOT are linked in an "available space list", to which a pointer is maintained in ECS at EC.ABPCK. The next available "unique name", issued serially, is kept at EC.ABPCK+1. A system disaster occurs when the MOT "available space list" is exhausted or the next available unique name exceeds $2^{39} - 1$.

~~Objects are the true residents of ECS and are classified as: Allocation Blocks, Capability Lists, Event Channels, Files, Operations, or Processes.~~ Each ~~of these~~ type of object occupies one block except files, which constitute a tree structure of blocks. The root of this tree is the file descriptor, the actual object. The leaf nodes (data blocks) and the other nodes (pointer blocks) are classified jointly as file blocks. Each file block is located by a single pointer, guaranteeing ease of relocation for file blocks as well as objects.

Each contiguous portion of unused space in ECS forms a free block, which is linked into a two-way circular list. Pointers to this, the Free Chain, are maintained ~~in two cells~~ at ~~EC.APACK.~~ (See Figure 4.) ~~The free space is kept in EC.APACK+2~~ EC.FINFO prow Y.P.

Allocation Blocks

The Allocation block is the object which regulates ECS allocation and CPU usage. An Allocation block can be provided with ~~a sum of money and~~ a portion of ECS space, which can only be obtained from another allocation block. ~~Every~~ Each object is associated with an allocation block; ~~these~~ all objects associated with a given AB are linked to the allocation block in a two-way circular list. The allocation block heads this list, and each object has a backpointer to its allocation block. The objects of ECS, therefore, form a tree. The root of this tree is the Master Allocation Block which is created at initialization and provided with ~~an infinite amount of money, and~~ all of ECS.

The allocation block will be ~~billed for CPU-time used by its descendant processes, and will be~~ charged rent on the ECS space occupied by its descendant objects. FUND is the routine which charges this rent and must be called whenever the size of a descendant object is to be changed. It must also be called periodically to prevent deficit spending. ~~As of this writing, policy decisions are pending regarding allocation blocks (e.g., what to do if an allocation block runs out of money).~~ {Write up the current accounting philosophy}

FUND is called with an allocation block, and an increment to ECS space. It compares the master (S.MASTR) clock with the "time of last bill" field, updating the latter, and charging rent for the interim on ECS space in use. "ECS in use" and ~~"$ used for rent"~~ ECSMSWDS Time*space are updated. "ECS in use" cannot exceed ~~"Allocated ECS"~~ "Charged ECS" ~~and "$ used" cannot exceed "$".~~ (See Figure 5.)

redo

FUND has three entry points:

FUND - ~~B2~~ X7 ... Increment to ECS space
B3 ... Return link
X5 ... 2nd word of capability for alloc bk

~~FUNDX7 - B3 ... Return link~~
~~X5 ... 2nd word of capab. for alloc bk~~
~~X7 ... increment to ECS space~~

FUNDB - B3 ... Return link
AØ ... S.ABLOCK - (allocation block in S.ABLOCK)
XØ ... ECS address of alloc bk
X7 ... increment to ECS space

probably will change

Block Manipulation; General

At system initialization time,

~~1)~~ the system dedicated storage is set up, leaving remaining space = INISPACE, starting at EC.FOBJ.

2 0-length FB's are constructed at the beg + end + a FB contain the rest is put betw

~~2) 2 ze~~

~~+ the remaining space which is ~~handled~~ divided as follows:~~

~~1) INISPACE = amount of remaining space~~

~~1) 2 "zero length" free blocks (~~two~~ to assure the the free chain ~~will~~ always have something to point to)~~

~~2) ~~as many max [illegible] size free blocks as possible~~ a FB consisting of everything else~~

~~3) A final smaller FB to occupy the remaining space~~

Then the Master Allocation Block is formed (using ALLOC, actually) + given ~~all the~~ a space of

? IP:PAD

INISPACE - EC.PAD. It is not quite a normal object in that it is charged for the space that it occupies + has no header word.

After that, block structure in ECS is in the hands of 4 routines:

page (1)

Block Manipulation, General

At system initialization time the following blocks are created: the Master Allocation Block, two zero-length free blocks, and several free blocks (max. size = $2^{17} - 1$) consisting of the rest of ECS. After that, block structure of ECS is in the hands of four routines:

ALLOC creates a block of specified size

The free chain is scanned for a block of sufficient size. If none is found, a compaction ~~garbage collection (GBGCOLL)~~ is called. Otherwise, a determination is made whether the free block is sufficiently larger than the requested size to justify splitting it up. If so, the new block is taken off the end ~~beginning~~ of the free block, whose size field is updated. If not, the entire block is used and is removed from the free chain.* The allocator's word is written and a pointer to the block is stored at a caller-specified cell. Finally, the block is zeroed.

DAE stuff

On entry: X7 ~~B2~~ - size of block (block size)
B3 - type of block (1=pointer block; 0=data block or object)
B7 - return link
X5 - ECS address of pointer to be set.

REALLOC changes the size of a block (always an object)

big rewrite, yet to be done

First it is determined if a new block will be required (it will not be if the increment is negative or less than the slop). If not, FUND is called with the increment, and the "size in use" field is updated. Otherwise, FUND is called with the total size of the new block, and ALLOC is called to find the block. FUND is again called to defund the original block (without this double call, a system diaster would occur if ECS were saturated). The contents are transferred from old block to new, FREE is called to release the old block, and the pointer in the MOT entry is updated.

On entry: X1 - increment
X2 - MOT index of object
X6 - return link

FREE inserts a block into the free chain

The block is merged with either or both adjacent blocks when they are free. The pointer to the block is zeroed.

On entry: B7 - return link
X5 - ECS address of pointer

GBGCOLL~~, when written, will~~ compacts the block structure.

* space in excess of the necessary size is not charged to the AB but disappears from F.L. + becomes slop. The ~~garbage~~ compactifier has to strip any slop off of objects during compactification.

Object Creation and Destruction

MAKEOBJ creates an object

FUND is called; an MOT entry + unique name are ~~is~~ created; ALLOC is called. A capability for the object (all option bits set) is created and stored in "CAPAB". The list associated with the father alloc bk. is updated. {The header word is written}.

On entry: B2 - size of object to be created (actual size, ~~=~~ one less than size kept in block by allocator)
B4 - return link
X5 - 2nd word of capability for alloc bk (father)
X7 - type of object
On exit: X5 - address of first usable word

DELOBJ destroys an object

The father allocation block is found, and the object is removed from its list. FUND is called to defund the space; FREE to release it. The MOT entry is added to the MOT free list.

On entry: B7 - return link
X5 - 2nd word of capability for object to be deleted

~~File Block Creation and Desctruction~~

~~MAKEFIL creates a file block~~

~~It calls FUND and ALLOC only.~~

~~On entry: B6 - type of block (1 - ptr blk; 0 - data blk)~~
~~B7 - return link~~
~~X5 - 2nd word of capab. for alloc bk.~~
~~X6 - ECS address of ptr to new block~~

~~RTRNFIL deletes a file block~~

~~It calls FUND and FREE.~~

~~On entry: B7 - return link~~
~~X5 - 2nd word of capab for alloc bk~~
~~X6 - ECS address of pointer~~

Miscellaneous Routines

Four ECS actions:

NEWUN changes a unique name

This is the system "Indian-giver"

will change

AP1 = D : C-List Index of Object whose unique name is to be changed.

CREALBK creates an allocation block

AP1 = C : Father alloc bk
AP2 = D : Index for new capability

CCCLOA constructs a capability (all option bits set) for the newest-born child of the alloc bk.

AP1 = C : Allocation block
AP2 = D : Index for new capability

DONATE transfers space and money from one alloc bk to another

redo

AP1 = C : Alloc Bk (DONOR)
AP2 = C : Alloc Bk (DONEE)
AP3 = D : ECS space to be transferred
~~AP4 = D : Money to be transferred~~

SETLIM

actually goes earlier (p 11+)

~~Nudge moves a file data block so that the first usable word is on a 100_8 word boundary & sets the DAE bit on the block~~

~~on entry B7 = return link~~
~~X5 = ECS address of PTR (will be reset to new location)~~
~~X? = 2nd wd of cap for alloc block (reqd only if the block occupies more than ECPAD cells)~~

Figure 3a MOT entry

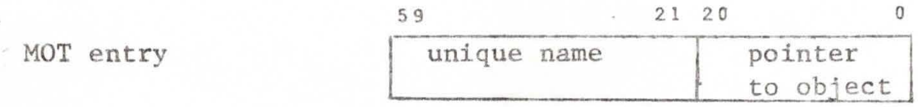


Figure 3b MOT available space list

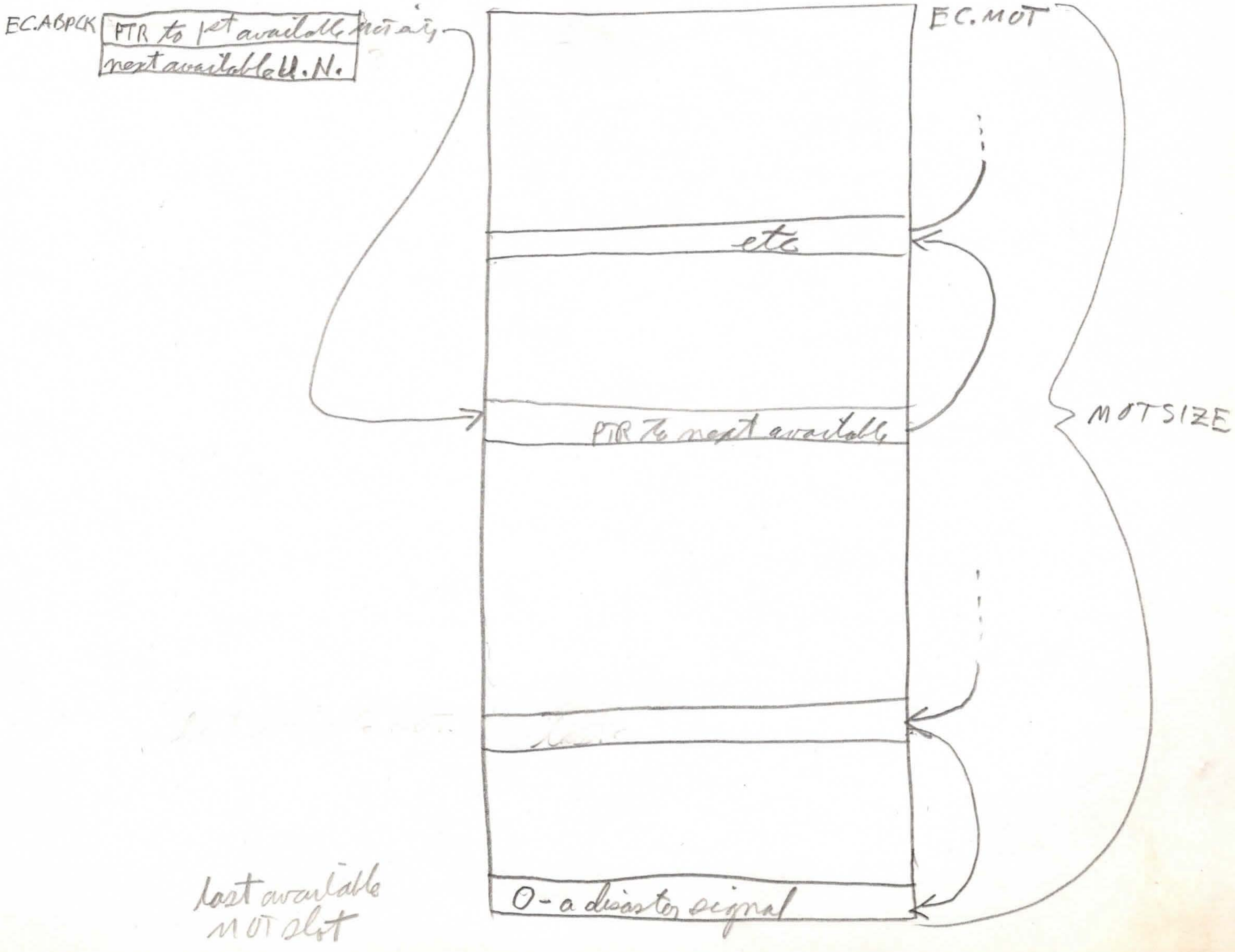


Figure 4

no

*

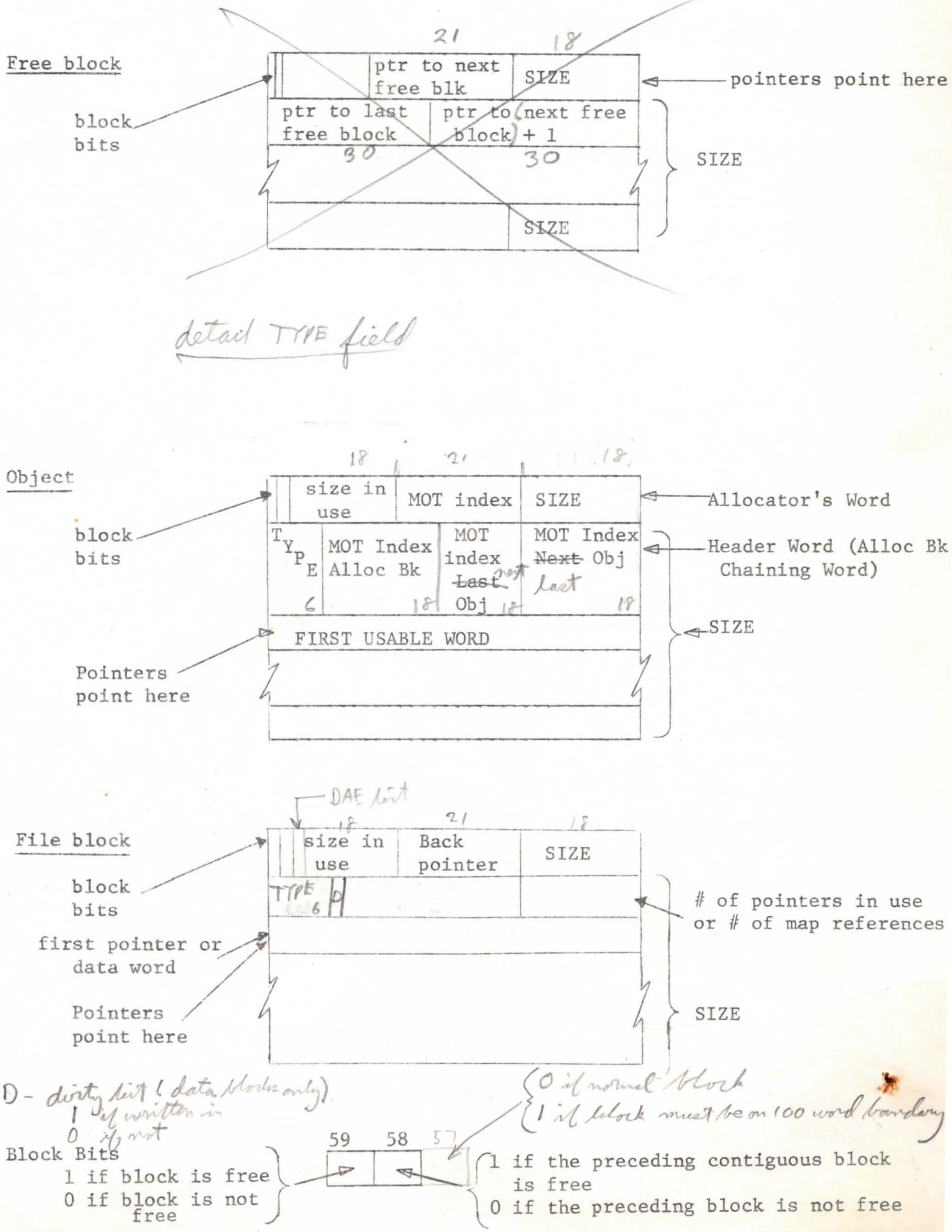

Free block
21
18
ptr to next free blk
SIZE
pointers point here
block bits
ptr to last free block
ptr to (next free block) + 1
30
30
SIZE
SIZE
detail TYPE field
Object
18
21
18
size in use
MOT index
SIZE
Allocator's Word
block bits
TYPE
6
MOT Index Alloc Bk
18
MOT index ~~Last~~ next Obj
18
MOT Index ~~Next~~ Obj last
18
Header Word (Alloc Bk Chaining Word)
SIZE
FIRST USABLE WORD
Pointers point here
DAE bit
File block
18
21
18
size in use
Back pointer
SIZE
block bits
TYPE 6
D
# of pointers in use or # of map references
first pointer or data word
Pointers point here
SIZE
D - dirty bit (data blocks only)
1 if written in
0 if not
Block Bits
1 if block is free
0 if block is not free
59
58
57
0 if normal block
1 if block must be on 100 word boundary
1 if the preceding contiguous block is free
0 if the preceding block is not free

Figure ~~3~~ 5 Allocation Block

~~get from writeup~~

Allocation Block

→

~~Alloc Word (see object)~~	
~~Header Word (see object)~~	
~~Allocated ECS~~	~~ECS in use~~
oldest obj ~~pointers to AB~~ HEAD	newest obj ~~- chain~~ TAIL
~~time of last bill~~	~~$ $ $~~
~~$ used for CPU~~	~~$ used for rent~~

(0 when nothing is hung on the block)

MOTPTR →

ALLOCATOR'S WORD	
HEADER WORD	
RESERVED SPACE	SPACE IN USE
HEAD PTR	TAIL PTR
TIME OF LAST BILL	CHARGE FIELD
CONT T * S	
DISCONT T * S	
CP'ms available	
" " consumed	
MOT slots available	

(based on CHARGE field)
" " " "

1) Should we have special handling of the REALLOC problem?

YES : 1) User only charged for what he's using
2) otherwise, Everybody will have to have space for 2 process descriptors

NO : 1) Imposes a limitation on the size of
a) process descriptors
b) operations

Best of both worlds: use mystery space, up to limit; if too big, try to charge it to him twice & generate the normal error if that doesn't work. Handles ~~nudge~~ NUDGE too.

2) Variable ECS space

1) ~~new system operations? change ECSFL~~

(change all ECSFL's in various system XPACKS)

if $ECSFL_N \geq ECSFL_O$, o

make a new free block(s) & say "thanks".

if $ECSFL_N < ECSFL_O$,

compactify

if enuf, chop off free block(s) & say "well, OK"

if not, ~~either~~ say "tough" & either don't give back any, or give back as much as possible

how to change current XPACK ECSFL???

3) allocation blocks & CPU time

4) ECSRA & ECSFL aren't mults of 100_8.

think about the dirty bit

file blocks (& hence DAE stuff), is limited to 2^{16} (next lower power of 2 from $2^{17}-1$)

How to handle compactification

if a creation or process descriptor increase requires a block that is not in one piece.

i) fix the process so that when next run will ~~[illegible]~~ remove the current ecs request

ii) make sure the rest of ecs is still in good shape

iii) hang the process on a special event channel or make a list of it

~~(iv)~~

~~start~~ incremental compactification
and running other processes
unless one of them wants a block from ecs.
hang it on the channel also

System initialization:

largely hypothetical

A tape containing 3 distinct ~~sets~~ files of code (core images, suitable for execution) is loaded via a one card loader from the card reader. The first routine is a PPU program which completes the reading of the first 2 files & places the code in its appropriate locations in the PPUs & CP, sets up an EX.PACK. & ~~jumps~~ starts the CP at INITYZ in INITL.

Files: PPU, CODE
CPU, CODE
BEAD, CODE

INITYZ → goes to ECSINIT to set up ECS, complete with MAB & cap for same at S.SYSAB

sets up 4 XJ packages S.CHIP
S.ARITH
S.FSWAP
I.BOX

dummies up an XPACK so it can look like a user

creates an initial C-List

Capabilities and Capability-Lists

User access to all objects within the ECS system is controlled by capabilities. A capability identifies the object it refers to, specifies the type of the object, and the set of allowed actions on that object (options). Capabilities are grouped together in capability-lists (C-lists) which are themselves objects within the ECS system. Individual capabilities are referred to by their index within a C-list. Since the capability, residing in a C-list, authorizes access to an object, the user is never allowed to fabricate a capability*. The system creates a capability with all options allowed when an object is created. System actions are provided to permit the user to examine a capability, to copy capabilities between C-lists and within a C-list, and to downgrade the option mask (see System Actions). Thus, the user can transfer the right to access an object and can curtail that access, but he may never manufacture that right or increase the set of allowable actions on the object.

CAPABILITY

A capability consists of two 60-bit words (see Figure 1). The first word contains the type of the object to which the capability refers and a bit mask indicating the allowed actions on the object. The type field occupies the lower order 18 bits of the first word and must have exactly 9 of the 18 bits set to allow the testing of options and type bits with one instruction (the implication function). The remaining 42 bits comprise the option mask. The meaning of the bits in the option mask, of course, depends on the type of the object.

The second word contains the information necessary for the ECS system to access the object (or, in the case of a class code, the object itself). The system uses the low order 18 bits of the second word, which contain the master object table (MØT) index, and the high order 39 bits, which contain the unique name of the object. The remaining 3 bits of the second word are unused.

Capabilities are created by the allocation routines at the point when storage is allocated for a new object. The new capability with all options allowed is placed at CAPAB and CAPAB+1 by the allocation routines. The routine creating the new object then moves the capability to its user-designated position in the user's full C-list by calling PUTCAP.

* some actions create a capability (more or less) as specified by the user.

CAPABILITY LIST

A capability list (C-list) is a sequence of capabilities and "empty" positions (see Figure 2). It is prefixed by the total number of spaces for capabilities. "Empty" positions are simply two zero words. Each C-list is filled with "empties" upon creation.

A C-list is assigned to every subprocess within a process. (See Figure 4). For every process there is defined a sequence of subprocesses called the full path. Corresponding to the full path, the full C-list is defined as the concatenation of the C-lists belonging to the subprocesses in the full path. When referring to capabilities within the full C-list, the capability index is interpreted as if the C-lists in the full C-list have been joined to form one long C-list.

The full C-list is implemented by maintaining a full C-list table within the process descriptor (see Figure 3). The full C-list table is a sequence of two word entries each of which identifies a C-list and the length of the C-list. P.CLIST in the process descriptor holds a pointer (relative to the origin of the process descriptor) to the first entry in the full C-list table. The full C-list table is terminated by a zero word. The first C-list (called the local C-list) in the full C-list is copied into core with the process while the remaining C-lists remain in ECS. P.CTABLE, in the process descriptor, holds a pointer to the end of the full C-list table (the zero word), the number of entries allowed in the table (maximum length of the full path), and the size of the core buffer for the local C-list (maximum local C-list size).

Three routines are used to access C-lists. GETCAP is used to fetch a capability from the full C-list. PUTCAP copies a capability to the full C-list. If the capability falls within the local C-list, it is copied to both the ECS copy and the in-core copy of the local C-list. Finally, ARBCAP is used to copy a capability to or from an arbitrary C-list (not the full C-list).

Figure 1

CAPABILITY

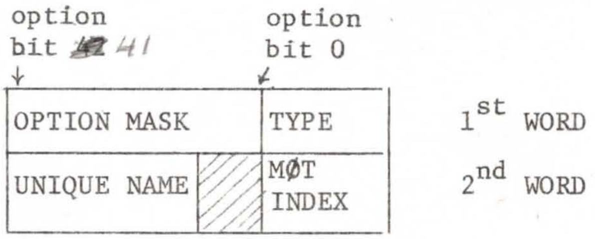


Figure 2

CAPABILITY LIST

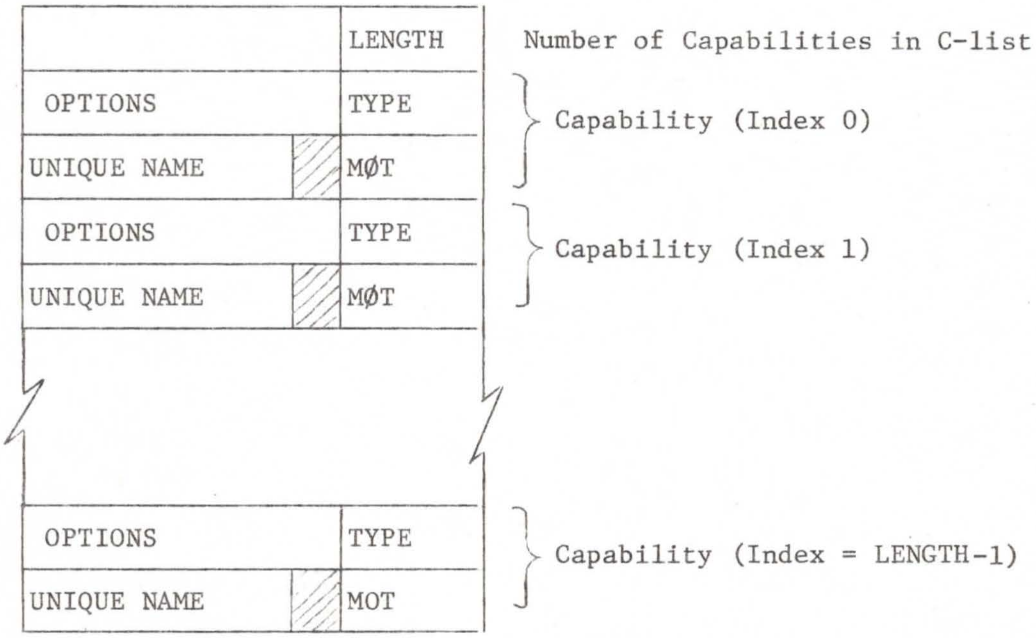
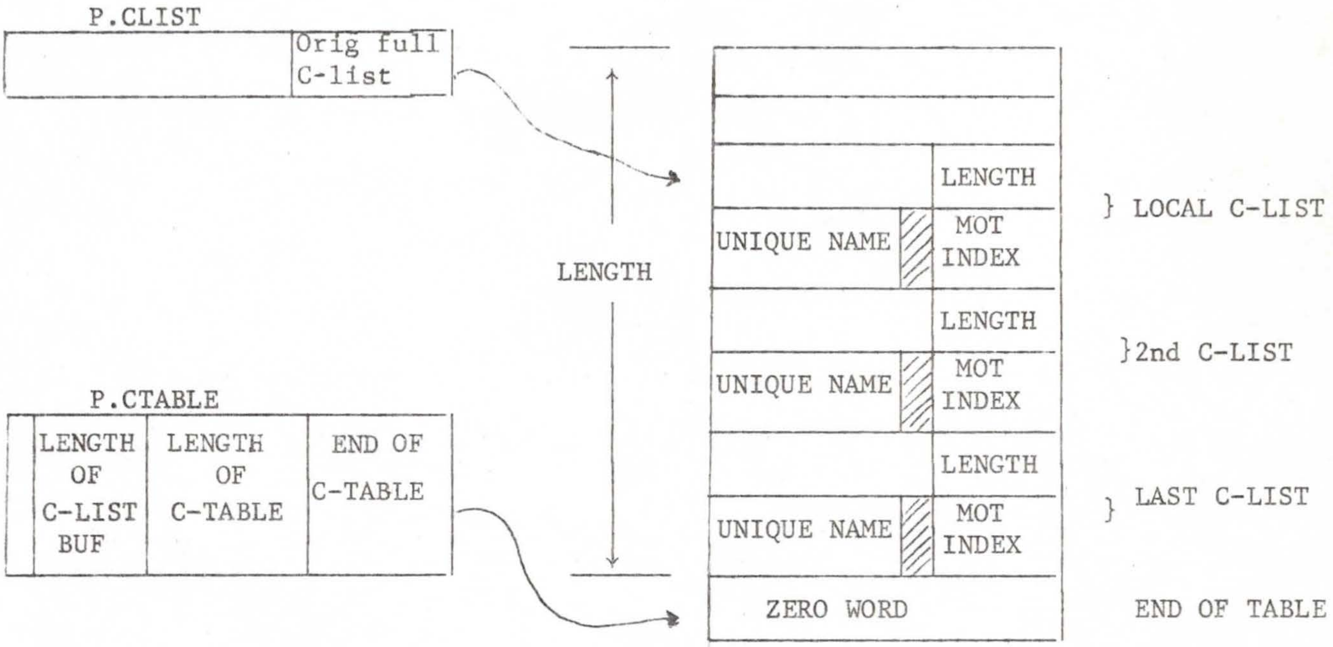


Figure 3

FULL C-LIST TABLE



SUBPROCESS DESCRIPTOR (C-LIST DATA)

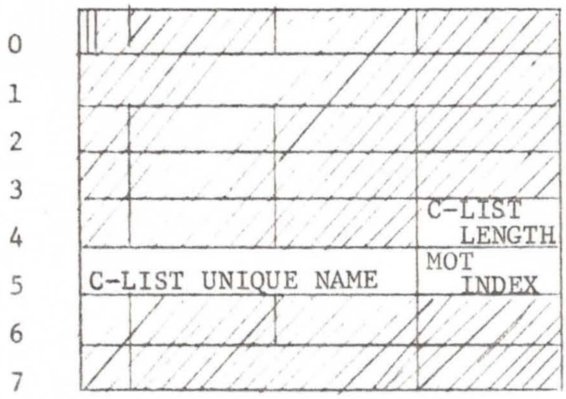


Figure 4

Files

A file is an ECS system object, containing a sequence of addressable (60 bit) words, used to provide storage for code and data. In order to permit a large file address space and, at the same time, make efficient use of ECS space, ECS files are organized in a tree structure. The "leaves" of the file tree are called data blocks and contain the addressable words of the file. The non-terminal nodes of the file tree are called pointer blocks (see Fig. 3) and contain links to either data blocks or other pointer blocks. With this tree structure, only the necessary pointer blocks and data blocks are allocated in ECS. Empty or non-existent portions of the file are not allocated until they are needed.

For any file, there is a sequence of positive integers, $(S_1, S_1, \ldots, S_n)$ $n \geq 1$, which describe the shape of the file. Each S_i, for $1 \leq i < n$, is the number of branches in the file tree at nodes of level i (the root of the tree is at level 1; all nodes connected to the root are at level 2; etc.). Each S_i for $i > 1$, must be an integral power of 2 (note: this does not apply to the first shape number S_1). The last shape number, S_n, is the size of the data blocks. Thus, the number of addressable words in a file is given by $L = \prod_{i=1}^{n} S_i$. The words of a file are addressed by integers which may range from 0 to L-1 .

The shape of a file is represented by the dope vector for the file and is stored in the file descriptor (see Fig. 2). The file descriptor is pointed to from the master object table (MOT). It contains the dope vector, the length of the file, a pointer to either a pointer block or a data block (~~zero~~ one level file), and the MOT index and unique name of the Allocation block which funds any changes in the ECS space occupied by the file. The dope vector contains instructions which are executed to obtain the path through the file tree which leads to a particular address within the file. When a file is created, only the file descriptor is constructed, and the file may be destroyed only when it is in this state.

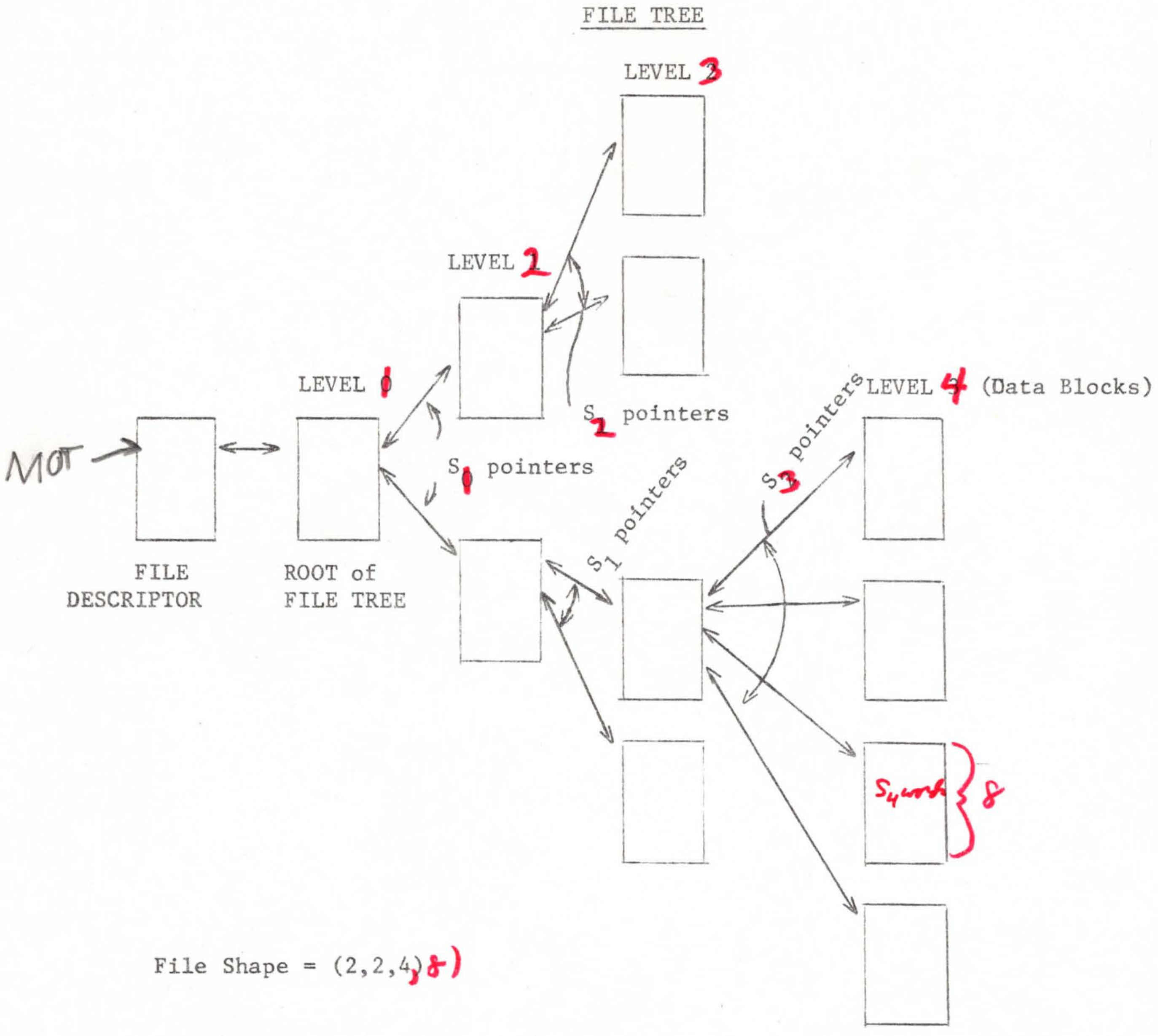
Pointer blocks (Fig. 3) link the file descriptor to the data blocks in all files with more than one shape number (n > ~~0~~1). Pointer blocks are constructed only when needed to link to data blocks. The allocation information which prefixes each block in ECS is used to provide a return path through the file tree. This backpointer contains the absolute ECS address of the single word which points to the pointer block (in the file descriptor or in a pointer block at the preceding level). A count of non-empty pointers within the pointer block is also maintained in the allocation prefix to the pointer block (note: the counter is greater than 0; otherwise, the pointer block is not needed). The word following the last pointer in the pointer block contains a negative number which is a relative pointer to the first word of the allocation prefix.

Data blocks (Fig. 4) contain the addressable words of the file. The count of maps (see Maps) which reference the data block is maintained in the second of the allocation words. A dirty bit is maintained on each data block. which can be tested & reset

File actions

When a file is created, only the file descriptor is formed. Data blocks may be subsequently added, one at a time, to hold data or procedures. When a data block is added to a file, it may also be necessary to create some or all of the pointer blocks between that data block and the file descriptor. Data blocks may also be removed and, again, one or more pointer blocks may be deleted if they are no longer needed to link to the remaining blocks in the file. A data block may not be deleted if it is referenced by an entry in some subprocess map (reference count ≠ 0).

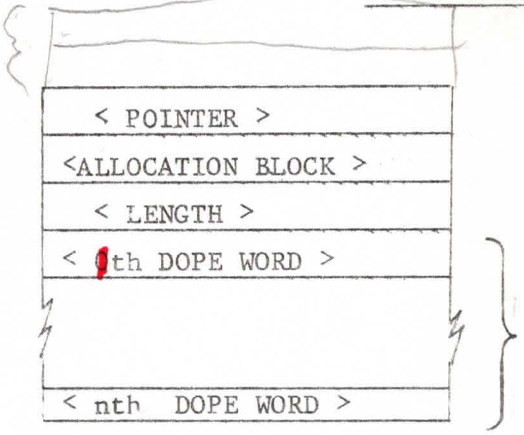
Files may be read and written. This action transfers words between the address space of the running subprocess and the data blocks of a file. If a transfer is requested which involves a file address corresponding to a non-existent data block, the transfer proceeds until the non-existent file address is encountered and then an FRETURN is initiated.



File Shape = (2,2,4,8)

Figure 1

FILE DESCRIPTOR



Pointer to Root of File Tree

Allocation Block Identification

File Length

Dope Vector

SHAPE = $(S_0, S_1, \ldots, S_n)$

< POINTER > : : =

+ 0

If root doesn't exist

or

12	6	18	3	21
0000	0	1	0	ABS ECS ADDR

If root is pointer block (n > 0)

or

12	6	18	3	21
1777_8	0	1	0	ABS ECS ADDR

If root is Data Block (n=0)

< ALLOCATION BLOCK> : : =

39	3	18
Unique Name	▨	MOT Index

< LENGTH > : : = (maximum file address) + 1 = $\prod_{i=0}^{n} S_i$

< 0th Dope Word > : : =

AX6 ℓ	MXO 0	JP B7

$\ell = \sum_{i=2}^{n} \log_2 (S_i)$

< jth Dope Word > : : =

AX6 ℓ	MXO m	JP B7

$\ell = \sum_{i=j+1}^{n} \log_2 (S_i)$

$m = 60 - \log_2 (S_j)$

< nth Dope Word > : : =

SX6 S	JP B7+4

$S = S_n - 1 \quad (n > 0)$

or

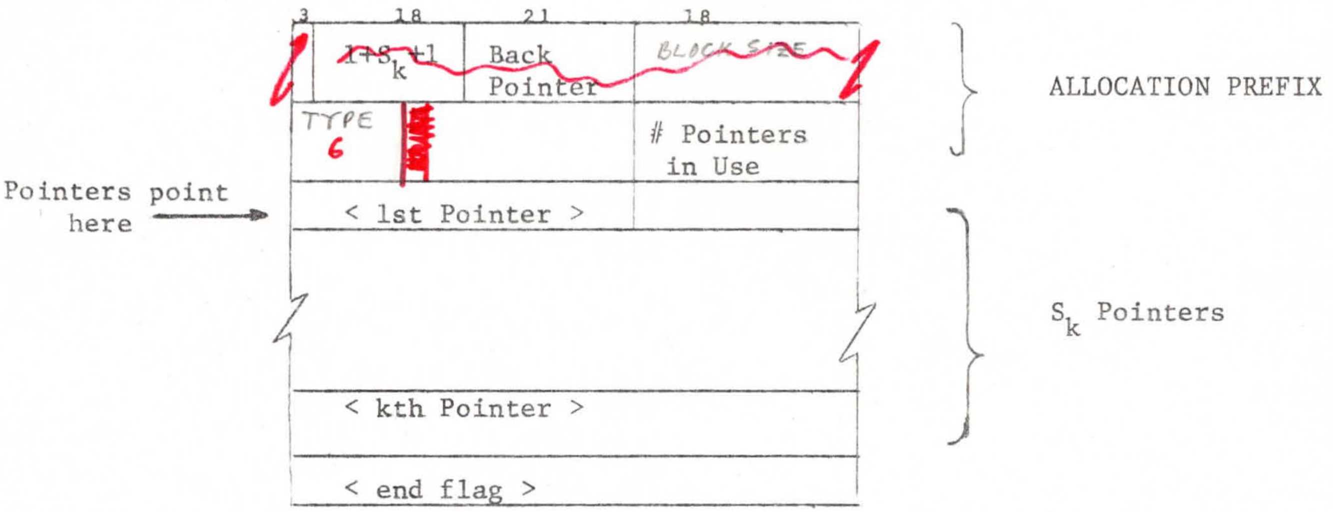
SB5 S	JP B7+5

$S = S_0 \quad (n = 0)$

Figure 2

POINTER BLOCK

Pointer block at level k



Shape = $(S_0, S_1, \ldots, S_k, \ldots, S_n)$

		12	6	18	3	21	
< jth pointer > ::= ($j \le S_k$)		+ 0					Corresponding pointer or data block doesn't exist
	or	0000	0	j	0	ABS ECS POINTER	Corresponding pointer block (k < n-1)
	or	1777_8	0	j	0	ABS ECS POINTER	Corresponding data block (k = n-1)
< END FLAG >	::=	$-(S_k + 1)$					Relative pointer to first allocation word

[illegible]

Figure 3

DATA BLOCK

Shape = $(S_0, S_1, \ldots, S_n)$

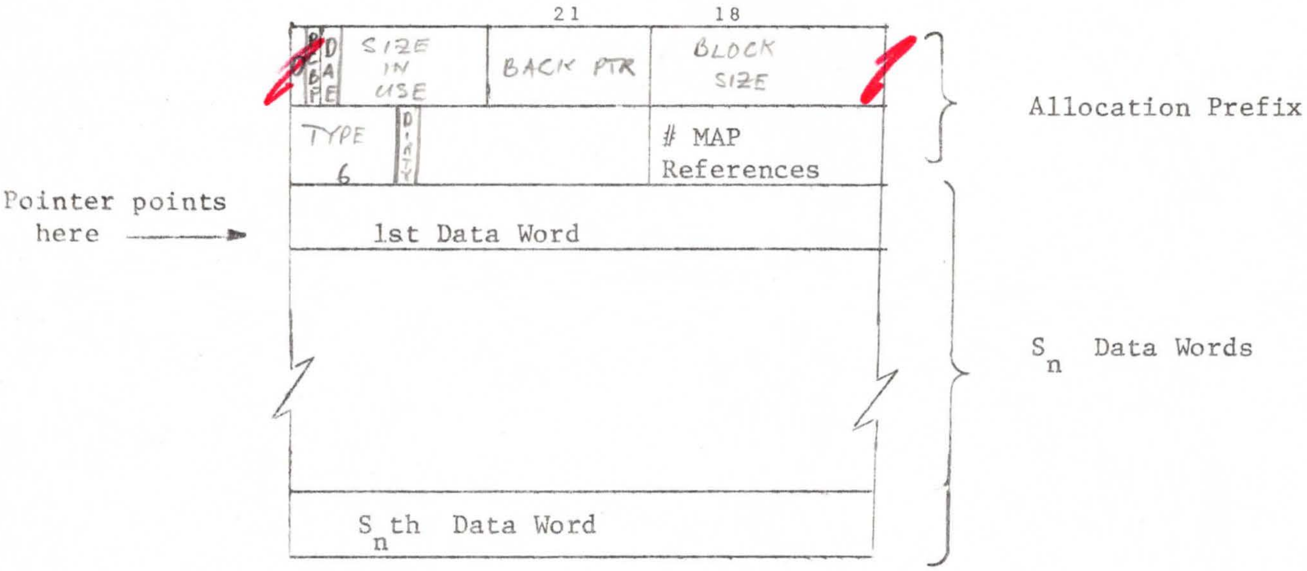


Figure 4

$$\text{PCBF} = \begin{cases} 0 \text{ if the preceding contiguous block is not free} \\ 1 \text{ " " " " " is free} \end{cases}$$

$$\text{DIRTY} = \begin{cases} 0 \text{ if the block needn't be written out by the disk system} \\ 1 \text{ if it should be} \end{cases}$$

Processes

Processes are the active elements of the ECS portion of the time sharing system. Only within the context of a process may code be executed and system actions initiated. A process consists of a set of central registers (exchange jump package), a set of subprocesses organized in a tree structure, a call stack recording the flow of control among the subprocesses, and a set of state flags describing the state of the process.

Swapping: Periodically, a process with its running flag set (see below) will be swapped into CM to run on the CPU. When this occurs, the process descriptor and local C-list are read in, and the entries in the full process map are swapped in from the indicated files in ECS to the indicated regions in CM. The exchange jump package of the process is loaded into the central registers of the CPU and the CPU is allowed to compute for awhile or until the process hangs. Then the central registers of the CPU are copied to the exchange jump package of the process, and the process is swapped out.

Process Descriptor

The data necessary to maintain and run a process are gathered together in the process descriptor, which is stored in two sections: the fixed length process descriptor and the variable length process descriptor. These two sections of the process descriptor are copied into CM when the process is being run on the CPU. While the process resides in ECS (See Figure 1), the fixed length descriptor and variable length descriptor are separated by the process queuing word buffer (see Event Channels). Information about the size of the queuing word buffer is contained in the first word of the process descriptor (P.ROHEAD). Data necessary to access and move the variable length descriptor are contained in the second word of the process descriptor (P.ROHEAD + 1).

When the process descriptor is copied to CM to run the process on the CPU (see Figure 2), it is preceded by a scratch area (used by the system while performing system calls) and the actual parameter area used to pass the parameters of system calls (P.PARAM). In addition, the local C-list is copied to CM following the fixed length descriptor and preceding the variable length descriptor. All pointers within the process descriptor are computed relative to the beginning of the scratch area. The absolute CM address of the scratch area is maintained by the system in S.USRB1 in system core and in B1 of the system exchange package.

The fixed length process descriptor is divided into the read only descriptor and the read write descriptor. The read only descriptor may not be modified without locking out the PPU interrupt system (I.LOCK). It contains (see Figure 3) the state flags of the process, process interrupt information, and process scheduling data. The read/write portion of the fixed length descriptor contains the process exchange jump package, data and pointers used to access and modify the variable length descriptor, and a few words of global process data.

The variable length process descriptor (see Figure 4) contains the full C-list table, the call stack, the subprocess descriptor table, logical map and error selection mask (ESM) storage, and compiled map storage. Organization of the variable length descriptor is maintained by pointers and values in the fixed length descriptor. When the process is in CM running on the CPU, the variable length descriptor is separated from the fixed length descriptor by the local C-list buffer, which is large enough to contain the largest C-list assigned to any subprocess in the process. Both the call stack and subprocess descriptors contain pointers into the variable length descriptor. These pointers, like those in the fixed length descriptor, are relative to the origin of the process scratch area (P.SCR).

PROCESS DESCRIPTOR (IN ECS)

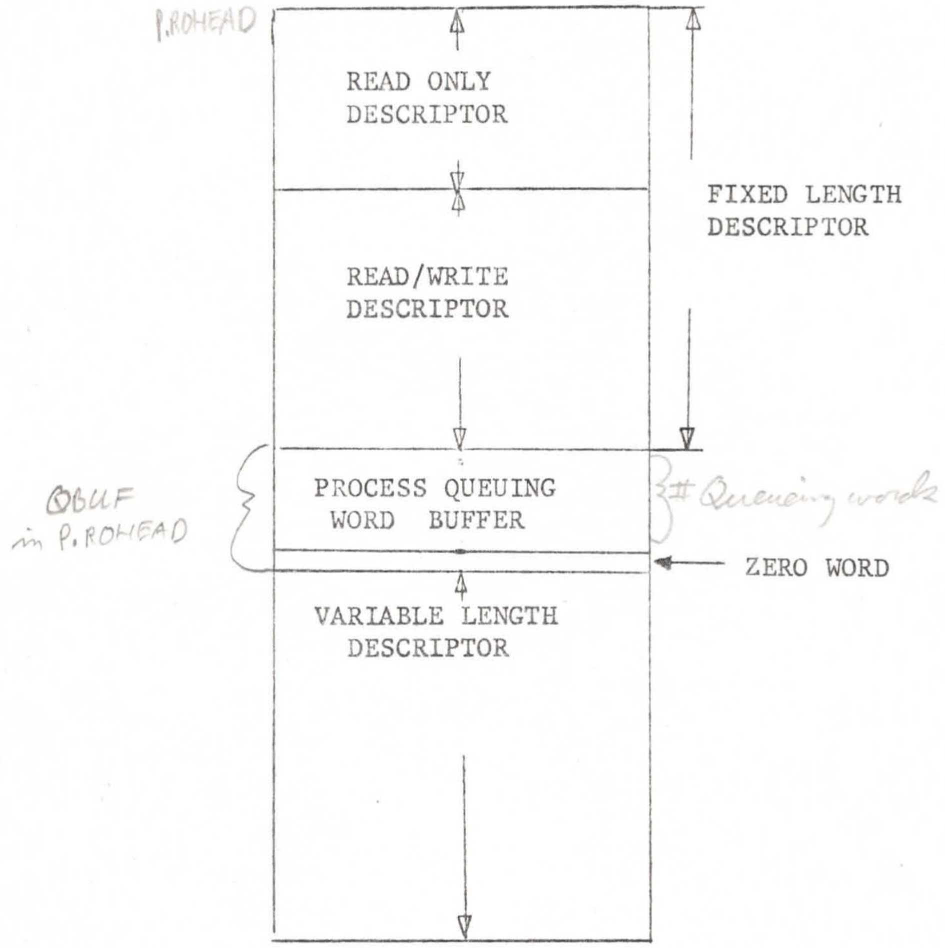


Figure 1

```
         LIST     -L
*
P.PARAML EQU      40B
P.SCHEDL EQU      5
P.SCR2L  EQU      6
*
         ORG      0
P.SCR    BSS      P.SCRL
*
P.SCR2   BSS       P.SCR2L
P.TEMP1  BSS       1
P.TEMP2   BSS      1
P.TEMP3  BSS       1
P.TEMP4  BSS       1
P.TEMP5  BSS       1
P.TEMP6  BSS       1
P.TEMP7  BSS       1
*
         BSS      1               DEAD CELL BEFORE ACRUAL PARAM AREA
P.PARAM  BSS      P.PARAML
P.PARAMC EQU      *-1
*
*
*                  PROCESS READ ONLY DISCRIPTOR
*
P.ROHEAD BSS      2
P.SCHED  BSS       P.SCHEDL
*
*                  SCHEDULER CLOCKS
*
P.USRTIM EQU       P.SCHED+1
P.SYSTIM EQU       P.SCHED+2
P.SWPTIM EQU       P.SCHED+3
*
P.PROCRO EQU      *-P.ROHEAD
*
*
*                  PROCESS READ / WRITE DISCRIPTOR
*
P.RWHEAD EQU      *
P.XPACK  BSS      16
P.CLIST  BSS      1
P.CTABLE BSS      1
P.STACK  BSS      1
P.SUBPDT BSS      1
P.MAPESM BSS      1
P.MAPSIN BSS      1
P.OLDP   BSS      1
P.INTERR BSS      1
P.IPLIST BSS      1
         BSS      1               DEAD CELL BEFORE LOCAL C-LIST BUFFER
P.PROCRW EQU      *-P.RWHEAD
P.LOCALC EQU      *
*
         USE      *
*
*
PS.TEMP  SET      P.PARAML/60
         IFNE     P.PARAML,PS.TEMP*60,1
PS.TEMP  SET      PS.TEMP+1       . ROUND UP
PS.MASKL EQU      PS.TEMP         . LENGTH OF PARAMETER BIT MASK AREA
         LIST     L
*
*BOTTOM
F
READ HERE
```

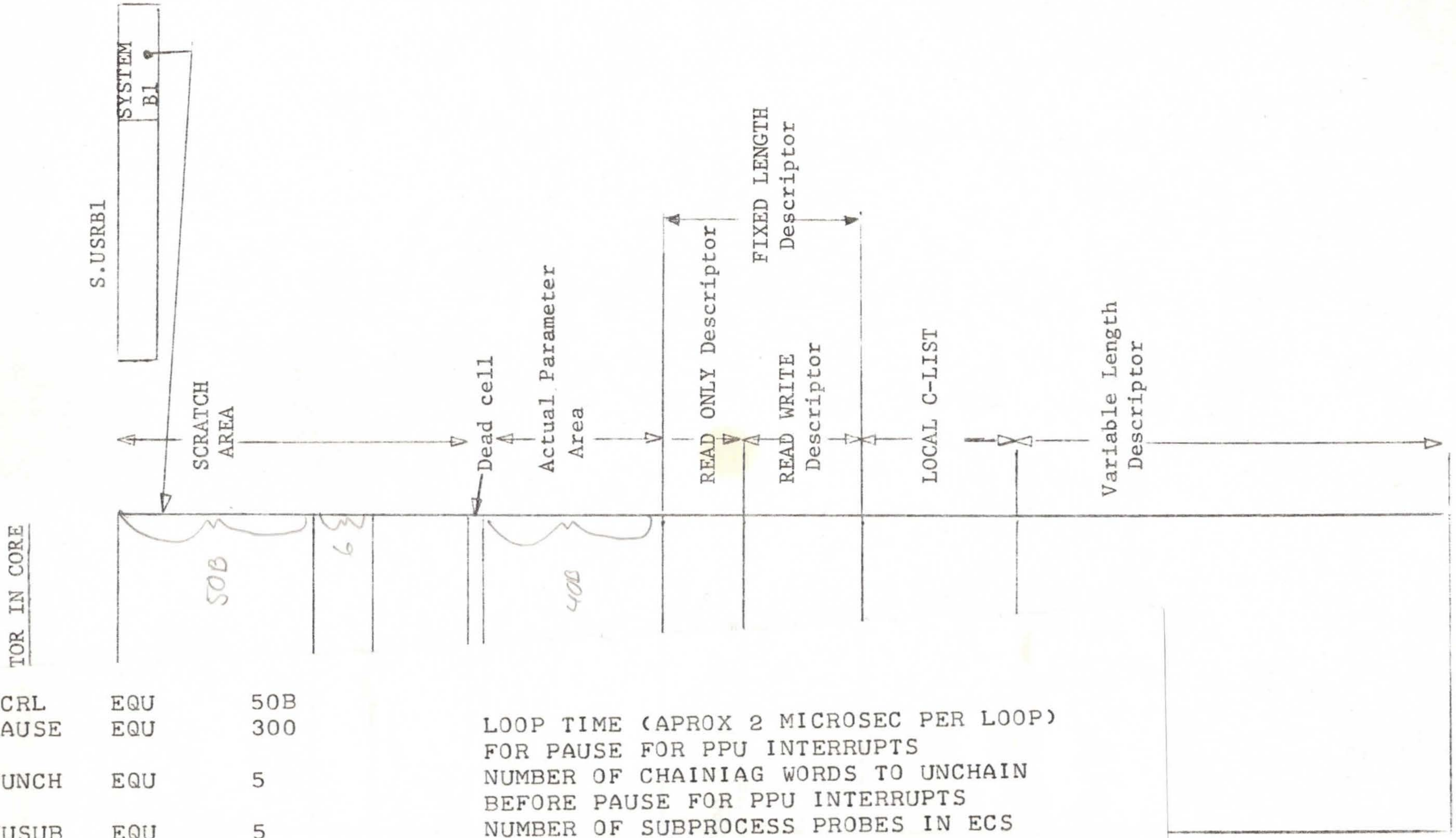
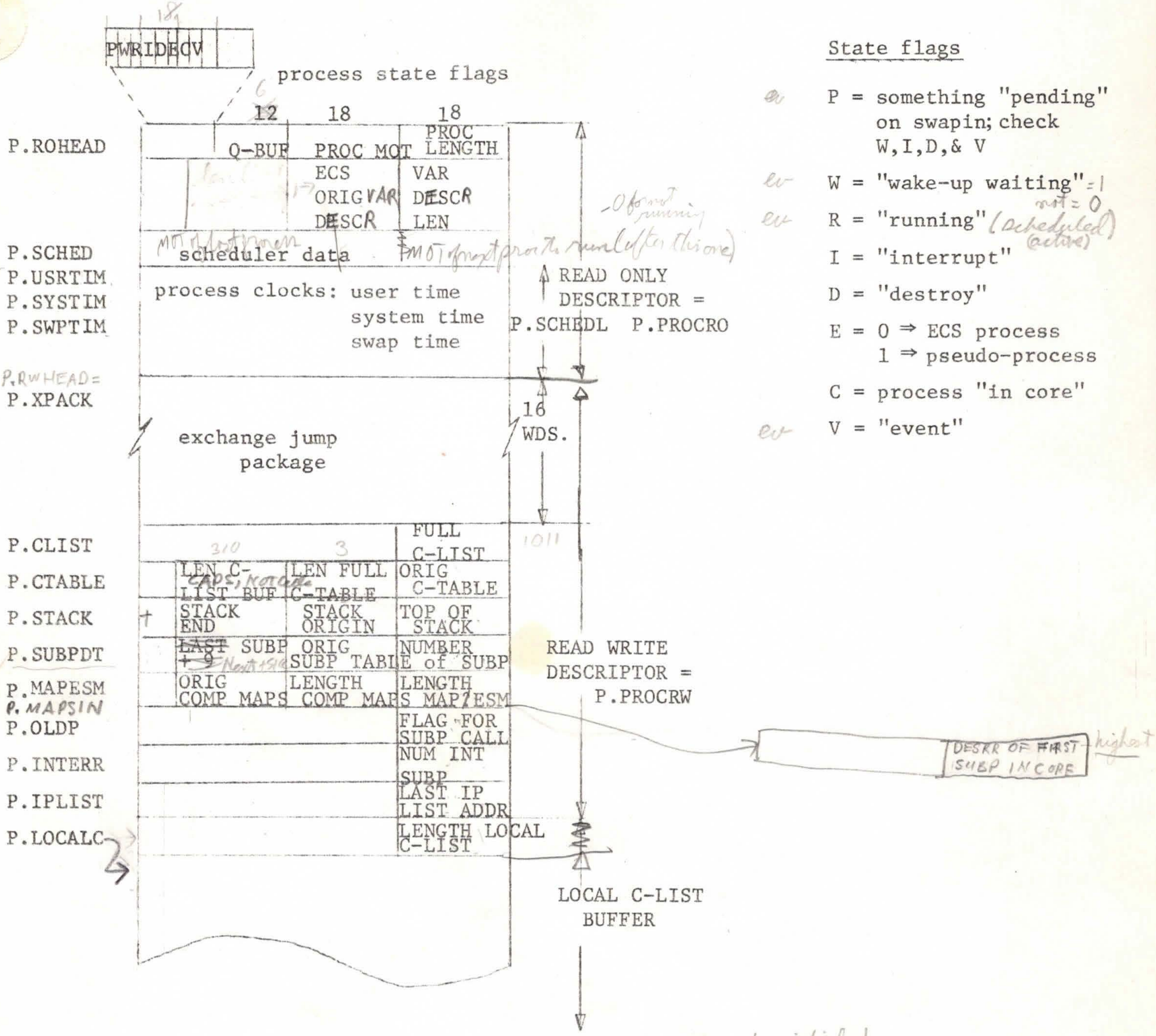


Figure 2

```
*
P.SCRL      EQU        50B
I.PAUSE     EQU        300              LOOP TIME (APROX 2 MICROSEC PER LOOP)
*                                       FOR PAUSE FOR PPU INTERRUPTS
I.NUNCH     EQU        5                NUMBER OF CHAINIAG WORDS TO UNCHAIN
*                                       BEFORE PAUSE FOR PPU INTERRUPTS
I.NUSUB     EQU        5                NUMBER OF SUBPROCESS PROBES IN ECS
*                                       DURING INTER-PROCESS INTERRUPTS
*                                       BEFORE PAUSE FOR PPU INTERRUPTS
*
```

FIXED LENGTH DESCRIPTOR



State flags

P = something "pending" on swapin; check W,I,D,& V

ev W = "wake-up waiting" = 1

ev R = "running" (scheduled) (active) not = 0

I = "interrupt"

D = "destroy"

E = 0 ⇒ ECS process
 1 ⇒ pseudo-process

C = process "in core"

ev V = "event"

<Q-BUF> ::= size of process queueing word buffer = max number of queuing words + 1 = numb specified + 1

<PROC MOT> ::= MOT index of process

<PROC LENGTH> ::= length of process in core [includes process descriptor (Fig.2)] + maximum full address space]

<VAR DESC LEN> ::= length of variable length descriptor

<ECS ORIG VAR DESC> ::= origin relative P.ROHEAD in ECS of variable length descriptor = Q-BUF + $\underbrace{\text{P.PROCRO + P.PROCRW}}_{\text{"constant"}}$

Figure 3

W: 1) tells the swapper to unchain the proc from event channels
2) prevents the proc from receiving any (more) events

VARIABLE LENGTH DESCRIPTOR

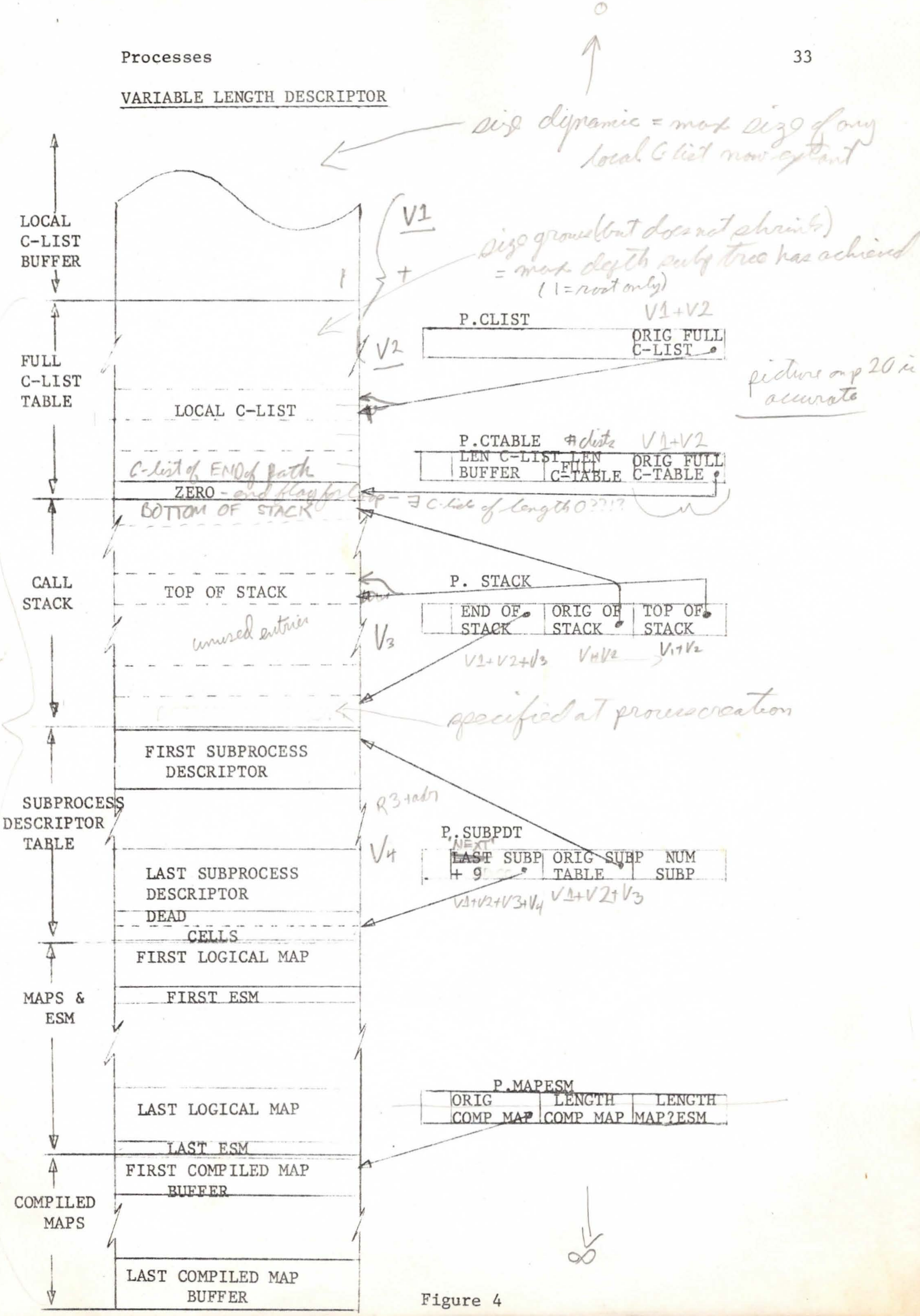


Figure 4

Process State Flags

Eight flags describe the state of the process. These state flags, stored in P.ROHEAD (see Figure 3), are used primarily to control the swapper, but are set and checked by other routines (event channel, process interrupt, and destroy process). Since the state flags are used to indicate the "state" of the process, they must never be modified without the PPU interrupts first being locked out to prevent 'test and set' overlaps.

The eight flags function as follows:

The E flag indicates that the process is actually a pseudo-process and is used by the event channel routines to distinguish between genuine and pseudo-processes.

a

The "in core" flag, C, is set whenever the process is actually running on the CPU. This flag is checked by the process interrupt routine.

The "pending action" flag, P, directs the swapper to interrogate the "W", "I", "D" and "V" flags. These four flags cause the swapper to:

W - (the wakeup waiting flag) unchain the process flow from the event channels;

I - check the "ancestors" of the current subprocess for an interrupt subprocess;

D - destroy the process; and

V - modify the swapper return because of the arrival of an event for the process.

The "running flag", R, indicates that the process is scheduled to run or is running on the CPU. The running flag (R) and the wake-up waiting flag (W) interact in the event channel routines as well as in the process interrupt routines. They are used to permit the process to "hang" on several event channels and still be able to accept an incoming event.

SUBPROCESS TREE AND FULL PATH

The subprocess tree is organized so that each subprocess references only its predecessor (see Figure 5). For each subprocess, the term "ancestors" refers to the sequence of subprocesses which starts with the subprocess and terminates with the root of the subprocess tree. Note that a subprocess is always an "ancestor" of itself. At any given time, there are two distinguished subprocesses within the process. They are known as the current subprocess and the end-of-path subprocess. The current subprocess is always an "ancestor" of the end-of-path subprocess; the sequence of subprocesses from the end-of-path to the current subprocess (inclusive) is called the full path. The end-of-path is defined dynamically by the flow of control among the subprocesses. The current subprocess may be considered to be the subprocess currently in control. The end-of-path and current subprocesses are reassigned whenever a new subprocess is called. The subprocess being called (the callee) becomes the new current subprocess. If the callee is an "ancestor" of the old end-of-path, then the end-of-path remains unchanged. If the callee is not an "ancestor" of the end-of-path, the new end-of-path becomes the same as the callee (i.e., the full path consists of a single subprocess - the callee). See Figure 5a.

The full path defines the sphere of protection invoked by the current subprocess. The access into the current subprocess permitted to other objects within the system is controlled by the full C-list. The full map determines the configuration of the address space available to the current subprocess, and the full address space is the size of the address space available to the current subprocess. The full C-list, full map, and full address space are defined by the full path. The configuration of the subprocess tree defines the static relationship between the subprocesses (subprocesses closer to the root may be given the privileges of their descendents) while the full path dynamically controls the boundaries of access applied to the current subprocess. This system of controlling the bounds of protection allows the construction of processes which may exercise varying degrees of protection while maintaining synchronization between the subprocesses involved.

SUBPROCESS TREE

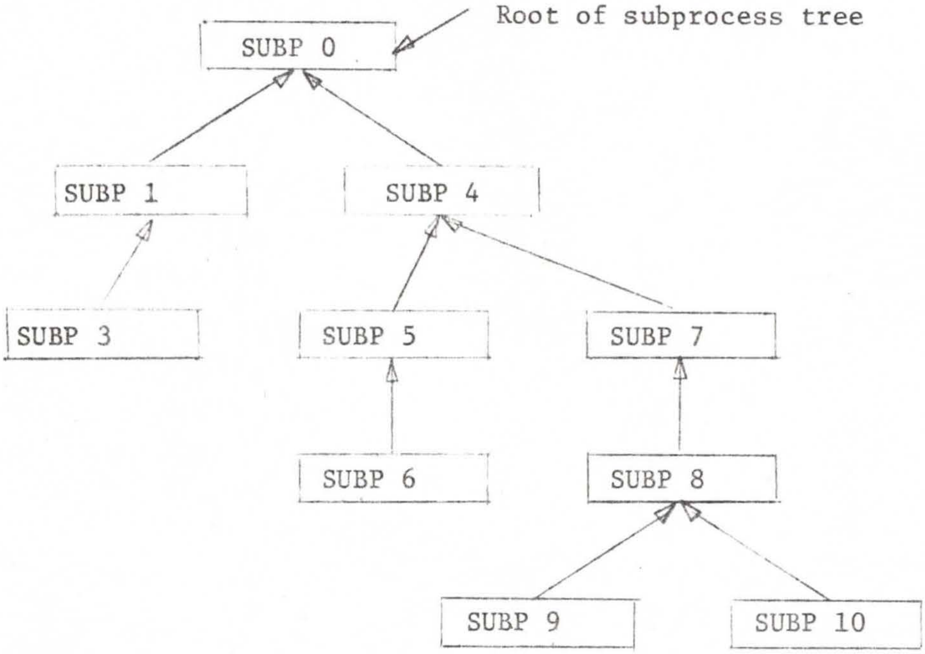


Figure 5

FULL PATH EXAMPLE

CALLING SEQUENCE	CURRENT SUBP	END-OF-PATH SUBP	FULL PATH
SUBP0	SUBP0	SUBP0	SUBP0
SUBP0 calls SUBP9	SUBP9	SUBP9	SUBP9
SUBP9 calls SUBP6	SUBP6	SUBP6	SUBP6
SUBP6 calls SUBP4	SUBP4	SUBP6	SUBP6,5,4
SUBP4 calls SUBP0	SUBP0	SUBP6	SUBP6,5,4,0
SUBP0 calls SUBP5	SUBP5	SUBP6	SUBP6,5
SUBP5 calls SUBP3	SUBP3	SUBP3	SUBP3

Figure 5a

CALL STACK

The call stack records the flow of control among the subprocesses. It contains the information necessary to reactivate a subprocess when control returns to the subprocess after one or more subprocess calls. Each stack entry is two words long (see Figure 6). The current subprocess, the end-of-path subprocess, and the P-counter must be saved at the time of the subprocess call to reconstruct the full path and to re-initiate processing where it was terminated by the subprocess call. The address (within the full address space of the subprocess) of the input parameter list (see System Entry/Exit) used for the last system call initiated by the subprocess, and the count of orders processed in the operation used in the last system call are retained to enable processing of F-returns. Finally, three flags are used to control the return of control to a subprocess. The "interrupted" flag indicates that the subprocess was interrupted and that the P-counter is not to be modified in the usual way (see System Entry/Exit). The "forced F-return" flag indicates that F-return action had been interrupted and instead of returning to the current subprocess, F return action should be initiated. Finally, an "inhibit interrupt" flag is used by the interrupt machinery to inhibit the interruption of the current subprocess by itself. P.STACK is used to control the call stack and contains the stack origin, stack end, and top of stack pointers relative to the incore process descriptor. The P-counter and input parameter list address in the top of the stack are not always maintained since the P-counter is in the process exchange package (P.XPACK) and the last IP list address is maintained in P.IPLIST. Each subprocess is assigned a maximum stack pointer value to prevent the stack from being filled to such an extent that the subprocesses closest to the root of the subprocess tree cannot be called to rectify the situation or to handle errors.

0

∞

CALL STACK

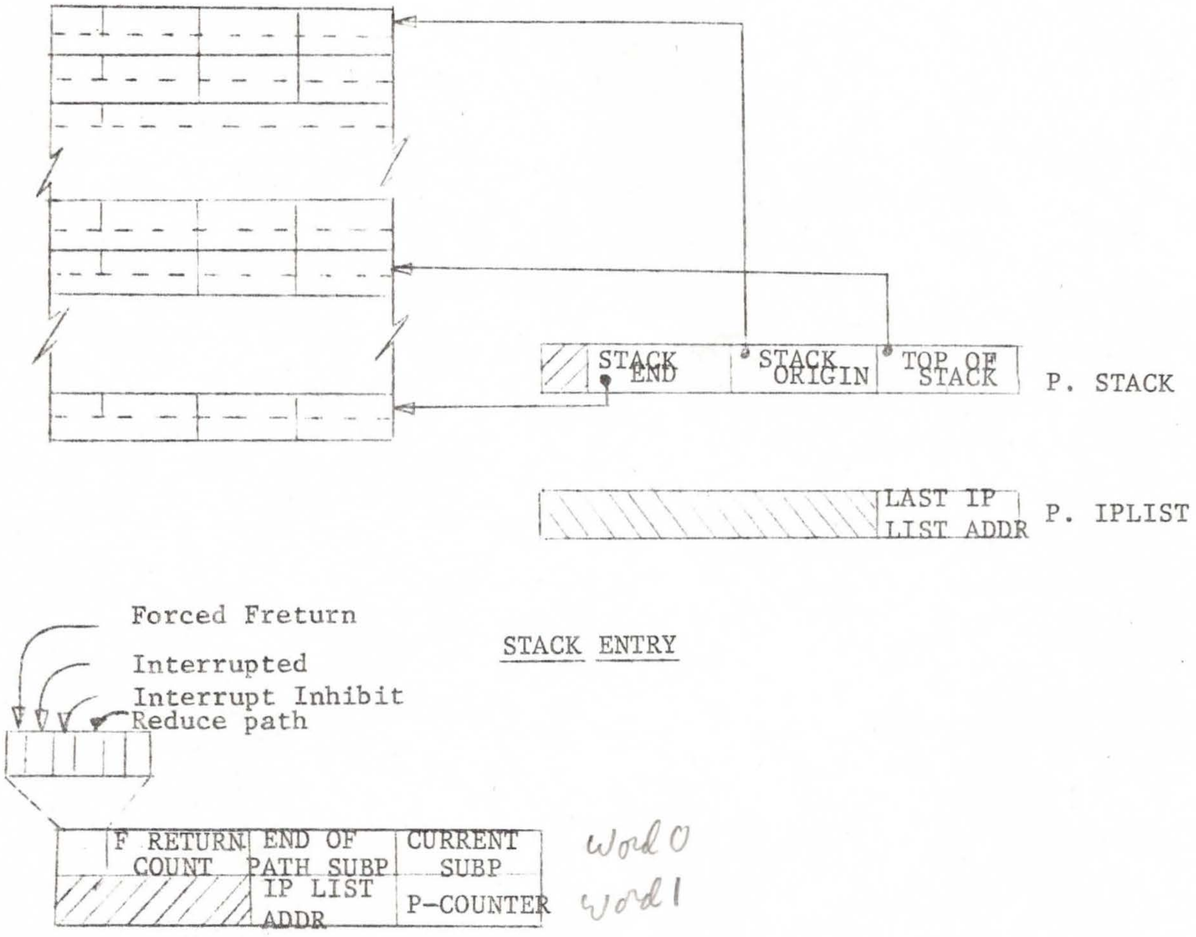


Figure 6

ERROR PROCESSING

The use of improper parameters in making an ECS system call is detected by the ECS system and is considered to be an error on the part of the process making the system call. The process must be informed of the existence and type of the error and in addition is given some control over which subprocess is to handle the error condition.

Associated with each error detected by the ECS system is an error class and an error number. Furthermore, associated with each subprocess is an error selection mask (ESM) (see Figure 7) indicating which classes of errors the subprocess is prepared to handle.

When an error is detected, it is first assigned an error class and error number. Then the "ancestors" of the current subprocess are checked (starting with the current subprocess) to find a subprocess whose ESM indicates it is willing to handle this class of errors. Finally, the subprocess which accepts the error is called, and is passed the error class and number as parameters of the call. In addition, in the ESM of the subprocess which accepts the error, the bit corresponding to the error class of the error is turned off to avoid error loops (i.e., a subprocess makes an error, accepts the handling of the error, and makes the same error).

ERROR PROCESSING AND PROCESS INTERRUPT

SUBPROCESS DESCRIPTOR (error processing data)
(process interrupt data)

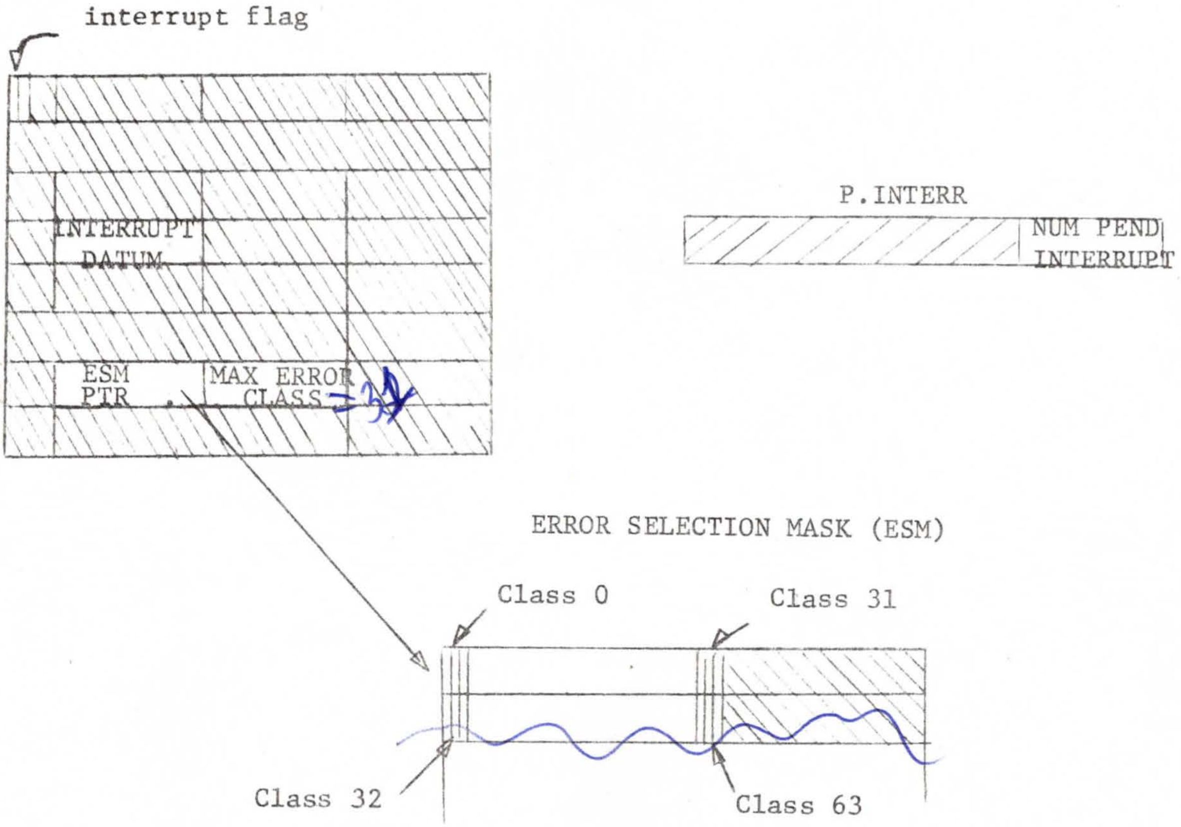


Figure 7

PROCESS INTERRUPT

Two mechanisms are available by which one process may affect the execution of another process: the event channel, used to synchronize otherwise asynchronous processes; and the process interrupt, used by one process to force the calling of a specified subprocess (called the *interrupt subprocess*) within another process (called the *interrupted process*). ~~Thus one process can force a second process to enter a specified subprocess. Furthermore,~~ The interrupted process will not enter the interrupt subprocess until the interrupt subprocess is an "ancestor" of the current subprocess. In this way, the interrupt is given a "priority" based upon the position of the interrupt subprocess in the subprocess tree of the interrupt process. With the process interrupt, an 18-bit *interrupt datum* is passed as the parameter of the call of the interrupt subprocess. Once a subprocess becomes an interrupt subprocess, and until that subprocess has been called as an interrupt subprocess, interrupts to that subprocess are disabled (i.e., additional interrupts specifying that subprocess have no effect). It is also possible to disarm interrupts which are the same as the current subprocess (recall that the current subprocess is an "ancestor" of itself and thus could interrupt itself). When an interrupt subprocess is called, interrupts are automatically disarmed for the current (= interrupt) subprocess.

If the interrupted process is "hung" when a process interrupt is initiated, the "ancestors" of the current subprocess (in the interrupted process) are scanned to see if the interrupt subprocess is among them. If the interrupt subprocess has "priority" over the current subprocess, the "wake-up waiting", "running", and "interrupt" flags are set in the interrupted process and the process is scheduled to run. how about P? Otherwise, the interrupt pending flag is set & the interrupt datum is stored on the interrupt SP & the interrupt waits for the flow of control to allow the interrupt SP to have priority. At every normal subprocess call and return, the number of pending interrupt subprocesses (P.INTERR) is checked. If there are interrupt subprocesses waiting, the "ancestors" of the new current subprocess are scanned to see if any of them are interrupt subprocesses. To facilitate this scan, the first bit of the subprocess descriptor (see Figure 7) is the "interrupt pending" flag. The interrupt datum is also stored in the subprocess descriptor. The "interrupt inhibit" flag (interrupt disarmed) in the

* subsequent interrupts are lost & the sender is so informed

stack is always checked if the interrupt subprocess is the same as the current subprocess. An interrupt subprocess call may also be initiated either when the "interrupt inhibit" flag is reset, or by the swapper, where a scan of the "ancestors" of the current subprocess is performed whenever the "interrupt" flag is set in P.ROHEAD (see Figure 3).

TRANSFER OF CONTROL

In general, there are six ways in which control can be transferred from one subprocess to another in the subprocess tree of a process. These may be grouped into two categories:

1. Subprocess call or jump: a new entry is made on the call stack, the full path is recomputed, parameters of the call action are passed, and execution is initiated at the proper entry point of the called subprocess. There are three kinds of subprocess calls: normal, interrupt and error. (See Subprocesses: Subprocess Calls).
2. Subprocess return: using an existing stack entry to obtain the new P-counter and the full path, the processing environment is reconstructed and control is returned to the subprocess. There are three kinds of subprocess returns: normal, F-return and forced F-return (see Subprocesses: Subprocess return).

SUBPROCESS

Every process is constructed as a set of related subprocesses in order to permit dynamic control of the privileges and protection applied to the process. The envelope of protection/privilege associated with a process may change as the process executes, but all changes in protection can be seen as being synchronous with the process execution. It is only through a subprocess transfer that the envelope of protection/privilege is modified.

SUBPROCESS DESCRIPTOR

The data necessary to describe each subprocess is gathered into an eight word subprocess descriptor (see Figure 1). The subprocess descriptors are stored together in the subprocess descriptor table in the variable length process descriptor (see Processes). Each subprocess has a name by which it can be identified and accessed. This subprocess name is a class code, the value of which is stored in the subprocess descriptor (word 1). In addition to its own name, each subprocess must maintain a link to its "father" in the subprocess tree (see Processes). This link is maintained in the descriptor (word 0) as a pointer to the parent subprocess. Process interrupt (words 0,4) and error handling information (word 6) are also maintained in the subprocess descriptor.

Associated with each subprocess is a local envelope of protection/privilege. The local C-list controls access to other objects within the system, while the subprocess map dictates the contents of the local address space. Information concerning the limits of the local address space (word 0), identification of the local C-list (words 4,5) and the subprocess map (words 3,4) are maintained in the subprocess descriptor.

The subprocess entry point (word 2) is the address, relative to the local address space, at which a normal subprocess call will initiate execution of the subprocess. The maximum allowable stack pointer (word 6) is used to avoid the filling of the process stack to such an extent that more privileged subprocesses (i.e., subprocesses nearer the root of the subprocess tree) cannot be called to rectify the situation or to handle errors. The sum of the lengths of the local C-lists and subprocess maps of all the subprocesses on the path to the root of the subprocess tree is maintained (word 2) to help compute the relative origins within the full map and full C-list of the calling subprocess during subprocess transfer operations. Finally, the last word of the subprocess descriptor is used to maintain a list of the maps which have been swapped into CM while the process is running on the CPU.

SUBPROCESS DESCRIPTOR

Figure 1

INTERRUPT FLAG
MAPIN FLAG 4250

SD·RAFL	WORD 0		RA + FL?	RA ?	PTR TO FATHER	✓ 462
SD·CC	WORD 1	SUBPROCESS NAME				?
SD·ORIG	WORD 2		ENTRY POINT hot	MAP ORIGIN 2	C-LIST ORIGIN 62	514
SD·MAP	WORD 3		COMP BUF SIZE 12	LOGICAL MAP PTR	COMPILED MAP PTR	536
SD·INT	WORD 4		INTERRUPT DATUM	# LOGICAL MAP ENTRIES 2	C-LIST LENGTH 43	
SD·CLST	WORD 5	C-LIST UNIQUE NAME		▨	C-LIST MOT	✓
SD·ERR	W RD 6		ESM POINTER 523	MAX ERROR CLASS #6	MAX STACK POINTER	54
SD·MLL	WORD 7	-0-			MAPIN LIST LINK	462

WORD 0 Interrupt flag: interrupt pending for this subprocess

mapin flag: set if map of subprocess has been swapped in

RA origin of local address space (relative to process CM origin)

1 beyond last useable cell → RA + FL end of local address space

Ptr to father: link to father in subprocess tree (relative to process CM origin)

WORD 1 subprocess name: the class code used to identify the subprocess

WORD 2 entry point: address relative to RA to begin execution on a normal subprocess call

Map origin: sum of "# logical map entries" of all "ancestors" except self

C-list origin: sum of "C-list length" of all "ancestors" except self

WORD 3 comp buf size: number of words allocated for the compiled map buffer

logical map ptr: pointer (relative to process CM origin) to logical map of subprocess

compiled map ptr: pointer (relative to process CM origin) to compiled map buffer

WORD 4 interrupt datum: interrupt parameter if interrupt flag set

# logical map entries: number of swapping directives permitted in logical map

C-list length: number of capabilities or "empties" in local C-list

WORD 5 C-list unique name and MOT index: identification of local C-list

WORD 6 ESM pointer: pointer (relative to process CM origin) of first error selection mask word

max error class: maximum error class which is possible to recognize in ESM

max stack pointer: maximum permissable stack pointer for the subprocesses to be called

WORD 7 mapin list link: if mapin flag is set then link to subprocess whose map is swapped in below this subprocess in CM. If this subprocess is at the end of the map chain; then zero.

SUBPROCESS TRANSFER

The envelope of protection/privilege applied to a process is modified by switching control from one subprocess to another. Subprocess transfers fall into two categories: subprocess calls and subprocess returns. A subprocess call causes a new entry to be made on the call stack*, the full path to be recomputed, parameters of the call to be passed, and execution to be initiated at the proper entry point of the subprocess. A subprocess return ~~passes no parameters and~~ draws the full path and P-counter from an existing stack entry. In each case, the processing environment must be reconstructed to reflect the new full C-list, full map, and full address space. This reconstruction requires the swapping of one or more subprocess maps, the re-building of the full C-list table (see Capabilities and C-lists), the fetching of a new local C-list and setting of the full address space limits.

SUBPROCESS CALLS

There are three kinds of subprocess calls. The normal subprocess call or jump call is initiated by calling on the system in the usual manner, using an operation (IPO) whose action is "subprocess call". A normal subprocess call may also be initiated as the result of F-return action under the control of a multi-ordered operation (see System Entry/Exit - Operation Interaction).

The error subprocess call is initiated by the ECS system or by a user request and will call the closest "ancestor" of the current subprocess which has the proper error class selected in its error selection mask (ESM) (see Processes, Error Processing). Finally, an interrupt subprocess call is initiated whenever a subprocess which is an interrupt subprocess has priority over the current subprocess (see Processes, Process Interrupt).

For all subprocess calls*, a new stack entry is constructed and the new processing environment is established. The P-counter and last IP list address of the current subprocess are stored in the old top of the stack**. Then cells 0 and 1 of the full address space are zeroed. These cells are used in the event of hardware arith errors and to simulate SCOPE system calls. Next, the origins (relative to the new local environment) of the address space, C-list, and map of the calling subprocess are computed and stored in cells 3, 4, and 5 of the full address space. If the calling subprocess is not a member of the new full-path

* jump call doesn't make a new entry ** skipped for jump call

+1 cell 2 is ~~ca~~used at present for interrupt datum (see Processes), then these cells are zeroed (see Figure 2). Following the relative origins of the caller's address space, C-list, and map, the parameters of the subprocess call are copied to succeeding words of the subprocess address space.

For a normal call, the parameters of the call are first formatted in the actual parameter area (P.PARAM) of the process descriptor by the system entry mechanism. These parameters are drawn from the user's input parameter list (IP list) under the direction of the operation being used for the subprocess call (IPO). In addition, the system entry routine places the name (class code) of the called subprocess at P.PARAMC, the number of parameters at P.PARAMC - 1, and a bit string denoting the types of the parameters at P.PARAMC - 2. After establishing the correct processing environment for the called subprocess, the parameters are transfered, under the control of the parameter type bit mask, to the local address space and local C-list of the called subprocess. Datum parameters are simply copied to the next parameter cell in the local address space. Capability parameters are copied to successive positions in the local C-list and the index of the parameter in the local C-list is stored in the next parameter cell in the local address space.* On completion of the parameter passing, execution is initiated at the entry point of the called subprocess.

During all subprocess transfer operations, if the interrupt pending count (P.INTERR) is non-zero, the "ancestors" of the current subprocess are checked to see if any of them are "interrupt" subprocesses (word 0 of subprocess descriptor). If so, the subprocess transfer operation is terminated and an interrupt subprocess call is initiated. As part of the termination of the previous subprocess transfer operation, the "interrupted" flag is set in the stack entry corresponding to the subprocess that was to be executed (if F-return action was interrupted, the "forced F-return" flag is set in the stack instead of the "interrupted" flag). As with the other subprocess calls, the processing environment, a new stack entry, and the origins of the previous subprocess are constructed for the interrupt subprocess call. The interrupt datum from the subprocess descriptor (word 4) is stored in cell ~~3~~ 2 of the new local address space, and the "interrupt inhibit" flag is set in the new stack entry.

Finally, the interrupt subprocess is entered 2 words before the entry point specified in the subprocess descriptor.

* ommission of BD, BC Type params

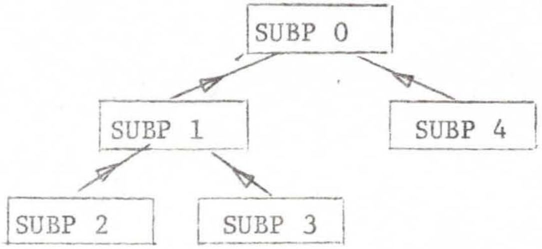
An error subprocess call is initated by the ECS system or by user request. An error subprocess call passes as its parameters the error class and error number which describe the error causing the call. Also, the bit in the ESM of the error subprocess corresponding to the error class ~~must be~~ is reset to avoid error loops (e.g. subprocess makes error - gets called as error subprocess - makes the same error - gets called as error subprocess - etc.). The entry to an error subprocess is one word before the normal entry point.

SUBPROCESS RETURN

Like the subprocess call, the subprocess return must construct a new processing environment before returning control to the user. The return routines re-activate a subprocess using information left in a stack entry. The full path recorded in the stack entry is sufficient to reconstruct the processing environment. The P-counter from the stack entry, along with the "interrupt" flag, control where in the subprocess execution is initiated. The normal return requires the P-counter to be modified by the low order 18 bits of the CEJ instruction which originally caused control to pass to another subprocess (see System Entry/exit). If the "interrupted" flag is set, the P-counter is not to be modified. Finally, the "forced F-return" flag in the stack will cause the subprocess return routine to transfer to the F-return routine (see System Entry/Exit - Operation Interaction). ?

STATIC STRUCTURE

	SUBP 0	SUBP 1	SUBP 2	SUBP 3	SUBP 4
Father	root	SUBP 0	SUBP 1	SUBP 1	SUBP 0
Subp origin (RA)	100B	300B	350B	350B	300B
local addr space (FL)	200B	50B	100B	250B	150B
C-list length	10B	20B	5B	15B	25B
C-list origin	0	10B	30B	30B	10B
map length	4	5	10B	6	3
map origin	0	4	11B	11B	4



Subprocess Tree

DYNAMIC STRUCTURE

				CALLER		
		ADDRESS	ADDRESS	ADDRESS	CALLER	CALLER
		SPACE	SPACE	SPACE	C-LIST	MAP
SUBPROCESS CALLS	FULL PATH	ORIGIN	LIMIT	ORIGIN	ORIGIN	ORIGIN
SUBP 0	subp 0	100B	300B	-0-	-0-	-0-
SUBP0 calls SUBP2	subp 2	350B	450B	-0-	-0-	-0-
SUBP2 calls SUBP1	subp 2,1	300B	450B	50B	20B	5
SUBP1 calls SUBP0	subp 2,1,0	100B	450B	200B	10B	4
SUBP0 calls SUBP3	subp 3	350B	620B	-0-	-0-	-0-
SUBP3 calls SUBP0	subp 3,1,0	100B	620B	250B	30B	11B
SUBP0 calls SUBP4	subp 4	300B	450B	-0-	-0-	-0-
SUBP4 calls SUBP0	subp 4,1	100B	450B	200B	10B	4

Subprocess Calling Example

Figure 2

CLASS CODES

A Class code is a protected 60-bit datum by which a user may identify himself or some ECS system object. Within the ECS cyctem, class codes are used as the names of subprocesses (See SUBPROCESSES); in the future they will be used to identify users within the disk system and will be called access keys.

The 60-bits of a class code are divided into two 30-bit parts (see Figure 1). The upper 30-bits constitute the "permanent part" and are assigned by the system when the class code is created. Once assigned, the permanent part cannot be altered. The low order 30-bits of a class code, called the "temporary part", are set by the user and may be altered by him any time.

Since each class code occupies only one word, they are not allocated space of their own in ECS, but instead each is kept in the second word of the capability which refers to the class code. Since the second word of the capability usually contains the unique name and MOT index for the object, this choice of location for the class code seems reasonable.

There are two system actions connected with class codes: The first allows the user to obtain from the system a new class code. The system keeps a counter for generating the "permanent part" of a class code, and each time one is requested, the counter is incremented and a new and unique class code is generated. The second action allows the user to set the temporary part of a class code. He must already have a permanent part, the capability for which (with the proper option bit set) he supplies as the first parameter. The second parameter is the 30-bit datum which is to be inserted into the temporary part of the class code. The 3rd parameter is a C-list index to return the updated class code. A class code is destroyed only when the capability is destroyed by being written over.

Figure 1. Class Code

59 18 0

OPTIONS	TYPE
Permanent part	Temporary part

50

Maps

Associated with each subprocess is a map which directs the swapping of the subprocess address space between central memory and ECS files. A map consists of a fixed length sequence of map entries each of which is either "empty" or contains a swapping directive. A swapping directive (see Figure 1) designates a contiguous portion of an ECS file, a CM address within the local address space of the subprocess, and whether or not that section of subprocess memory is read only (not to be swapped out).

When a subprocess is to be swapped into CM, each non-empty map entry is processed in sequence and a file read action is effectively performed to copy the section of the file designated by the swapping directive to the local address space of the subprocess starting at the designated CM address. When a subprocess is to be swapped out, only those swapping directives not marked as "read only" need be processed. Note that there is nothing to prevent several swapping directives from designating overlapping areas in CM or in a file. The results of overlapping swapping directives may be determined by remembering that swapin/swapout processes the map entries in sequential order.

To minimize the time spent in swapping maps in and out, the logical map (sequence of "empties" and swapping directives) is "compiled", or converted, to a form containing the absolute ECS address of the sections of ECS files referenced by the swapping directives (see Figure 2). Since one swapping directive may span several data blocks in a file, the size of the compiled form of the map will reflect the need for additional entries in the compiled map. Both the number of entries in the logical map and the number of words to be allotted for the compiled map are declared when the subprocess is created and may not be altered.

The absolute ECS addresses in the compiled map are sensitive to changes in ECS due to garbage collection. Thus, the map must be re-compiled whenever a garbage collection is in progress or has occurred since the last re-compilation. A word in ECS (GARBCNT) indicates whether or not a garbage collection is in progress and contains the number+1 of garbage collections since system initialization. Each compiled map contains, as a prefix, the count of garbage collections at the time the map was last compiled. This count is compared with GARBCNT whenever the compiled map is about to be "executed" and will cause a recompilation if the counts are unequal. A recompilation of a map may be forced by setting the count in the compiled map prefix to zero.

Access to both the logical and the compiled forms of the map is through the subprocess descriptor (see Fig. 3). The subprocess descriptor also contains the number of entries in the logical maps and the size of the buffer allocated for the compiled map. In addition, the subprocess descriptor contains a flag indicating whether the map for that subprocess has been swapped into CM and a chain pointer used to keep track of which subprocess maps are in CM. The origin (relative to B1 , the CM process origin) of the subprocess address space (RA) and the origin + length (RA + FL) of the subprocess address space are also available to the map machinery in the subprocess descriptor.

The maps of the subprocesses in the full path are concatenated to form the full map in much the same way as the full C-list (see C-list) is formed. Each map however, is swapped relative to the address space of its subprocess, as if it were the only map being considered. The address space of the running subprocess is enlarged to form the full address space, which includes the address space(s) of all other subprocesses in the full path. The code and data in the maps above (in the full path) the running subprocesses may be accessed as if the address spaces of the other subprocesses were simply added (one after another) onto the end of the local address space of running subprocess. Note, however, that the data and code within these maps is not relocated to reflect the new addresses used to access them.

Map Actions

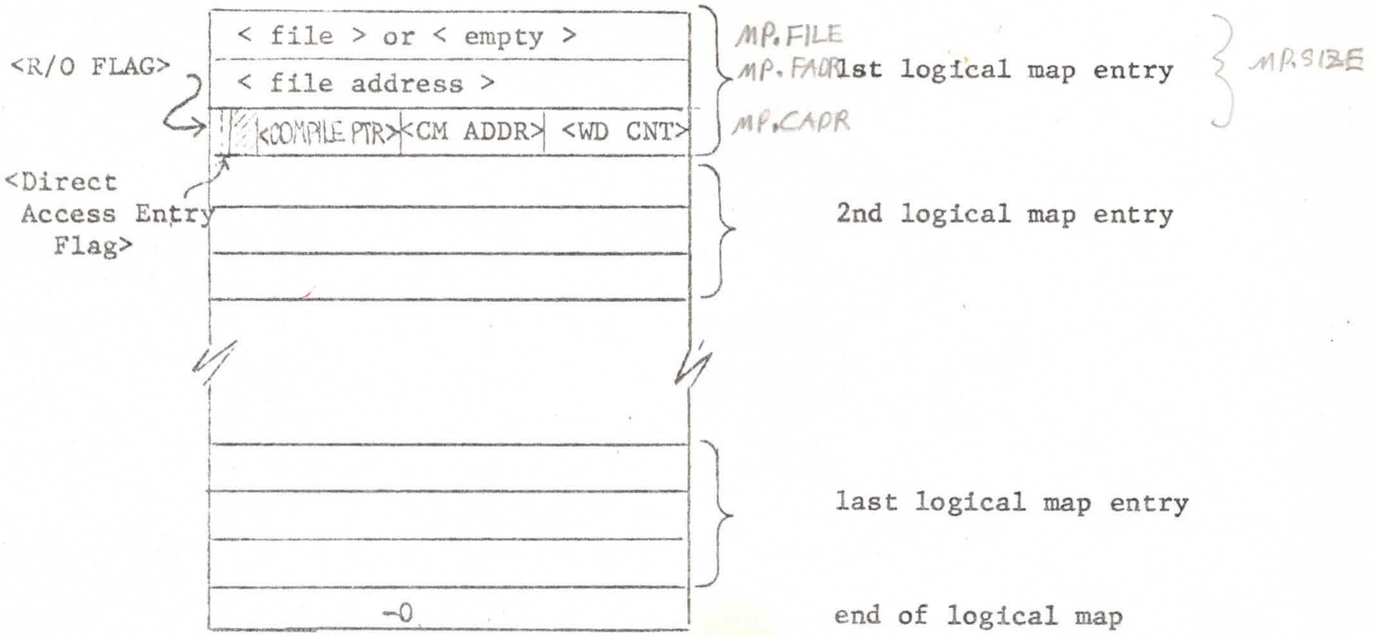
When map entries are to be changed, care must be taken when the map involved is part of the full map. In this case, if the map entry involved is not empty, it must be swapped out before it can be replaced. The new entry (if there is one) can then be constructed and swapped in. Note that overlapping map entries will behave oddly since the portions swapped under one map entry may be partially or completely overwritten by the area swapped under a subsequent map entry. At the present time, the entire map is recompiled, since a change in the logical map may change the length of the compiled map. Incremental compilation is not precluded by the design since the logical map contains pointers into the compiled map; however, the implementation of this feature has been deferred.

Direct User ECS Access

To allow the user an ECS RA and FL, so that he may access directly an often used segment of ECS, the current subprocess is permitted to have one direct ECS access map entry. If present, it must always be the first map entry, and may reference only one file block (due to physical limitations).

The Direct Access Entry (DAE) is implemented as a regular map entry (as far as the map compiler is concerned) except that the CM address part is always zero and the DAE flag bit (appearing only in the first map entry) is set. Two special ECS system actions are available to set/clear the DAE flag bit.

Figure 1 Logical Map



< empty > ::= +0 Denotes an "empty" map entry

< file > ::= | UNIQUE NAME (39) | ▨ | MOT INDEX (18) | file identification

< file address > ::= 0 → 2^{60} -1

< R/O FLAG > ::= 1 ⇒ read only; 0 ⇒ read/write

< compile ptr > ::= index in compiled map buffer of first compiled map entry for this swapping directive

< CM ADDR > ::= CM address within subprocess local address space

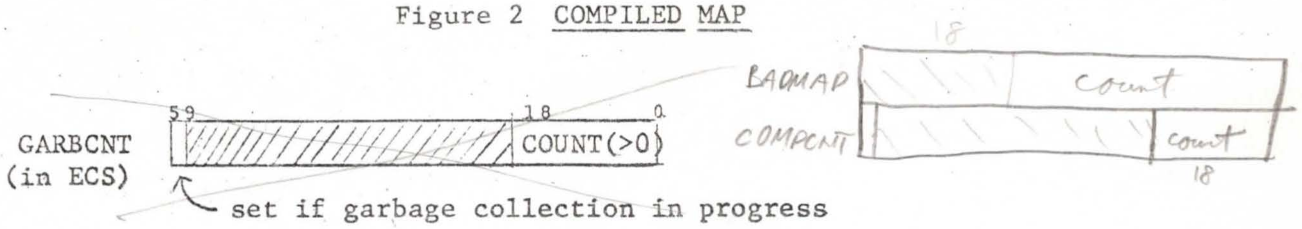
< WD CNT > ::= word count

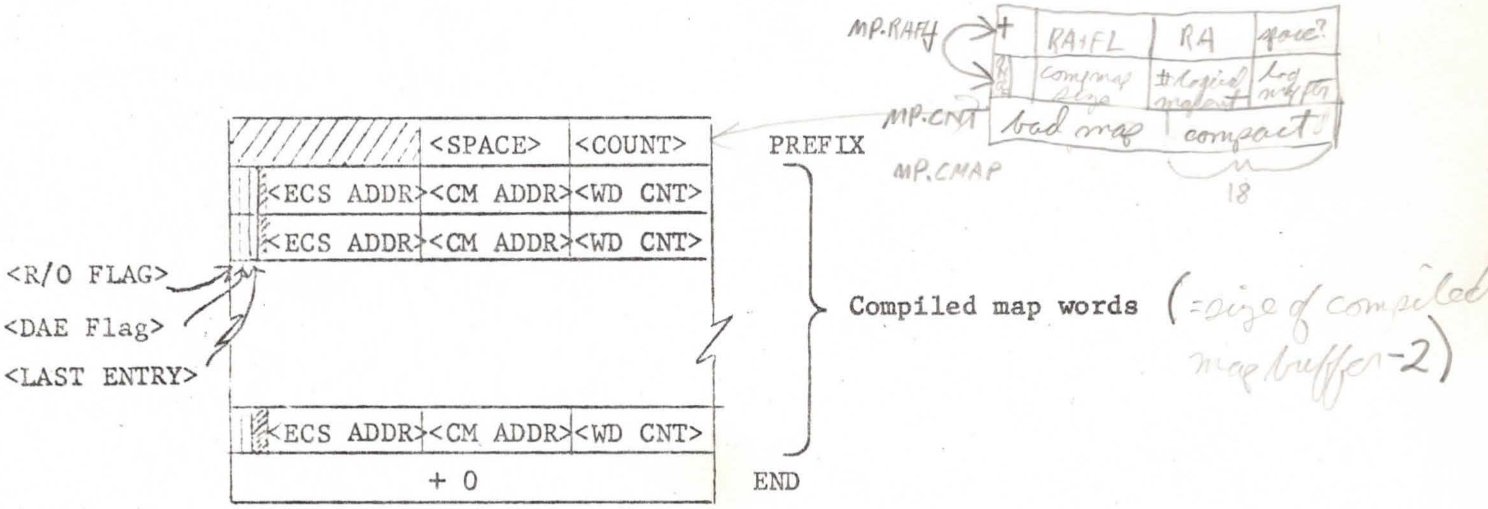
Note: < CM ADDR > + < WD CNT > ≤ length of subprocess local address space

< DAE Flag > ::= 1 -- this is a direct ECS access entry (Legal only for first entry)

0 -- regular map entry

Figure 2 COMPILED MAP





< COUNT > ::= { 0 ⇒ must recompile
>0 ⇒ map is good if same as GARBCNT }

< SPACE > ::= number of un-used words in the compiled map buffer

< WD CNT > ::= number of words to transfer

< CM ADDR > ::= CM address relative to CM process origin (B1)

< ECS ADDR > ::= absolute ECS address to start transfer

< R/O flag > ::= read only flag { 0 ⇒ read/write
1 ⇒ read only }

< DAE flag > ::= 1 -- DAE (legal only on 1st entry in compiled map)

< last entry > ::= 1 -- last compiled map word corresponding to a particular swapping directive

SUBPROCESS DESCRIPTOR (MAP DATA)

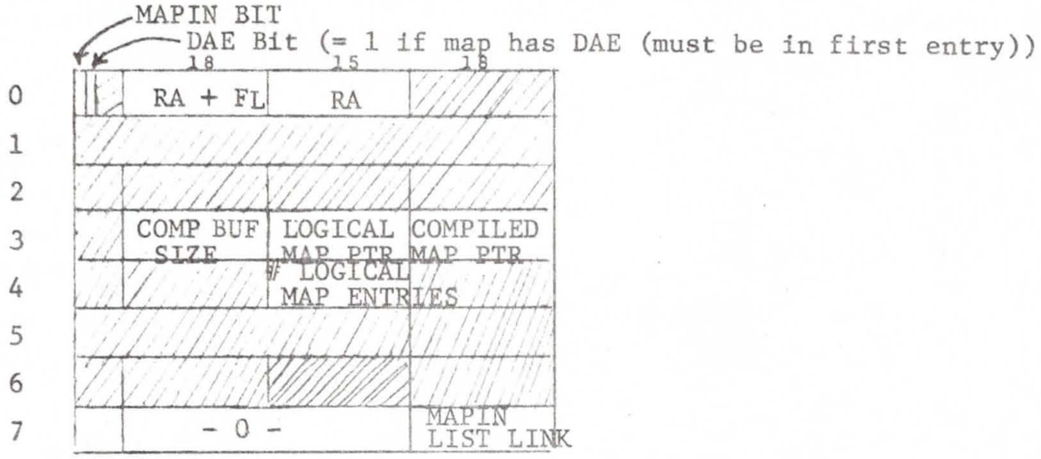


Figure 3

```
Map Compiler

        Error flag ← 0
        Save registers
        New count word for compiled map
        Initialize regs to 1st logmap, 1st compmap etc
        first entry null (or removed)?  —yes→  Done
            ↓ no
Cloop   read file MOT  ←──────────────┐
        gone?                          │
          ↓ yes                        │
        map entry ← null               │
        errorflag ← nonzero            │
          ↓                            │
          cloop 1                      │
    no
        → set new compmap ptr in logmap
find  ( read file descriptor
1st   ( read data block (block gone → disaster)
DB    ( yes  last comp entry for this log entry?
                ↓ no
            RO?
            yes set dirty bit on block
            → set 1 log map word
            get next DB  ←──── (else disaster)
            last entry?
            RO?
            yes set dirty bit
            → set 1 log map word
```

```
mkMAP2   RO?
         yes  set Dirty bit
              set last Compmap-entry
                                                     clop
CLOOP1   next log entry
    yes  good entry  ──────── yes ────────┘
Done ←   null entry  ←────────────────────────

         set 0 word at end of Compmap
         restore registers
         X6 ← error flag
         exit
```

EVENT CHANNELS

Event Channels are ECS system objects used to synchronize running processes as well as to implement "block" and "wake up" mechanisms. Basically, a user process may request an event from a particular event channel. If the event channel does not have an event, the user's process is blocked (stops running) until some other process sends an event to the event channel. The exact mechanisms of sending and receiving events will be described in full detail.

The event channel (see Figure 1) consists of a three word header followed by the event queue. The event queue is a circular buffer controlled by pointers and values located in the first and third header words.

First header word: The "in" and "out" pointers in the first word are manipulated to point relative to the beginning of the event channel. The "in" pointer always points to the location in which an event is to be put should one arrive. The "out" pointer points to the location of the next event to be removed from the event queue. The "in" pointer will equal the "out" pointer when the event queue is either empty or full. Therefore, the number of empty places in the circular buffer is maintained in the third header word. Finally, the length of the entire event channel is recorded in the first header word.

Second header word: The second header word is used to maintain a queue of waiting processes. When a process requests an event and the event queue is empty, the process is added to the process queue. The process queue is a bi-directional list through the processes on the queue and the event channel (see Figure 2). The high order 30 bits of the second word of the header, called the process queuing word, hold the forward pointer while the low order 30 bits hold the backward pointer. Each pointer consists of a Master Object Table (MOT) index and a queuing word index. The queuing word index, in the high order 12 bits of the pointer, is an index relative to the beginning (in ECS) of the process which is designated by the MOT index of the low order 18 bits of the pointer.

-1!

Within the process, at the location indicated by the queuing word index, there should be another process queuing word with forward and backward pointers. The queuing word index is stored in such a way that the unpack (UXi Bj,Xk) instruction will result in the true queuing word index in the B register. Furthermore, if the pointer refers to the event channel, the queuing word index will unpack to a -2 in the B register. For example, the pointer: $2061_8|000123_8$ refers to the 61_8-st word (in ECS) of the process with MOT index 123_8. Similarly the pointer: $1775_8|00321_8$ refers to the process queuing word of the event channel with MOT index 321_8. If the process queue is empty, the process queuing word in the event channel will point to the event channel itself (e.g., $(1775_8|000321_8|1775_8|000321_8)$).

Event Channel Routines

It is important to note before discussing the event channel routines that they are one of the few places in which there is interaction between the ECS action routines and the interrupt system. Since the interrupt system may call upon the event channel routines at any time, it is necessary to lock out the interrupt system while manipulating event channels and to release the lockout upon completion of any event channel manipulations. To lock out the interrupt system, it is only necessary to set I.LOCK (in system core) nonzero. To release the lock, simply clear I.LOCK.

Sending Events

Events are sent by user processes and by the interrupt system. An event consists of two words. The first word ~~is~~ contains the unique name ~~MOT index~~ of the process which is sending the event with the chaining word index tacked on. The second word is a 60 bit datum provided by the sender of the event. A response is always given to the sender of the event to indicate the disposition of the event (see Figure 3). For a user process, the response is returned in X6.

del

If the event queue of the appropriate event channel is not empty, then it may or may not be searched for an event duplicating the new event. This is to allow for the elimination of redundant events. If the event queue search was desired and if a duplicate event is found, a response is given to the sender indicating that a duplicate event was discovered, and the event sending routine returns.

del

If no duplicate event checking was requested or no duplicate event was found, the event queue is checked to see if it has more than one empty slot. If the event queue is full, the sender of the event is notified that the queue is full, and control returns to the sender of the event. If there is only one slot left in the event queue, the datum word is replaced by a special "you lose" datum (-0) and the sender is notified by the "you lose" response. This "you lose" datum allows the process which ultimately receives that "you lose" event to discover that the event queue had been full and that information was lost.

If the event survives ~~the duplicate event checking and~~ the full event queue condition~~s~~, it is copied into the event queue and the pointers are moved to reflect its presence. Again, the sender of the event is notified of the deposition of the event.

If the event queue is empty, the process queue must be checked. (Note that if the event queue is not empty, then the process queue must be empty.) The process queue is scanned for the first process which does not have its "wake-up waiting" flag set, i.e., has not already been handed an event, received a process interrupt, or been marked for destruction. If such a process is found, and it is not a pseudo process (used by interrupt system to interface with the event channel logic and other purposes), the "wake-up waiting" flag is set on that process. The P counter in the process exchange package is incremented and the event is copied to X6 and X7 of the process exchange package in ECS. Note that the testing and setting of the "wake-up waiting" flag must not be interrupted by any other access to this flag. If the process is not running ("running" flag) the scheduler is called to schedule the process to run. If the first process without "wake-up waiting" is a pseudo process, it is removed from the process queue; otherwise, it is not removed until the process is swapped in to run. Also, in the case of a pseudo process, the event channel routines return to UNHUNG1 in the interrupt system.

Finally, the "running", "event", and "pending action" flags are set in the process. The "pending action" flag, the "event" flag, and the "wake-up waiting" flag are used to control the swapper and the routines for hanging a process on several event channels, process interrupt, and process destruction.

If the process queue is empty or has no processes without "wake-up waiting", and the event queue is empty, the event is copied to the event queue and the appropriate response is passed to the sender.

Getting Events

A user process may attempt to get an event from an event channel. If the event queue is empty, the process may wait ("hang" or "block") until an event arrives before resuming execution. Also, a process may attempt to get an event from any one of a set of event channels and, in the absence of any events, the process may discontinue execution ("hang" or "block") until an event arrives for one of the event channels. If more than one process is awaiting an event on a single event channel, the first event to be set to that channel is passed to the first process while the other process(es) continue to wait.

The mechanism of getting an event or hanging (waiting for an event to arrive) begins with a check on the event queue of the event channel. If the event queue is non-empty, the head of the event queue is removed and the event is passed to the process (in X6 and X7 for a user process).

If the event queue is empty the process must be added to the queue of waiting processes (process queue) using a process queueing word in the ECS image of the process. The "running" flag in the process is cleared and the process is removed from the scheduling queue (de-scheduled). Next, the P-counter of the process is decremented by one. This is to allow for the possibility of a process interrupt causing the process to resume execution. In this case, when the interrupt subprocess returns, the process will re-execute the exchange jump, which calls the system to try to get an event from the event channel. When the process has been chained on the process queue, the system and user clocks are updated and the event channel routines exit to SWAPOUT in the swapper to swap out the process.

When an event arrives for a process which is hung on an event channel, the event sending mechanism will set the appropriate flags and schedule the process to run as described above. The swapper will detect the "event" flag and return through SYSRET instead of TOUSER of the system entry/exit routines. The swapper will have already removed the process from any process queues on which it had been hung.

To get an event from one of a set of event channels, the event channel routines must interrogate the event channels one at a time. If an event channel has an empty event queue, the process is queued in the process queue of that event channel using the next queuing word of the process. The sequence of "in use" queuing words in the process must be terminated by a zero word. Between the interrogation of event channels, the "wake-up waiting" flag is checked. If this flag is set, an event has arrived on one of the event channels which has already been interrogated. If an event has arrived or an event is discovered on an event queue of an event channel, the process is removed from all the process queues on which it is already chained, and the event channel routines exit to the system entry/entry mechanism. When interrogating the set of event channels periodic pauses must be made to allow the interrupt system to run. Otherwise, the interrupt system might be locked out for an intolerably long time. If, after interrogating the last event channel, the "wake-up waiting" flag is not set (note that the interrupt system is still locked out), the process is descheduled, the P-counter is decremented, and the event channel routines exit to SWAPOUT in the swapper.

Figure 1

EVENT CHANNEL

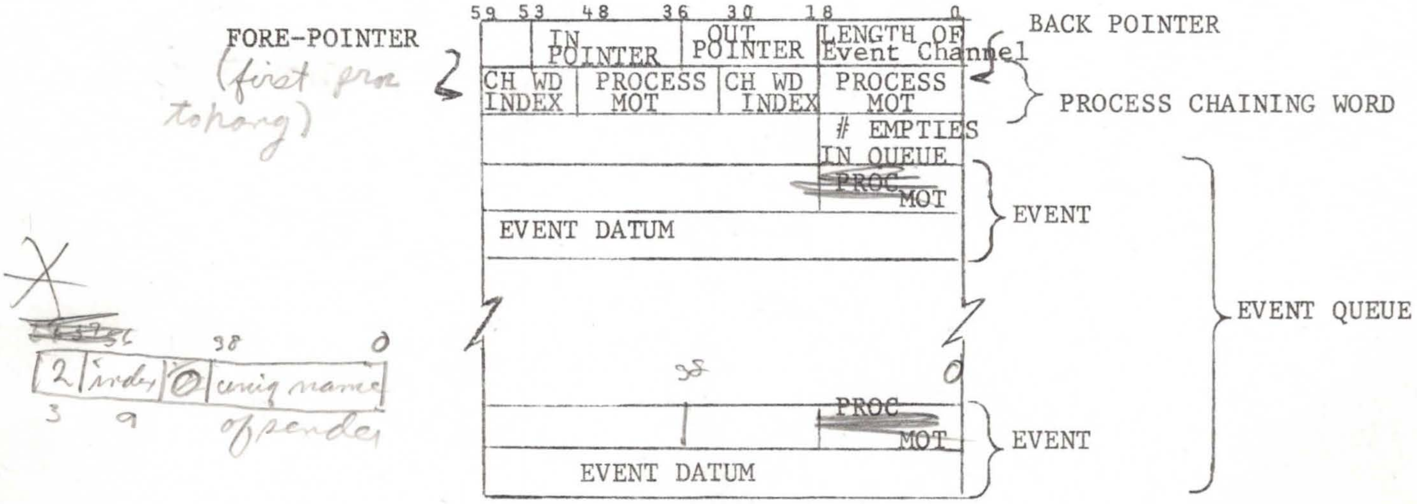


Figure 2

PROCESS QUEUE EXAMPLE

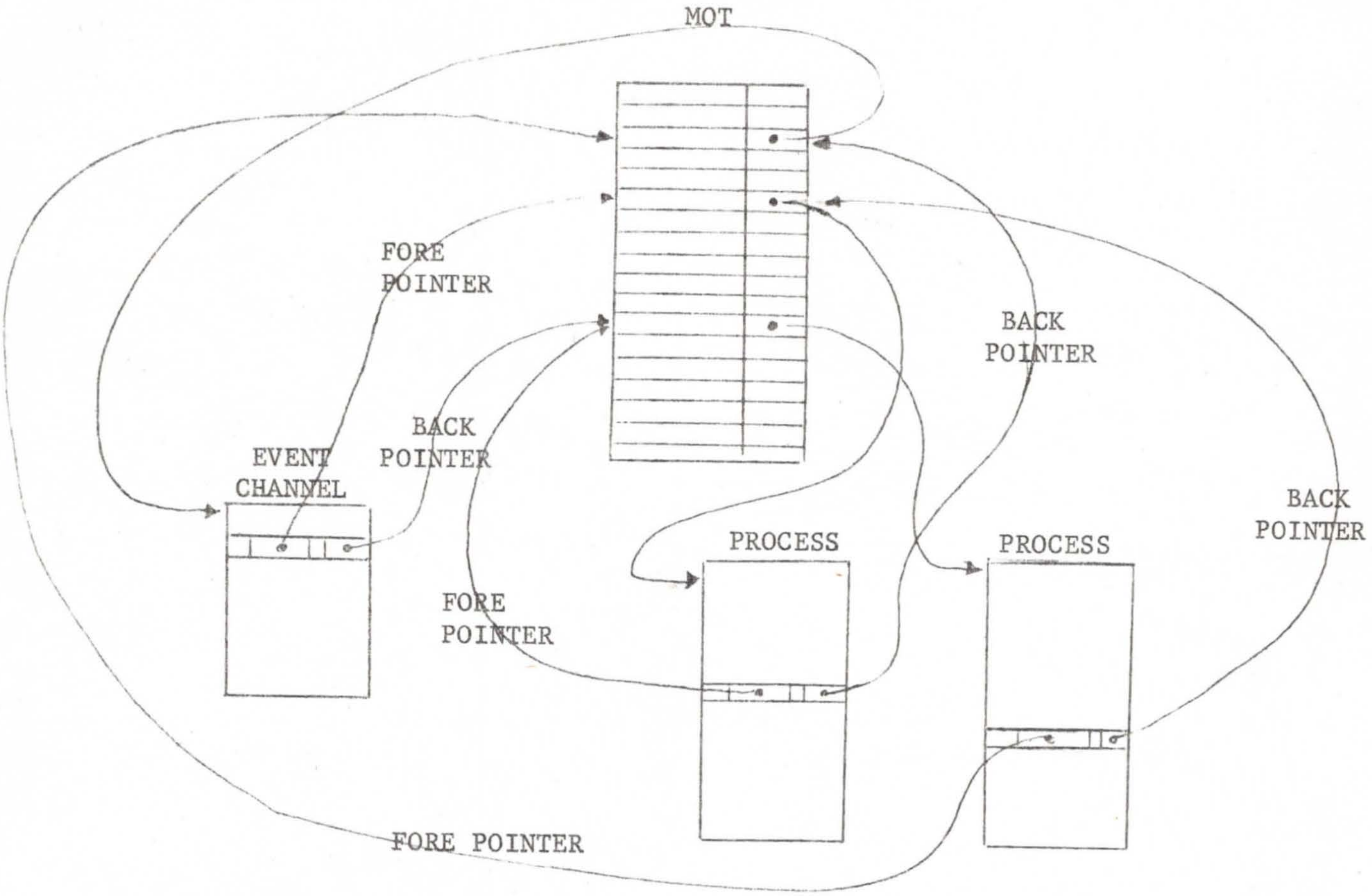


Figure 3

RESPONSES TO EVENT SENDER

CONDITION	RESPONSE
EVENT PUT IN EVENT QUEUE	1
EVENT PASSED TO A PROCESS	2
"YOU LOSE" EVENT PUT IN QUEUE	3
EVENT QUEUE FULL	4
~~DUPLICATE EVENT FOUND~~	~~5~~

TIME SHARING SYSTEM TEXT STANDARD

The System Standard Text (Systext) is the standard method of storing coded information for the Time Sharing System. Information in Systext format exists in a file (a semi-infinite array of 60-bit words) and is terminated by an end-of-information word. A Systext file is composed of lines, which contain character coded information, and segments which contain no information and are called sloppy segments.

Systext Lines

A line is a sequence of 7 bit ASCII characters terminated by the control character CR (= 155_8). There is no limit to the length of a line and they may be split across file block boundaries. Each line is packed left-justified into successive 60-bit words, 8 characters (56 bits) per word. The first 4 bits of each word serve to signal the beginning of a line: for the first word of a line these leading bits are 1001; for all other words in a line they are 0000. Consider the line ABCDEFGHIJ CR which would be stored in Systext as:

1001	A	B	C	D	E	F	G	H

0000	I	J	CR	*	*	*	*	*

Characters which follow the appearance of CR in a word are ignored.

Multiple blanks in a line are compressed by inserting a count of the number of blanks rather than the blanks themselves. The ASCII character ESC ($=173_8$) is reserved for this purpose. Whenever ESC occurs in the Systext file, the character following it is interpreted as a blank count, 'n' ($0 \leq n < 128_{10}$). On output these two characters are replaced by n blank characters.

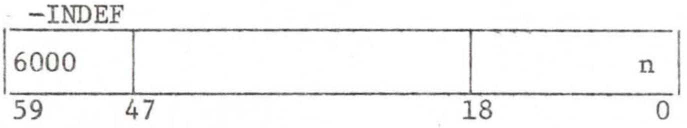
Character Representation

The internal ASCII code used in System Standard Text is the external ASCII + 140_8 (mod 200_8). The conversion is performed by the system I/O routines (see

p. 73). This scheme maps blank onto 0, 0 onto 20_8 and A onto 41_8 . See Table 1. Non-graphic characters, however, are not allowed to occur in System Standard Text. (CR and ESC in the contexts described above are the only exceptions.) Therefore, the character % has been reserved as a special prefix for representing non-graphic characters; if the graphic following a % maps onto a control character under the mapping: internal ASCII + 100_8 (mod 200_8), the pair is interpreted as that control character (see Table 2). Otherwise the % leaves its successor unchanged. So %% represents % and %M represents CR .

Sloppy Segments

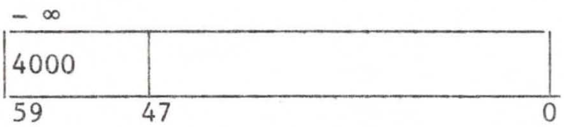
A sloppy segment in the Systext file is a group of n words $(0 < n < 2^{18})$ that are to be ignored. The first and last word of such a segment is of the form:



where n is the count of words in the segment. The system ignores the middle 30 bits of this header word and the succeeding n-1 words. A sloppy segment may not occur within a line and cannot be split across file block boundaries.

End-of-information

The end of Systext is signaled by an end-of-information (EOI) word of the form:



The low order 48 bits of the word are ignored.

Table 1

Graphic TTY Character Representation

TTY Character	Internal ASCII Representation	CDC Printer Character
b̸	0	b̸
.	1	v
"	2	↓
#	3	≡
$	4	$
%	5	%
&	6	^
'	7	≠
(	10	(
)	11	)
*	12	*
+	13	+
,	14	,
-	15	—
.	16	.
/	17	/
0	20	0
1	21	1
2	22	2
3	23	3
4	24	4
5	25	5
6	26	6
7	27	7
8	30	8
9	31	9
:	32	:
;	33	;
<	34	<
=	35	=
>	36	>
?	37	≥
@	40	≤
A	41	A
B	42	B
C	43	C
D	44	D
E	45	E
F	46	F
G	47	G
H	50	H
I	51	I
J	52	J
K	53	K
L	54	L
M	55	M
N	56	N
O	57	O
P	60	P
Q	61	Q
R	62	R
S	63	S
T	64	T
U	65	U
V	66	V
W	67	W
X	70	X
Y	71	Y
Z	72	Z
[	73	[
\	74	↱
]	75	]
↑	76	↑
←	77	→
'	100	≠
a	101	A
b	102	B
c	103	C
d	104	D
e	105	E
f	106	F
g	107	G
h	110	H
i	111	I
j	112	J
k	113	K
l	114	L
m	115	M
n	116	N
o	117	O
p	120	P
q	121	Q
r	122	R
s	123	S
t	124	T
u	125	U
v	126	V
w	127	W
x	130	X
y	131	Y
z	132	Z
{	133	(
|	134	
}	135	)
~	136	b̸
rubout	137	b̸

Table 2

Non-Graphic TTY Character Representation

Character	Internal ASCII Representation	Key Combination Systext Representation	Function
NUL	140	% @	
SOH	141	% A	
STX	142	% B	
ETX	143	% C	
EOT	144	% D	
EN	145	% E	
ACK	146	% F	
BEL	147	% G	Bell
BS	150	% H	Backspace
HT	151	% I	Horizontal Tab
LF	152	% J	Line Feed
VT	153	% K	Vertical Tab
FF	154	% L	Page Eject
CR	155	% M	
SO	156	% N	
SI	157	% O	
DLE	160	% P	
DC1	161	% Q	
DC2	162	% R	
DC3	163	% S	
DC4	164	% T	
NAK	165	% U	
SYN	166	% V	
ETB	167	% W	
CAN	170	% X	Delete Line
EM	171	% Y	
SUB	172	% Z	
ESC	173	% [	
FS	174	% \	
GS	175	%]	
RS	176	% ↑	
US	177	% ←	

THE LINE COLLECTOR

The line collector collects a line from the TTY using the previously typed line as a template. It maintains two lines simultaneously, an old one and a new one. The old line is the last line received by the Teletype (or from INITIAL) and is local to the virtual TTY buffer; it may possibly be empty. A new line is constructed from the old one using the characters typed in from the Teletype. To visualize the process of constructing each new line, imagine two cursors or pointers, one called OLD which runs over the old line and one called NEW which is positioned on the new line as it is created. Normally when a character is entered from the TTY, it is appended to the new line and both cursors advance on place. If certain non-graphic characters, called Control Characters (see Table 3) are entered, the cursors can be manipulated so that, for example, characters are COPIED from the old line to the new one, or parts of the old line are SKIPped, or the cursors BACKUP over undesired characters.

The most obvious application for the line collector would be in conjunction with an on-line compiler which performs a simple syntax check of each line as it is entered. If the line is bad it output a diagnostic, rejects the line, and calls on the line collector. The user edits the old line which still resides in the virtual buffer and resubmits it to the compiler.

The line collector permits the following actions to be performed via the appropriate control characters*:

Operation	Control Characters*	Action
Accept	CR	The current new line is accepted as is.
Type State	CTRL SHIFT \ k	Advances the printed paper to a fresh line. Spaces to the current position of the New cursor, prints a copy of the remainder of the old line, and on the following line prints a copy of the new line up to the current position of the cursor. e.g.: remainder of old line current new line ↑ (New cursor)
Concatenate and Accept	CTRL ← O	Concatenates the remainder of old line onto the current new line and accept.
Concatenate, Print and Accept	CTRL @ P	Concatenates the rest of the old line onto the new line, prints it out, and accepts.
Tab Set/Release	CTRL \ k	Sets (releases) a tab stop at the current position of the cursor in the new line if entered an odd (even) number of times.
Tab	CTRL TAB I	Inserts blanks up (both cursors advance) to the next tab stop.
Insert Change:	CTRL L	If entered an odd number of times since the beginning of the first line, the cursor in the old line is not moved on Backup or normal entry operations, thereby allowing the insertion of characters into a line. Odd numbered entries of the control characters are echoed by < . Even numbered entries return the cursor to its normal action and are echoed by > .

* If the first key specified is CTRL , the second key must be pressed while the first key is still depressed.

Concatenate and re-edit	CTRL SHIFT L	Concatenate new line with remainder of old line and make the concatenation the old line, positioning the cursor at its beginning.
Special CR	CTRL U	Identical to CR except that the line collector notifies the calling routine that this line is special. (Can be used to switch modes, i.e., to leave "append mode" in the Editor.)
Panic	CTRL SHIFT @ P	Interpreted by the PP, this command sends an interrupt to the process. May be used to get a process out of a loop or to get its attention.

For each of the three actions Backup, Copy, and Skip, the distance can be specified in 6 ways (see Table 3). In the descriptions which follow, a word is defined as a sequence of one or more non-alphanumeric characters delimited by non-alphanumerics; when looking for the beginning of a word, the cursor passes over all non-alphanumerics until it encounters one or more consecutive alphanumerics. Next character entered refers to the first occurrence in the line of the next character typed in after the control characters. If at any time an edit request is made which cannot be fulfilled, the line collector echoes a bell instead of the graphic specified.

Operation	Control Characters	Action
Backup one character	CTRL Q	Cursor in the new line backs up (erases) one character* ← is echoed on the printer.
Backup one word	CTRL W	Cursor in the new line backs up (erases) one word* ← is echoed once on the printer.
Backup to next character entered	CTRL WRU E	Cursor in the new line backs up (erases) up to but not including the new character entered* ← is echoed on the printer.
Backup to and including next character entered	CTRL TAPE R	Cursor in the new line backs up (erases) up to and including the next character entered* ← is echoed on the printer.
Backup to tab	CTRL TAPE T	Cursor in the new line backs up (erases) up to the preceding tab setting* ← is echoed on the line printer.
Backup to edge	CTRL Y	Cursor in the new line backs up (erases) up to the left edge, thereby starting the line anew* ← is echoed on the line printer.
Copy one character	CTRL A	The next character in the old line is appended to the new line, and the character is printed.
Copy one word	CTRL XOFF S	The next word in the old line is appended to the new line and is printed.
Copy up to next character entered	CTRL EOT D	Characters in the old line up to but not including the next character entered are appended to the new line and printed.

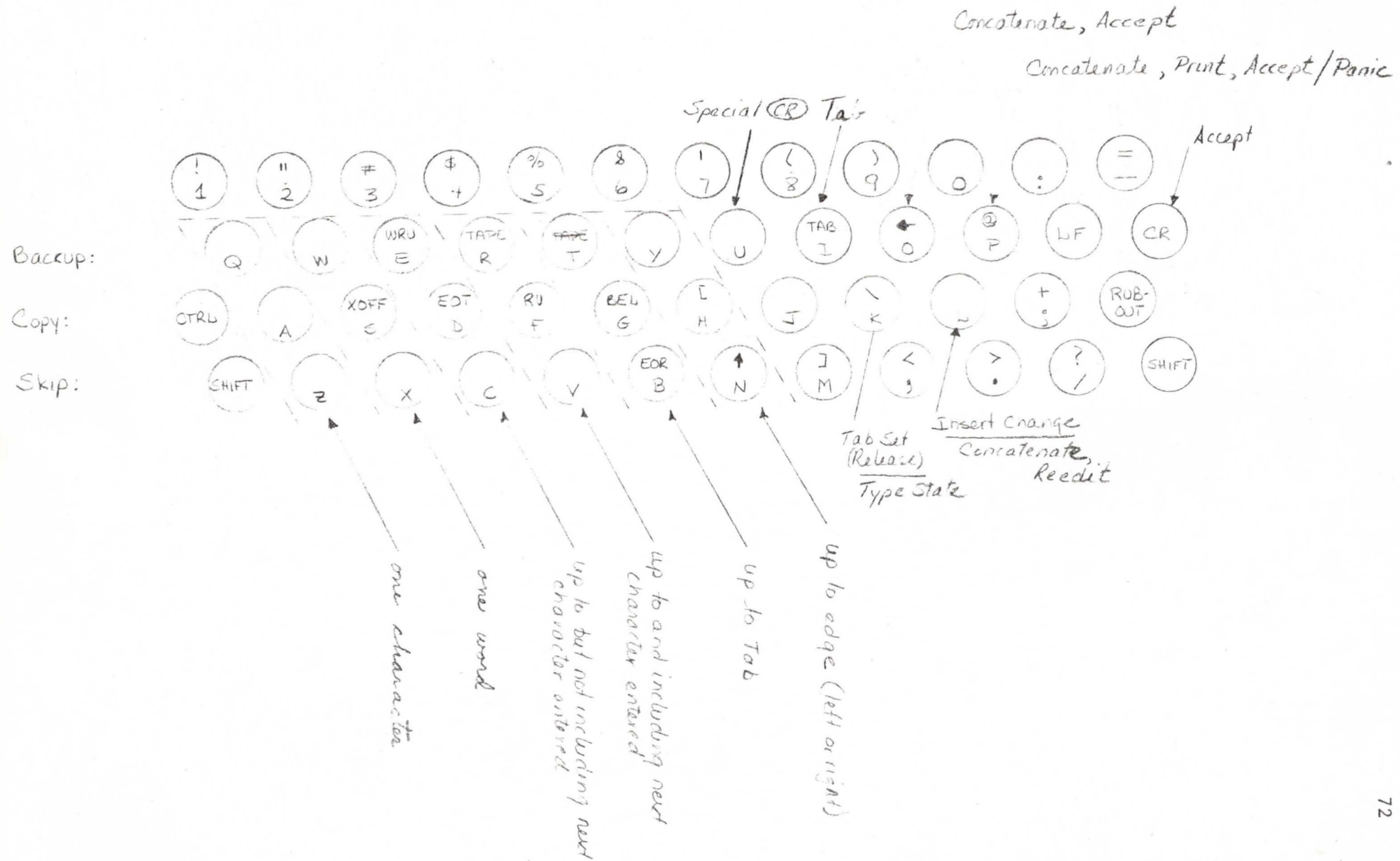
* The old cursor moves simultaneously with the new cursor.

Copy up to and including next character entered	CTRL RU F	Characters in the old line up to and including the next character entered are appended to the new line and printed.
Copy to tab	CTRL BEL G	Characters in the old line up to the next tab setting are appended to the new line and printed.
Copy rest of old line	CTRL [H	The remainder of the old line is appended to the new line and printed. Note that CTRL H CR is equivalent to CTRL ← O above.
Skip one character	CTRL Z	Cursor in the old line moves ahead (skips) one character* $ is echoed on the printer.
Skip one word	CTRL X	Cursor in the old line moves ahead (skips) one word* $ is printed for each character skipped.
Skip to next character entered	CTRL C	Cursor in the old line moves ahead (skips) to but not including the next character entered* $ is printed for each character skipped.
Skip up to and including next character entered	CTRL V	Cursor in the old line moves ahead (skips) to the position immediately after the next character entered.* $ is printed for each character skipped.
Skip to tab	CTRL EOR B	Cursor in the old line moves ahead (skips) to the next tab setting.* $ is printed for each character skipped.
Skip to end of line	CTRL ↑ N	Cursor in the old line moves ahead (skips) to the end of the line* $ is printed for each character skipped.

* The cursor on the new line moves simultaneously with the cursor on the old line.

Table 3

(33/35) Teletype Keyboard and Control Characters



Teletype I/O Functions

The TS System I/O functions are a set of reentrant routines which should be loaded into a continuous section of core. If absolute images are used, they must reside in the right part of core. To initialize these functions, one jumps to .TTY. ON with

B1 set to the base of a 133_8 CM word data area (TTYBUFF) for this teletype.

B2 set to the index in the C-list for the TTY file.
(B2)+1 is the index of the CP to PP event channel
(B2)+2 is the index of the PP to CP event channel.

X7 is set to the return address in calling program.

I/O operations are performed upon strings or lines where a string is a sequence of characters and a line is a string terminated by a CR character. Every string or line is quantified by a two word entity called a <u>string descriptor</u>. The first word of a string descriptor points to the word base address of a given string; the second word indicates the length of the string, or for a line, the upper bound on the length, since the terminating CR character signals the end of a line.

Output

To output a string described by the string descriptor DESC, DESC+1 the following macro call is invoked:

```
.PUTOUT        MACRO          TTYBUFF, DESC
               SB1            TTYBUFF             The data area for the TTY
               SA4            DESC+1
    +          SX7            *+1
               JP             PUTL
               ENDM           .PUTOUT
```

.PUTOUT outputs characters up to and including a CR or until the length specified in the second word of the descriptor is exceeded, whichever occurs first. Lines with blanks compressed as well as uncompressed lines may be output by .PUTOUT. If a CR is encountered, a LF is also echoed.

NOTE: If the flag at TTYBUFF + FORCE (FORCE = 23_8) in the TTY data area is up, the TTY buffer will be flushed (PP is notified that there is something in the buffer) each time .PUTOUT finishes. This kind of call-by-call flushing is expensive and should be suppressed when possible. Therefore, if a large file is to be listed, the FORCE flag should be turned off until the last line. With the flag off, lines will be forced out only when the TTY buffer becomes full. Initialization leaves FORCE up.

A single character is output when a macro call to .OUTPUTC is invoked:

```
.OUTPUTC        MACRO           TTYBUFF, CHAR
                SB1             TTYBUFF
                SX1             CHAR
    +           SB7             *+1
                JP              PUTCTTYT
                ENDM           .OUTPUTC
```

The output buffer is flushed when a macro call to FLUSH is invoked:

```
FLUSH           MACRO           TTYBUFF
                SB1             TTYBUFF
    +           SB7             *+1
                JP              FLUSH
                ENDM            FLUSH
```

Input

Teletype input is significantly more complex than output. The routine INGET is called to get a line from the TTY:

```
INGET           MACRO           TTYBUFF
                SB1             TTYBUFF
    +           SX7             *+1
                JP              GETL
                ENDM            TTYBUFF
```

INGET causes a new line to appear as the string described by the string descriptor stored at TTYBUFF + NEW (NEW = 101_8). This new line does not yet have blanks compressed and the first four bits of each word are zeros. INGET obtains the new line from the teletype using the line described by the descriptor TTYBUFF + OLD (OLD = 76_8) as a template. To modify the template merely involves updating the OLD descriptor and its image with desired new line. The line must not exceed 86 characters in length since that is the maximum length of a line which INGET can return.

A call to the following macro enables the user to detect the reserved control character % U .

```
INGET.          MACRO           TTYBUF, COMMAND
                SB1             TTYBUF
                SX7             COMMAND
                LX7             18
                SX6             *+2
+               BX7             X6+X7
                JP              GETL
                ENDM            INGET.
```

If the line gotten from the TTY buffer is terminated by % U instead of CR , then control returns to COMMAND rather than *+1 . This allows the TTY to earmark certain lines as special. For instance, consider a file editor which allows lines to be appended to a file. There must be a way for the user to signal which line is the last line to be appended to the file. However, every key has a pre-assigned meaning or can appear in a line; the only exception is % U . Thus the editor could designate % U to terminate the last line of the file and control will return to COMMAND.

The input buffer can be cleared (the contents are removed and discarded) by a macro call to CLEAR:

```
CLEAR           MACRO
                SB1             TTYBUF
+               SB7             *+1
                JP              .CLEAR
                ENDM            CLEAR
```

Since these routines should suffice for most circumstances, the following esoteric features can be ignored by the majority of users.

The routine GETS concatenates characters up to and including the next break character (see p. 4) onto the string described by the string descriptor DESC.

All but the break character are echoed; the break character is returned in X1. GETS is called as follows:

```
GETS            MACRO           TTY,DESC
                SB1             TTY
                SA4             DESC+1
                SB6             1
+               SX7             *+1
                JP              GETS
                ENDM            GETS
```

There is one anomoly connected with GETS; if no check were provided, it would be possible for GETS to accept a string that was long enough to clobber storage when it was concatenated onto the string described by DESC. To avoid this, GETS expects DESC+2 to contain an upper bound on the length of the resulting string. If GETS receives a string which when concatenated would exceed this upper bound, it returns in X.CHAR the negative of the first character in the string which causes the bound to be exceeded.

The routine GETCTTY gets the next character from the TTY buffer placing it in X1; it is called as follows:

```
GETCTTY         MACRO           TTYBUF
                SB1             1
+               SB7             *+1
                JP              GETCTTY
                ENDM            GETCTTY
```

GETCTTY does not echo the retrieved character even if the SOFTECHO (= 21_8) flag in TTYBUFF is on. (The SOFTECHO flag signals that the PP has not been able to echo a character and therefore that GETS should.)

The macro call to NEWBREAK is used to switch from one table of break characters to another.

```
NEWBREAK        MACRO           TTYBUFF,I
                SB1             TTYBUFF
                SB2             I
+               SB7             *+1
                JP              NEWBREAK
                ENDM            NEWBREAK
```

If the break table is switched, it should be restored to break table #2 before using GETL. Other routines will work with any break table.

Table Number	Characters which signal a break
0	none
1	any character
2	non-graphics
3	non-alphanumerics
4	non-numerics

~~July 22~~ Sept 2, 1969

TS Interrupt System

Introduction

The Interrupt System, which provides the sole interface between user processes and the outside world, is divided into two parts, the Central portion consisting of the code proper, and the PPU portion comprising the actual communication with the external devices.

The Interrupt system uses several objects which reside in ECS:

1. Files and event channels, which provide the immediate interface between the ECS System and the Interrupt System. They are seen by user level processes and appear just like ordinary files and event channels, except that they are stationary in ECS.

2. Pseudo-processes, which are used by the Interrupt System to simulate processes for hanging on event channels. (They are also used as a convient storage place for information not being used.) Pseudo-processes are seen by user level processes only in that events are taken from, and placed on, event channels by something unknown.

I Central Memory Portion

There are two sections to the Central code of the Interrupt System. The first, Initialization, sets up ECS at the beginning of time, sets up a few things in Central Memory and disappears. The second section consists of a collection of routines which work with individual devices, plus some miscellaneous "stuff".

1.1 Initialization of ECS

The routine INTINIT constructs in ECS all objects needed by the Interrupt System. This is done at the beginning of time so that they will be in a position in ECS such that they never have to move. When INTINIT is called, there must be no gaps in ECS.

INTINIT is called at 2 different times:

1. At INTINITA , early in initialization before more than a very few things have been constructed. It is used to construct a file to contain C-list indices into the master C-list of interesting things constructed later.

2. At INTINITB for the actual construction of objects. It first constructs a file to be used for the interrupt queues needed by UNHUNG1. (See 1.2.2) (No user ever sees this file, and in fact, its Master C-list entry is destroyed.) It then calls the Device Routines to construct the objects for each of the various kinds of devices. Currently there are two separate kinds of devices: MUX and the simple devices.

The Device Routines first call the routine NEWCLASS, which initiates construction of a class of objects. (Each type of device is considered as a class of objects, associated with which is an interrupt queue and a file in ECS containing pointers to the pseudo-processes. See Figure 1.)

NEWCLASS - This routine sets up all classes of objects. It expects five parameters:

1. the location of the Interrupt Queue Index within the prototype pseudo-process for this class of objects,

2. the location in Central of a pointer in ECS to the file containing the locations of the pseudo-process for this class,

3. the location in Central pointing to the interrupt queue in ECS for this class of devices,

4. the interrupt index for this class of devices,

and 5. the number of devices in the class.

The Device Routines then call MEC, MPC, and MFILE which are specially provided for INTINIT, to construct the particular objects in ECS used by the Interrupt Routines.

MEC - This subroutine constructs an event channel and leaves the capability in the Master C-list. It is entered with the count of the maximum number of events expected in the event channel at one time.

MPS - This subroutine constructs a pseudo-poocess. It is entered with the size of the process in words. It makes no permanent Master C-list entry; the Master C-list entry is destroyed at the end. It returns with the absolute address of the psuedo-process in X5 and the MOT index and unique name in X4.

MFILE- This subroutine constructs a one level file with a data block whose size is given in X6. It leaves the entry in the Master C-list and returns with the absolute ECS address of the data block in X0.

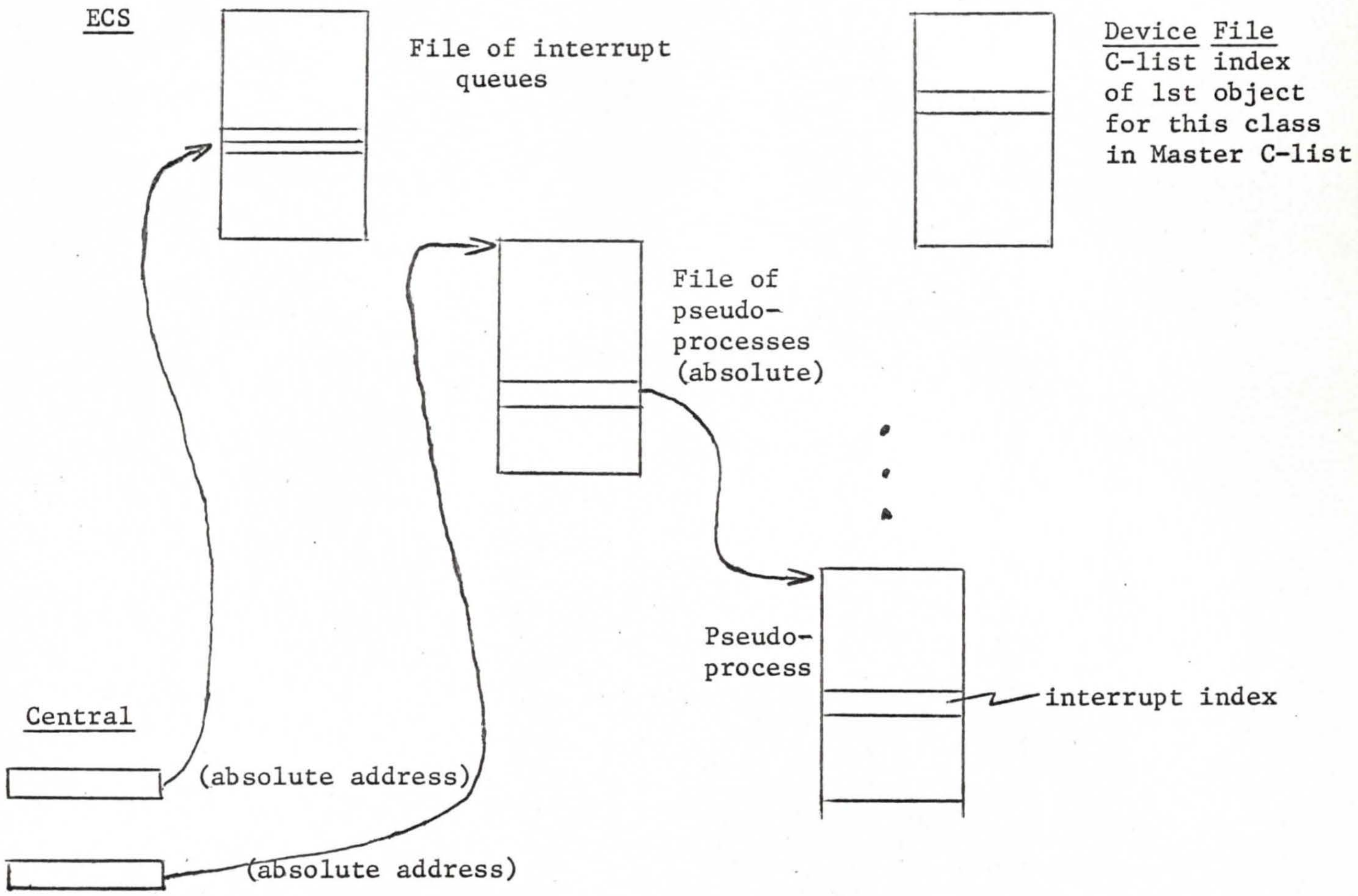


Figure 1

In summary, the basic function of INTINIT is to create a file in ECS available to user processes which contains the first C-list index of an object belonging to each class of interrupt devices (INTINIA), to create a file (never seen by a user process, in fact removed from the master C-list) which

1.2 Interface between the Interrupt System Central Code and the ECS System Central Code

1.2.1 Calls from the Interrupt System to the ECS System

HANG - This routine is called to hang a pseudo-process (or process when called by other routines) on an event channel. It expects the following parameters:

1. the address of a scratch area it can use
2. a queuing word index to use, found in the pseudo-process,
3. identification of the pseudo-process
4. identification of the event channel to be used.

EVENT1 - This routine places an event on the specified event channel. It expects the following parameters:

1. event channel to be used
2. identification of process or pseudo-process sending the event
3. event datum
4. origin of scratch area including an address relative to this origin to which the disposition of the event is returned.

1.2.2 Calls from the ECS System to the Interrupt System

UNHUNG1 - This routine signals the arrival of an event to a pseudo-process. It expects the following parameters:

1. return address
2. origin of scratch area
3. the absolute address in ECS of the pseudo-process
4. the event received.

UNHUNG1 looks into the pseudo-process for data: first it uses the address specified in word 4 of the pseudo-process to chain it on an interrupt queue designed for each particular device. The interrupt queue is maintained by two words in a file in ECS. (See Figure 1.) The first word points to the absolute address of the first pseudo-process in the queue, and the second word points to the last one in the queue. Pseudo-processes are chained on the queuing word (word 2) in the pseudo-process.

Next UNHUNG1 takes the Interrupt Index (also found in word 4 of the pseudo-process), which points to a particular device, and stores it in I.WAKE when I.WAKE has gone zero.

The Interrupt System calls this routine when it has tried to hang a pseudo-process on an event channel (using HANG) and gets an event back immediately.

Figure 2

1st Part of Pseudo-Process

<table>
<tr><th>59</th><th>56</th><th>54</th><th>48</th><th>40</th><th>36</th><th>18 0</th></tr>
<tr><td>1</td><td>1</td><td>0 0</td><td colspan="2">7 7 7 7</td><td>MOT index of the Pseudo-process</td><td>0 0 0 0 0 0</td></tr>
<tr><td colspan="7">Queuing word (Pointer to next pseudo-process - used by event channels and interrupt queues)</td></tr>
<tr><td colspan="7">Zero word (stops chaining words - used by event channels)</td></tr>
<tr><td colspan="4">0 ——— 0</td><td colspan="2">Interrupt queue address</td><td>Interrupt Index</td></tr>
<tr><td colspan="7">Event word 1 (placed here by UNHUNG1)</td></tr>
<tr><td colspan="7">Event word 2 (placed here by UNHUNG1)</td></tr>
<tr><td colspan="7">.
.
.</td></tr>
</table>

1.2.3. Interlock Facility

The Interlock Facility is used to prevent interrupt code from referencing event channels while ECS is doing so. The cell I.LOCK is set non-zero by the ECS system whenever the ECS system is about to work on event channels, and is set zero when the work is completed.

Currently, the interrupt system always checks the cell upon entry to any of its code, and if it is non-zero, quits immediately. (Eventually the interrupt system could be more discriminating and only check I.LOCK when it was about to work on event channels. This could be used by interrupt routines desiring immediate access to a file and only a file.)

1.3 The Central Interrupt Routines

The basic operation of an ordinary interrupt routine involves the following actions:

> If working with event channels (or if the coder was lazy), an Interrupt Routine first checks I.LOCK. If I.LOCK is non-zero, the routine must promptly jump to location zero within a few microseconds (like 4 or 5). If I.LOCK is zero, the routine proceeds to do whatever it was planning to do. When finished, it jumps to zero, signalling the end of the interrupt.

An interrupt routine gets a pseudo-process off of the appropriate interrupt queue by calling DINTQ, with the absolute address in ECS of the queue. DINTQ either returns with an indication that there were no pseudo-processes on the queue, or it unchains the first pseudo-process and returns its absolute address. (The event can be found in the pseudo-process.)

II The Peripheral Processing Unit Portion

There are two areas to the PPU portion of the Interrupt System. The first, the Master PPU, serves to synchronize the Interrupt System. The second area consists of the indiiidual PPUs which handle the individual devices. In some instances, several devices are handled by one PPU, and in at least one instance (the disk) one device is handled by 2 PPUs.

2.1 The Master Peripheral Processing Unit (MPPU)

The master PPU handles the synchronization of the Interrupt System with a large loop, starting at MLOOP, which performs the following actions by means of a succession of return jumps:

1. Calls a routine which checks for the status of the user; e.g. arith errors, or RA+1 $\neq 0$ (indicating a simulated SCOPE call). If either of these two conditions holds, the PPU calls the ECS system via a monitor exchange jump (MXN).

2. Checks I.WAKE to see if there are any calls on the interrupt system from the ECS system.

3. Checks a channel, INTCHAN (as spelled in listing for MPPU), for calls on the Interrupt System from the other PPUs.

4. Calls a routine to update the master clock in Central (S.MASTR) which is run in steps of one microsecond and contains the true time in microsecond since the system was started. This routine must be entered at least once every 4 milliseconds.

5. Calls a routine to update the clock S.QUANT which signals the end of a quantum for a user program by going positive (over-flowing). In this case, a monitor exchange jump (MXN) is sent to Central.

6. Calls a routine to update a charge clock, S.CHARG, which is updated whenever user code or system code, but not interrupt code, is running in Central.

7. Calls the routine DOINT to check a table for pending interrupts. (MPPU maintains in the table a list of those interrupt routines which have been signaled via either the ECS system or INTCHAN and have not yet been called.) If they are sending interrupts, it scans the table for the first one pending and having found it, finds the P-counter in a table in Central, copies it into an exchange jump package located in Central (at I.BOX in the routine GENLINT), and then performs an EXN to that package. Since the table is ordered by interrupt number, those with low interrupt numbers are called first. It then enters a short loop of 12-24 milliseconds and checks the P-counter. If it is zero, MPPU assumes that the Interrupt was unsuccessful due to I.LOCK being non-zero when checked by the Interrupt Routine. It then goes away, and will make this interrupt call later. If the P-counter is non-zero, it assumes that the interrupt routine is running. It then continues cycling through this short loop, watching for the P-counter to go to zero, checking now and then for new interrupt requests coming in on INTCHAN or in I.WAKE , and recording them. It also maintains the master clock (but no other clocks). When the P-counter goes to zero, it restarts Central with an Exchange Jump (EXN), and updates the master clock (S.MASTR) and charge clock (S.CHARG) to compensate for sloppiness at the beginning and end of the routine.

2.2 General overview of How a User Event is Transmitted into Action by an Interrupt Routine

1. The user sends the event to the event channel.

2. If the event channel routines detect the fact that there is a pseudo-process hung on that event channel, they unchain that pseudo-process from the event channel and transfer control to UNHUNG1.

2.2 UNHUNG1 looks into the pseudo-process (word 4) and finds that the interrupt index is - which it stores into I.WAKE.

2.3 UNHUNG1 finds (word 4) what the absolute address in ECS of the interrupt queue is and chains the pseudo-process into that queue as described.

2.4 UNHUNG1 also places the events in event word 1 and 2 in the pseudo-process.

2.5 The master PPU then discovers I.WAKE ≠ 0, records this fact in its own tables and sets I.WAKE back to zero.

2.6 When a suitable time occurs (hopefully before too long), it does an XJ to the appropriate interrupt routine as determined by its own tables.

2.7 The interrupt routine then does various things, including calling the general routine, DINTQ , which takes the pseudo-process off the interrupt queue and passes it back to the interrupt routine.

3. Finally, if when the event was received by the event channel, there were _no_ pseudo-processes hanging, the event is stored on the event channel queue, and later, when the interrupt routine desires to hang a pseudo-process on an event channel, the event channel routines return with the event, and UNHUNG1 is called by the code associated with the Interrupt routines themselves.

11/5/69

S-device user interface

This interface will be used for devices such as tape drivers, printers, card readers, card punches and the console display. For each devices the interface consists of one file and two event channels. The 2 event channels are called req and rsp.

An individual request on a device will be a call for a certain action to be taken on one or more buffers within the file. The buffers will be specified by giving the address within the file of the first word of the first buffer, the size of the buffers (all must be the same size), and the number of buffers. Also associated with each request will be an error recovery bit which must match the value of an error recovery flag associated with the device. Each time an error occurs on the device, the error recovery flag will be changed in value. Thus more than one request may be sent at one time, and if an error occurs on an early one, then the rest will be ignored. Finally, the request will contain an index to allow the user to associate responses with requests.

A response will contain a bit to signal the presence or absence of an error. An indication will also be given if the request was ignored, either for bad error recovery bit, or bad action. If the request was not ignored, a count will be returned to indicate how many buffers were acted upon. If an error occurred, it occurred on the last buffer acted upon. The response will contain the index of the associated request. Finally, the response will contain 2-12 bit bytes of status information, which for devices on the 6681 will be the 6681 status and the device status.

The actual request is made by sending the following event on the event channel req.

1	11	12	12	12	12	Field Size
6	5	4	3	2	1	Field number

Field Contents

1. action code, request will be ignored if equals 0
2. file address of 1st word of 1st buffer
3. buffer size
4. count (number of buffers to be acted on)
5. user index
6. error recovery bit, request will be ignored if does not match error recovery flag associated with the device.

The actual response will be the following event on the event channel rsp.

12	12	12	12	12	Field size
5	4	3	2	1	Field number

Field Contents

1. user index in request
2. 1st status byte (for 6681 devices, 6681 status)
3. 2nd status byte (for 6681 devices, device status)
4. count of buffers acted on
5. 3777B if request is ignored [fields 2,3 and 4 will equal 0]
   4000B if an error occurred
   0000B if no error occurred

ECS LIST

ECS system

~~1) destroy event channel~~

~~2) zero level files? → (see 33)~~

~~3) put event without duplicate checking (remember interrupt interface)~~

~~4) get event with chaining word index masked in~~

(change MOT index to unique name in X6)

½ ~~5) ~~parameter~~ indirect C-list reference~~

~~6) get event or F return~~

I Dave ~~7) find descendant of subp~~

Bruce ~~8) set temp part of class code~~ (not in system!)

} ~~written~~ not debugged

~~9) subprocess entry points~~

~~10) put check in PUTACT and PUTECS for length of action~~

Later ~~11) update cpu $ in alloc block~~

) reset end of path to self (restore) } written

~~13) exit mode - ~~create process bits in wrong place~~~~ } got correction

~~14) hang on multiple event channels~~

½ ~~15) option bits~~ Vance

D 16) ~~garbage collector~~ & allocation mod 64 Vance

~~17) file → file copy~~ never

see 26 ~~18) delete object (not already in)~~

19) fast actions

D ~~20) move block, check maps ref count?? Paul (return dirty bit in X6)~~

~~21) error return (return w/ error max error class = 3)~~

I 22) map compiler now does disaster if missing map block should do error (F-return) Paul & Bruce

~~23) option bit for change unique name~~

? ~~24) fix up CCLOA~~

? ~~25) date & time (real - interval)~~

26) general destroy: ? ~~27) move from one alloc to another~~ : ~~29) Type any not special case~~ Bruce

28) Send interrupt to pseudo-process (written w)

~~30) 12 Jan '70 Move CLASSCNT to ECS~~

31) 4 Feb Check GARBCNT in Subprocess environment establishment at the point of doing a DAE map entry

? ~~32) 12 Feb Move allocation block operation?~~

~~33) 17 Feb check what files do for 0 level files~~ Paul

Feb ~~34) 20 Feb Fix Capab re ≤0 length Clist~~

~~35) 20 Feb Fix error returns from OPINTR~~

36) 20 Feb F: return when delete subproc not a leaf (Bruce

I 37) 1 Mar Get more buffers going for keeping ~~stuff~~ system code in ECS

38) In process & subprocess creation, correct test of lower limit for entry point (written w)

~~D 39) 16 Mar Change PROBE operation to return # map refs in X7; see U.M. for specs.~~ Paul

~~40) 16 Mar design & implement display process descriptor operation~~

D41) 27 Mar Turn (off / on) all map entries for a subprocess

~~D 42) 30 Mar Block parameters < Data & capab~~

~~Return parameters~~ Bruce

43) 30 Mar Environmental map compilation Paul

Interrupt to Pseudo process

Pass: class code = X ~~1~~ 2
interrupt datum X ~~2~~ 1
return link = B7 (SYSRET)

lock interrupts & then call routine w/ ~~index into table found in~~ datum found in 4th wd of the pseudo-process

— Howard must fix up IPROC
to save capabilities for pseudo processes

questions:

✓ 1) pseudo ✓ before incore ✓ ?

Done 2) unique name = class code ?

✓ 3) is the 4 OK in RF ?

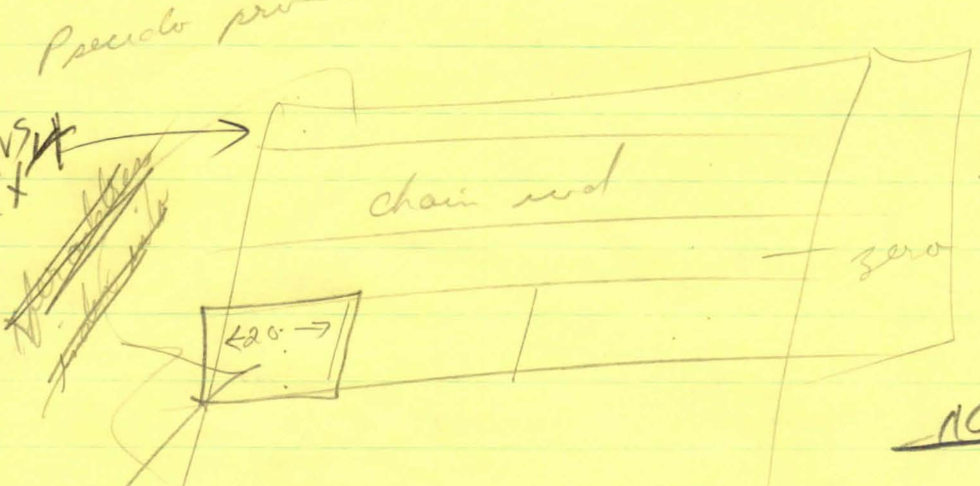
✓ 4) position of interrupt lockout ?

✓ 5) JP to NAME

NO ✓ 6) class code + MOT ptr in X2 OK ??

✓ 7) PUTINT96 change

Pseudo proc



Howard can retain P.P. capabilities by calling incap in INITL

pass in X3

1) Name
2) C(X3)
3) destroy pseudo-process??
4) ~~SYT~~ ...

Modification of "process stuff" to send an interrupt to a pseudo-process.

This code to replace PROCESS00783 – BGL1A00055 inclusive

```
          .
          .
          .
          RE     4              Get first 4 words of process desc.
          RJ     E.ECS
          SA1    Aφ
          LX1    5
          PL     X1,PUTINTφ5    Skip if interrupt isn't to a pseudo process
          SA1    B1+PPARAM+4    interrupt datum to X1
          SX4    X5             ECS addr of pseudo proc. to X4
          SA2    B1+PPARAM+3    class code to X2
          SA3    B1+3           20 mystery bits to X3
LX3    20
MX6    20
BX3    -X6*X3
          SX6    1              kill PPU interrupts &
          SA6    I.LOCK           call Howard
          SB7    PUTINT3        ~~(will Howard clear I.LOCK??)~~
(declare external)  JP     PSUDINT



PUTINTφ5  LX1    1
          NG     X1,PUTINT8     check incore flag .....
                                ask Bruce about the comment
          .
          .
          .
```

This code replaces BGL1200013 thru PROCESS 00847 incl.

```
          BX6    X3             Exit if interrupt
          LX6    6                now impossible because
          NG     X6,PUTINT8       process is in core.
```

also change comment on PROCESS 00921 etc.

```
~~        IDENT     SB100A~~
~~        ENTRY     S100A~~
~~        EXT       XIDMP~~
~~NAME    VFD       30/5HSB100,30/2~~
~~S100A   DATA      0~~
~~        SB7       B1~~                              Loop addresses to B1, B2, B7
        SB~~4~~       ~~M161~~ NOPROC4
        SB2       ~~M162~~ NOPROC5 ~~SB2  B1+1~~
        SB3       16                             Loop count for 16 patterns.
        SA1       B~~4~~                             Pick up initial two patterns.
        SA2       B2
        MX5       0                              Clear sum initially.
NOPROC3
~~MLOOP~~   IX5       X5+X1                          Sum the 16 items of the lower list.
        BX6       X1                             Move patterns for storage.
        BX7       X2
        BX3       X2-X1                          Compare patterns
        NZ        X3,~~ERROR1~~ NOPROC7              Error if patterns don't match.
        SA1       B2-B3                          Pick up next two patterns
        SA2       B~~4~~+B3
        SA6       B2-B3                          Save last two patterns in new
        SA7       B~~4~~+B3                              locations.
        SB3       B3-1
        NE        B3,B0,~~MLOOP~~ NOPROC3
~~        BX7       X5~~                  ZR   X5, NOPROC6       Error if the lower list
~~        SA7       B7~~        NOPROC7   RJ   DISASTR           failed to sum to 0.
~~        JP        S100A~~
~~ERROR1  BX7       X3~~
~~        SA7       B7~~
~~        JP        S100A~~
```

```
~~USE           /COM100/~~
      DATA      000000000000
      DATA      000000000000
      DATA      077777777777777777777
      DATA      077777777777777777777
      DATA      052525252525252525252
      DATA      052525252525252525252
      DATA      025252525252525252525
      DATA      025252525252525252525
      DATA      0000000000000
      DATA      052525252525252525252
      DATA      077777777777777777777
      DATA      025252525252525252525
      DATA      052525252525252525252
      DATA      000000000000
      DATA      025252525252525252525
NOPROC4
~~M161~~  DATA      077777777777777777777   — B1,3
NOPROC5
~~M162~~  DATA      077777777777777777777   — B2
      DATA      025252525252525252525
      DATA      000000000000
      DATA      052525252525252525252
      DATA      025252525252525252525
      DATA      077777777777777777777
      DATA      052525252525252525252
      DATA      000000000000
      DATA      025252525252525252525
      DATA      025252525252525252525
      DATA      052525252525252525252
      DATA      052525252525252525252
```

✓1) Change B1 to B4.

~~2) Maybe don't use B7?~~

✓2) Change MLOOP & M161, M162

NO 3) Really just go to DISASTR???

RJ Z DISASTR

```
         DATA      077777777777777777777
         DATA      077777777777777777777
         DATA      000000000000
         DATA      000000000000
         ~~END~~
NOPROC6  EQU       *
```

This change ~~fot~~ in process swapper goes at NOPROC2+2

* The code starting here & ending at NIPROC6 is a memory test. It may be changed (or removed) at any time.

* Steps in the Memory Test.

* 1) If B3 =0, some massive, compensating failure took place & the ~~bit~~ pattern table has to be inspected for clues. This step is ~~[illegible]~~ unlikely.**N.

2) If B3 ~~≠0~~, is non-zero, ~~stp~~ then M162-B3 didn't match M161+B3 in the bit positions which are on in X3 at the time of the stop. X1 & X2 contain the

* offending patterns.

followed by code above

Process & Subprocess creation
correction

In Process at p4 L00100:

```
~~KEC~~    SA2    NC          Check that the entry ~~point~~
           SX3    X2-8        does not conflict with
~~KBE~~    NG     NC          cells of fixed usage in
                                        low subprocess are
           IX2    X2-X1
           PL     X2,MK36X    Error ... entry .GE. FL
```

In Subprocess at p3 L00065:

```
SA1   NC
SX2   X2-8
NG    NC
```

Currently in small ~~yellow~~ black binder for easy transport to machine room & back

2 CP disaster sheet

1) ~~processor~~ CP A is making a map entry, finds a block missing;

~~processor~~ CP B creates the block;

~~processor~~ CP A decrements map counts on blocks & goofs up newly created block + any that follow

2) ~~processor~~ CP A is making a process, finds all necessary blocks for map entries present;

CP B deletes one of the blocks;

CP A goes to disaster.

Speed phreaques, scheduling, compactification, ECS code, DAE, & other theological questions made simple.

SF's are supposed to give damn fast responses, like for a real-time somethingoranother.

Thus, when an SF is $\left\{\begin{matrix}\text{awakened}\\ \text{interrupted}\end{matrix}\right\}$, the scheduler has to do something snappy. The current guy has to be suspended & the SF fired up ~~in~~ within some short time ≤ l_1. Normally, system code being recontinued & all, enuf of the CM to accomodate the SF could be copied by brute force to ECS & the SF brought in & run. The CM is then restored & allowed to run. I know of some problems:

1) The CM may be in the middle of a system call which is executing ECS code in some buffer. The SF may wipe out the buffer. It seems like the contents of the buffers have somehow to be preserved.

2) ECS may be all bent 'cause compactification is in progress. The compactifeer has to be told to cool it. ~~& allowed~~ It will require some piece of time l_2 to get itself straight. In this sense, it must

be incremental.

3) The allocator isn't really reentrant.

The fight between l_1 + l_2 is obvious. ~~Two~~ Some things should be noted:

1) ~~If the~~ The SF's may may have to be recompiled.

2) The compactifier may be moving something gargantuan. Either

a) it is allowed to finish

or

b) a mechanism for half-moving something has to be thunk up.

3) The SF better hadn't cause anything to be allocated (even destroyed is annoying).

4) If there's more than 1 SF, things get complicated fast

5) How long can I-LOCK remain set?

Details of initiating a SF

a process may be fired up by:

1) getting an event

2) receiving an interrupt

3) being created.

For now, we consider only 1 (& will include 2 & 3 later if it falls out (or if we're forced to)).

A) The interrupt code calls the event code calls the scheduler. The scheduler detects that it's a SF awakening.

B) The scheduler may have to determine what was interrupted:

1) user

2) system

so as to get into (& out of) monitor mode correctly

7 Dec '70

Chapter XXimvldc of the
Continuing Interrupt Hassle

We're all aware of rumblings of discontent withthe current interrupt structure and several people have seen fit to propse sweeping alterations in ~~the~~ said structure. ~~currently implemented.~~ I think of the interrupt structure as fulfilling two widely different ~~xxxxxxfunctionxx~~ categories of tasks. First, tasks without which we cannot write the system as currently envisioned. Second, ~~xx~~ if a useful, coherent structure can be evolved to handle system necessities, it would be nice to make it available f or non-essential tasks. A poll of available staff gives the following list, to which additiions are ardently solicited:

A) SYSTEM NECESSITIES
   1) Major/minor panic from TTY
   2) Initiation of forced swapout
   3) Accounting interrupts (eg, too much ecs time*space)
   4) Forced logout for system shutdown
   5) Timer interrupts

B) USER TOYS

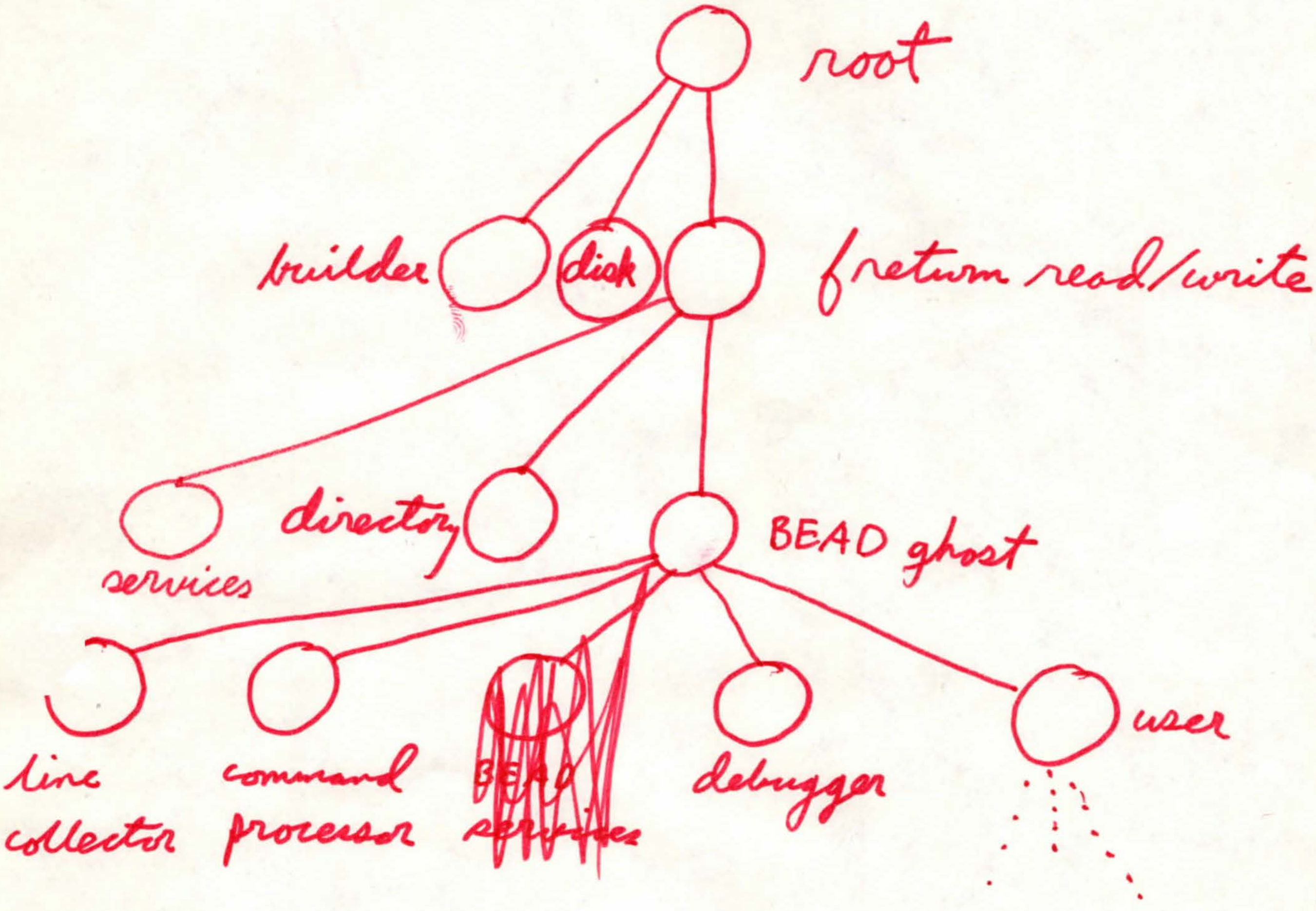
Arguments that a particular item deesn't belong in list A will in general~~xx~~ only be heard if the arguments are given in a quiet tone of voice (or in writing), and if they are accompanied by a fairly detailed method of implementing the featurein so me other fashion. (Item A1 is a necessity in the sense t hat wi thout it, the system would be hideous.)

The only one of these which has been tackled in detail, to my knowledge, is A1 with which Howard has been valiantl~~x~~y struggling for the last few weeks. H~~ex~~e claims that the objectives he has specified for cleaning up the call stack and getting to a debugger and suchlike other things cannot be implemented with the current logic. And he has an extensive proposal for a redesign. There is at le ast oneother (partial) proposal in the air, namel~~x~~y , Bruce's "linear interrupt priority" scheme.

I wo uld very much like to avoid the situation where a new implementation of interrupts is coded for Howard which turns out to fail to handle the other cases, so I want to provoke at le ast minimal discussion of the other system necessities before ~~xxxfinalize the design~~ coding of the second-generation interrupt~~s~~ system is begun.

Projected Process Profile
Disk

7 Dec '70



Notes

1) Bead services may be moved to son of fretum R/W to avoid being interrupted

7 Dec '70

Random Ramblings

1 TTY interrupts are aimed at the BEAD ghost
BG runs with interrupts continuously inhibited
- a panics won't take while BG is running until it ~~takes itself off the~~ ceases being the top of the stack
- b if more than 1 interrupt arrives while BG is on the top of stack, all but the first will be lost

3 Some system routines are protected from panic interrupts by the priority scheme; since no current proposal extends protection from caller to callee, the disk's potential acces to the line collector, et al, is gravely complicated.

4 System routines below the BG run with interrupt always armed; they protect themselves occassionally by setting the inhibit bit

5 Loops in the directory system can lock out ~~interrupts~~ panics for arbitrarily long periods of time (roughly controllable by a parameter specified how and by whom?)

6 The time ~~delay~~ required by the disk is unknown?, so delays in panics due to the disk are an unknown quantity

7 The forced swapout interrupt must be aimed at the f-return r/w node, or above*

8 Accounting interrupts and system shutdown can probably be aimed at the bead ghost**

9 Howard's algorithms depend heavily on the tree-scan feature of processing call-with-interrupt type interrupts; it would be nice if external interrupt processing were consistent, but it's out of joint with Bruce's linear scheme.

10 It isimpossible to imitate the tree structure priority scheme with the linear scheme.

11 If more than 1 interrupt arrives at a given node, only the first is guaranteed to take. Consequently, you aim different kinds of interrupts at one node only at peril of losing interrupts. (So 8 won't work.)

* Howard objects

** VV objects, see 11

7 Dec '70

ECS OPERATION TIMING

Operation	System time, us_8
DISPLAY USER CLOCKS	273
WRITE 1 WORD, shape (2,2,2,1)	465
WRITE 8 WORDS, "	1313
WRITE 1 WORD, shape (8)	411
WRITE 8 WORDS, "	414

Tune in next week for MAP FLAPS and SUBPROCESS CALL/RETURN !!

2 April '70

INTERRUPTS

A) GENERal considerations, or, the conflict.

1) Three categories of interrupts are envisioned. In order of decreasing urgency, they are:

   a) The system wants to do something to the process because

      i) he's used all his money

      ii) people have to be swapped out to unjam ECS.

      These interrupts must be honored in a hurry.

   b) The user sends an interrupt from his TTY. (Two levels of urgency currently exist, CSP and BREAK.) The faster these interrupts take effect, the better the system looks.

   c) The user has given a friend a capability to interrupt him, but only wants it to strike under certain conditions. It doesn't matter if this type of interrupt never strikes.

2) There exist manipulations which cannot be terminated gracefully in mid-stream. Here the difference between interrupted and terminated ~~xakexxitxxelf~~ should be mentioned. Most things can be interrupted, provided that they are later allowed to finish without having been disturbed in any way in the interim. Examples of things which shouldn't be terminated at arbitrary times are

   a) the DISK SYSTEM, when it's twiddling pointers

   b) the DISK SYSTEM, when it's in an ungainly posture with respect to having something half-way swapped in or out

   c) the LINE COLLECTOR, when it's twiddling pointers

   d) a DATA BASE UPDATER, when it's updating

   The last one poses serious problems, because it is a manipulation which must be invokable by the user.

3) The orderly termination requirement conflicts wth the semi-instantaneous response requirement. Any solution involves a compromise in that some small time interval must eventually be allowed for graceful completion of manipulations requiring graceful completion. Two radically different styles of solution have been discussed:

   a) Some sort of GLOBAL INTERRUPT INHIBIT BIT, or GIIB, which can be set locally to guarantee completion of critical operations. A bug which leaves this bit set indefinitely is intolerable, so that some mechanism of limiting the length of time that it is set must exist in the system. An implementation of this method is discussed in C below. Note that this scheme implies that allinterrupts are subject to some minimum delay whenever any critical manipulation is in progress.

   b) Making use of the current interrupt machinery, interrupts which must be honored fast are directed to an appropriately prestigious node of the subprocess tree (such as the root). The interrupt occurs immediately and then the SP fielding the interrupt has to decide what the hell to do with it. The bookeeping and implementation seem to be a nightmare, but this method is mentioned because it makes it possible to decide to interrupt something and then let it terminate later, giving potentially greater fesxibility and faster response than method a. A ghost of a suggestion as to implementation is given in D.

B) Current interrupt structure.

1) Subprocesses are arranged in a tree. Nodes above a given node are called its ancestors. A node is an ancestor of itself. Interrupts are directed to a particular subprocess, called the interrupt subprocess. An interrupt subprocess doesn't actually start execution until it becomes an ancestor of the subprocess currently executing, called the current subprocess. See fig 1.

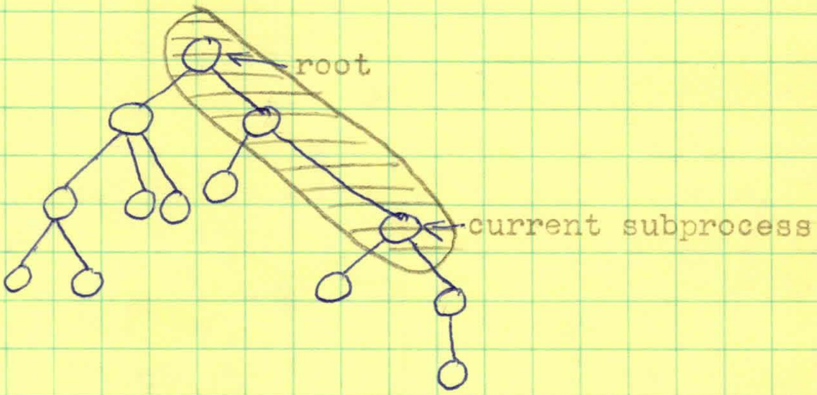


fig 1 - subprocess tree. interrupts directed to subprocess in the shaded area strike right away, modulo the IIB explained in 2; other interrupts wait.

2) Local interrupt inhibit bit (IIB). When an interrupt subprocess is fired up, an IIB is ~~set~~ automatically set which prevents the subprocess from recieving any further interrupts. The IIB goes away if the subprocess returns and doesn't have any effect if the subprocess has called another subprocess which is executing. The IIB may be set and reset by explicit system calls.from within the subprocess itself.

3) Interrupts arriving for ~~an interrupt subprocess~~ a pending interrupt subprocess are lost and have no effect; the first interrupt to arrive for a subprocess with the IIB set is remembered, subsequent ones are lost.

4) Howard justly observes that the tree structure for subprocesses serves a second function, namely, it determines how many nodes coexist in ~~storage~~ CM. An undesireable effect of this second use of the tree is that a subprocess which is logically an ancestor of some other sp may be put "off to the side", so as not to cramp it's (logical) descendents core. To salvage the interrupt logic, the ancestor must be split into a small piece, to intercept interrupts, and a main piece off to the side which is called by the small piece. This results in a proliferation of subprocesses.

C) Butler Lampson's (BCC's?) global interrupt inhibit with timer solution.

1) Basically, there is a GIIB which may be set and cleared by system calls. Associated with the GIIB is a real-time timer which is set to LIM whenever the GIIB is set. If the timer runs out while the GIIB is still set, error processing is initiated.

2) BCC allows a subprocess to set the GIIB even when it has already been set by a calling sp. So,

   a) the actual time that interrupts are locked out may by LIM*(depth of call stack), roughly.

   b) the GIIB has to be manipulated in the call stack, or some other stack.

3) The scheme makes it necessary for the system periodically to touch every process in the system (or every process on a list of processes with the GIIB set) to update the timer. When ~~the process is fired up~~ a timer runs out, the offending process must be fired up and error processing initiated. God knows what becomes of ~~pending~~ any interrupts pending on the process. Also, error processing has to be revamped to prevent undesireables from intercepting the error.

4) A big objection to the implementation of this scheme is the stack of timers - is it really necessary?

D) The magic, all-knowing subprocess solution. I can't get excited about really implementing this scheme, but a rough idea follows. It is a theoretically interesting solution, asit allows the possibility of "suspending" critical manipulations for later completion and avoids locking things up absolutely every time a manipulation deemed critical is being performed. (Consider the ~~d~~ case where the system has decided to destroy the process absolutely; it no longer seems too important to allow the LINE COLLECTOR to terminate gracefully.)

1) Interrupts are handled as at present. Important interrupts are directed to sufficiently prestigious node of the tree.
2) The prestigious subprocess (PSP) is responsible for any idiosyncracies of his dependents. He decides what to do.
    - a) If PSP decides to process the interrupt right away, there are no problems (unless, of course, he's wrong).
    - b) If the PSP decides something critical might have to be wrapped up before processing the interrupt, he sets some kind of real-time timer and does a special call of the critical sp, warning it to tidy up. If the critical sp returns in time, fine. If not, we're in the same bag as when C's timer runs out.

E) Conclusions. As of this writing, we are short on conclusions. Everyone seems resigned to implementing some sort of GIIB with some sort of timer, but various people are still trying to conjure solutions simpler than C.

Also under discussion is the possible organization of the subprocess tree for the "typical user", vis-a-vis handling of various categories of interrupts. Nothing worth writing down has emerged from these discussions as yet. (People are still proposing radical alterations of the current interrupt structure. Boo-hiss.)

details of C
transcribed by Jim

1 Apr 70 Dick meeting

Each call stack entry has call
which is timer on nointerruptibility
which is expiration date on this
process — starts = 0

If ~~set~~process sets it set to
min(val, 2 min)
If clear then reset to 0
If return it losses
elapse = error
call ⇒ copy timer to new guy
set it once inherited once thereafter must be set from 0.

If send interrupt then
If clear takes
If time then saved and
when reset it strikes
if returns it strikes

(wants to prevent termination not interruption)

Details of C (cont)

processes chained

every 2 minutes chain is searched

30 March '70
1

EVALUATION OF WORK YET TO BE DONE ON THE ECS SYSTEM AS OF 30 MARCH 70

STUFF NEEDED FOR THE OPERATION OF THE DISK SYSTEM

1) Allocation mod 64 for DAE map entrues (Vance)
2) Compactifier (Vance)
? 3) Change to movex block operation (Paul)
    a) Check map reference count
    b) Return dirty bit in X6
cancelled 4) Change probe operation to return # of map refs in X7 (Paul)
later 5) New operation to turn off/on map entries for a subprocess (?)
6) Implementation of two new parameter types, block parameters and return parameters (Bruce)
7) Indirect C-list (Bruce)
8)
9) Return capability of specified type

Stuff IMPORTANT TO THE OPERATION OF THE ECS SYSTEM

1) Find descendent of subprocess (Dave)
2) Change map compiler to do F-return in case of missing map block instead of DISASTER (Paul & Bruce)
   error
3) Change to change unique name operation vis-a-vis option bits   temporarily, do without change U.N.
4) Get more system code out of central and into ECS (Vance)

STUFF WHICH WILL BE NICE WHEN IT GETS DONE, IF IT EVER DOES

1) Set temporary part of class code
2) Put check in PUTACT and PUTECS for length of ACTIONL
3) Implement accounting of CPU time
4) Reset end of path to self
5) Get the option bits into the operations (Vance)
6) Fast actions
7) Implement the error return operation
8) Fix up CCCLOA (what does this mean?)
9) General destroy operation
10) Send interrupt to pseudo-process (Howard, is this still needed?)
11) Move ~~Class~~ CLASSCNT to ECS
12) Check ~~Garb~~ GARBCNT in subprocess environment establishment at the point of doing the direct access map entry
done 13) Fix up the O-level file name hassle (Paul)
14) Fix error returns from OPINTR
15) F-return when subprocess to be deleted is not a leaf (Bruce)
16) In process and subprocess creation, correct test of lower limit for entry point
17) Design and implement display process descriptor operation
18) Incremental map compilation (Paul)

STUFF ON WHICH THERE WAS NO IMMEDIATE CONSENSUS

1) Provide date and real time
2) Move from one allocation block to another
3) Move an allocation block to another allocation block
4) What about ~~data~~ message channels?

Disagreements as to the above classifications will be cheerfully discussed. ~~And perhaps changed.~~ People indicated as being somehow responsible for performing changes may try to wriggle out of it (volunteers for ~~new projects~~ unassigned projects will be courteously received).

I left the meeting without and understanding of how the file block dirty bit was to perform ~~its function~~ its function. It was supposed to be maintained by the ECS system and somehow save the disk system the trouble of writing out blocks from read-write files unless they had actually been altered. Exactly what is the proposal?

ALLOCATION BLOCKS

Much thinking has been going into allocation blocks, ECS space accounting, and CPU time accounting. Here is a semi-solid proposal.

1) CPU time should be taken out of allocation blocks and put , probably, into the process descriptor. Several reasons
   a) AB's are really to control ECS usage and the current CPU time stuff is just a hopeful, incompletely evaluated after-thought
   b) If a process is allowed to run at different weights, the time has to be accumulated separately for the different weights (and you don't want to keep a weight in the AB)
2) CPU time should be counted down. When a process ~~[illegible]~~ is swapped in, if it has no time in the slot currently being charged, an error is generated and either
   a) if there's more than one pool of CPU time, control is switched to another pool to cover the processing. If the last pool runs out, it's an error error or the equivalent, and the process is shut down, perhaps destroyed.
   b) if there's only one pool of CPU time, the process is loaned epsilon time by the swapper and marked bankrupt. If it's already bankrupt, error error. The system operation which puts money in the CPU time pool clears the bankrupt signal. ~~(and probably accounts for the loan if it's really~~

   In both methods, it is anticipated that the initial error will be intercepted by somebody competent to straighten things out, like a very priveleged system accounting subprocess. If the user intercepts the error inone of his own subprocesses and blows it, he gets had.
3) ECS space accounting in AB's is to be changed to charge for the amount of ECS that the AB has tied up, not the amount that it happens to be using. The latest model allocation block will contain 3 space parameters, 2 time-~~[illegible]~~space integrals, and the invisible time of last bill field. (See fig. 793-42B.03f)
   a) UPPER BOUND - can be set arbitrarily and doesn't reflect any real memory anywhere. It is used to control somebody you don't trust.
   b) CHARGED SPACE - this is available to the AB on demand and is the amount charged for. Space ~~[illegible]~~ added to this field comes from its father AB and increases may fail for lack of space in the father or for exceeding the local limit.
   c) SPACE IN USE - space currently in use, may not exceed charged space or an error is generated.

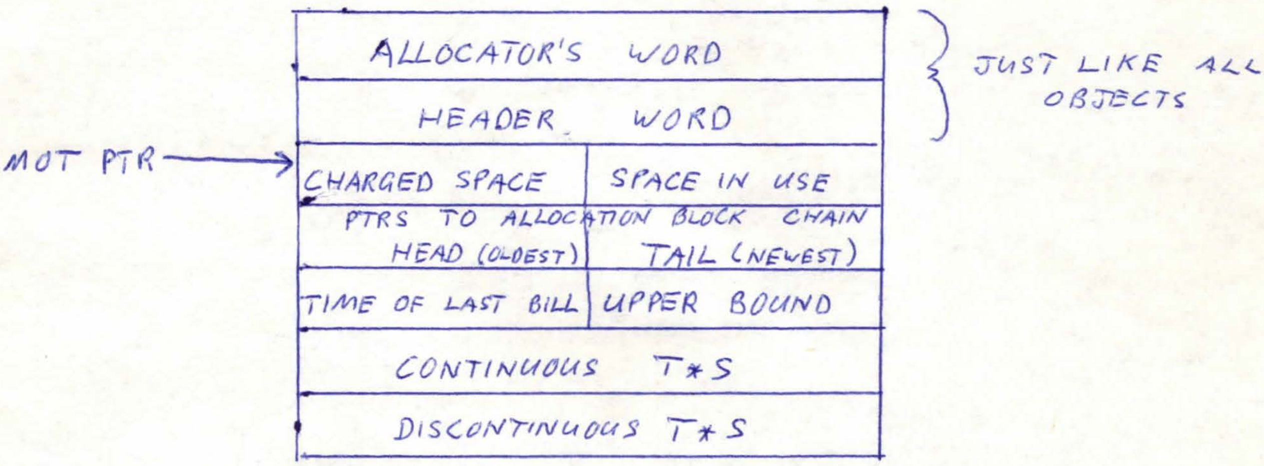
   Nice guys and poor guys will try to keep charged space dwon around space in use; rich guys may keep a lot of charged space in case they might need it. ~~Increases~~ Meaningful increases to charged space will presumably entail a call on a priveleged system routine to get space from a system pool, and delays may result 'cause space isn't available.
   d) CONTINUOUS T*S - starts at 0 when the AB is created and builds up continuously. Facility to display it will be provided.
   e) DISCONTINUOUS T*S - I don't like this field for reasons explained below. When it is displayed, it is reset to 0.

   A DAEMON process runs periodically and touched the AB's, to prevent deficit spending. Bruce wants to use the discontinuous field and charge the guy right away, so that if the system crashes, he stands charged for some T*S, which he

13 april '70

may or may not have derived any benefit from. The guy will almost certainly complain bitterly. I think that the continuous field should be used by the DAEMON to check against deficit spending, but that the DAEMON should do nothing in the normal case, leaving the log-off procedure to do all the actual charging.

FIGURE 793-42B.03f



ALLOCATOR'S WORD	
HEADER WORD	
CHARGED SPACE	SPACE IN USE
PTRS TO ALLOCATION BLOCK CHAIN HEAD (OLDEST)	TAIL (NEWEST)
TIME OF LAST BILL	UPPER BOUND
CONTINUOUS T*S	
DISCONTINUOUS T*S	

If the time of last bill is kept in units of micro-seconds/1024, 30 bits allows about 16 days of running. If this is deemed insufficient, speak now. More bits may be used or the units can be changed.

13 April '70
2

ALLOCATION BLOCK OPERATIONS

A) Create allocation block (no change)
IP1 C: father AB (OB.CREAB)
IP2 D: C-list index for returned capability

B) Transfer charged space
IP1 C: Donor AB (OB.GIVE)
IP2 C: Donee AB (OB.GET)
IP3 D: Space to be transfered, or donation
fails if CHARGED SPACE+DONATION exceeds UPPER BOUND in donee
or DONATION exceeds CHARGED SPACE*SPACE IN USE in donor

C) Set upper bound
IP1 C: AB (new option bit)
IP2 D: new upper bound
fails (or F-returns) if new upper bound less than charged space

D) Read discontinuous T*S
IP1 C: AB (new option bit')
IP2 D: where T*S is returned (or return it in X6?)
resets discontinuous T*S to 0 and returns updated value

E) Display AB
IP1 C: AB
IP2 D: buffer
updates both versions of T*S, doesm't reset discontinuous T*S

F) Return capability for nth object on the AB (no change)
IP1 C: AB (OB.GOD)
IP2 D: full C-list index for returned capability
IP3 D: number of desired object (n)

G) Destroy AB (no change)
IP1 C: AB (OB.DSTRY)

13 April '70
3

DIRTY BITS

In order to save the disk system some unnecessary writes, it was decided to provide a dirty bit on file blocks which would enable the disk system to tell whether or not a block had to be copied back out to the disk. The final specs were:

1) File blocks are created clean
2) Blocks are dirtied by
    a) File writes, including ones with words counts of 0
    b) Being put in a map RW
    c) Being put in a direct-access map entry
3) An operation to test and reset the dirty bit will be provided.
~~4) File writes with 0 word counts set the dirty bit~~
4) Move block carries the dirty bit along with the block.

With this machinery, it is claimed that blocks from a file opened RW will not have to be written out to the disk if

1) Somebody just scans through the file and doesn't actually write in some of the blocks
2) A block hasn't been written in since it was last written out, due to a pseudo-close or somesuch mechanism

just for some concrete examples.

13 April '70
4

DELIVERY OF INTERRUPT DATUM

It is proposed to alter the location where the interrupt datum is delivered from IP0 (cell 6 of the subprocess) to cell 2 of the subprocess. Current delivery clotbbers the first input parameter. Any objections?

CHANGE TO CHANGE UNIQUE NAME OPERATION

It is proposed that CUN be altered to have 2 parameters:

~~IP1 E: C-list index of object (OB.CHNAM)~~
~~IP1 D:~~
IP1 C: capability for object (OB.CHNAM)
IP2 D: C-list index for return of new capability.

This is a funny thing from the point of view of the user, since the old capability becomes no good after the operation, but it allows the system to do its option bit testing in the usual place instead of in the CUN code.

CHANGE UNIQUE NAME AND MISSING MAP BLOCKS

A block refered to in a map may be caused to disappear by the use of the change unique name operation. The question is, what should the map machinery do when it encounters a map entry with miss ing blocks? The only answer seems to be that the offending map entry should be zeroed and error processing should be initiated. This is unpleasant, as the error is go ing to be discovered in the swapper, but it seems like there is no alternative. How about it?

ALLOCATION

Work on the allocator (initially undertaken to write a compactifier) has revealed certain problems:

1) The documentation is scanty and not overly helpful. For example, the purpose for the two 0-length free blocks isn't mentioned, how compactification is to be (incrementally) achieved is left as an exercise, etc.

2) There are bugs
   a) Free blocks are merged without due regard for limitations on their size
   b) Interrupt objects are scattered through core in such a way as to make keeping them fixed during compactification a somewhat bewildering problem
   c) There are miscellaneous quirks in the initialization.

3) Objects are limited to 2**17 - 1. This limits DAE's to
   2**17 - epsilon for 0-level files
   2**16 for other level files

4) The top and bottom of ECS are both fixed by various factors. This makes it difficult to dynamically change the size of ECS.

It's easy enough to fix the bugs and improve the documentation. And the top of ECS can be freed by various ploys which can be simple and inefficient or medium difficult and as efficient as at present. The stopper is item 3. Extensive rewriting will give a factor of 4; extensive rewriting plus an additional word or redsign of the allocation chain are necessary to completely unrestrict object sizes.

It is roughly true that the redesign and changes necessary to deal with 3 and 4 are internal to the allocator and can be redone later without affecting other code (the main possible exception is the file code, which shares one of the allocator's words). I feel that it is smewhat a matter of style as to whether we fix these things now or later, but I would like to have some commitment on item 3 right away.

INCREMENTAL COMPACTING

It is deemed desireable that the compactifier should be designed in such a way that some process may run while compactification is in progress. Namely, a speed freak shouldn't have to wait for compactification to complete before running. There seem to be two different schemes which allow suspension of compacting in mid-stream;

1) To tell the compactifier in advance only to do so much and then to check for speed freaks when it returns. You could tell it to collect n objects ofr example. But you have to understand that it may get into something big that it has to finish.

2) To have a flag which the compactifier looks at which tells it to stop as soon as possible. I prefer this, as it is more efficient. There is still a limit to how fast the compactifier can react, but ~~it~~ control is better than with 1.

20 April '70

ADDITION TO DIRTY BIT SPECS

Howard points out that it should be mentioned that move block carries the dirty bit along with the block.

KARL'S OBJECTION

With respect to the blurb on allocation last time, Karl says that the objects in ECS are set up so that one object can be moved ~~and~~ without necessitating the relocation and/or massaging of some large fraction of ECS and that's all there is to incremental compacting; he was annoyed that I said incremental compacting was left as an exercise, since it is obvious, trivial, etc., how to do it.

Karl is right about this. What I was objecting to was a lack of any sort of general description of the problems that were supposed to be handled by incremental compacting, like speed freaks, interrupt code, compacting during the idle loop, etc. This lack is no doubt not ascribable to Karl as it seems to be a more or less general quality of the documentation. I'm putting together a little blurb on the compactor which may be out next week.

Meanwhile, if Karl isn't mollified by this, he can submit his own disclaimer for next week. OK?

ALLOCATION DECISIONS

Last week, we decided to do whatever was necessary to

1) fix the existing bugs
2) free one end of ECS
3) provide an incremental ~~xxxpxxxxfixwithx~~ compactor which will:
    a) massage some number of real cells or collect some number of free cells or both (see compacting document)
    b) do the normal I.WAIT/I.LOCK logic to allow interrupts
    c) monitor a new cell (I.COOL?) which will cause the compactor to suspend compacting when the cell gets set and take a special exit (for to run speed freaks).

It was decided not to do the engineering necessary to make it possible to create arbitrary size objects (unless it somehow falls out).

BENT FILES

All talk about speedfreaks, incremental compacting, etc., is vacuous unless we get to the bottom of the bent file problem. I would like to hear exactly what the problem is and a decision as to what's to be done about it.

CPU TIME ACCOUNTING

I would like to have Jim Gray order Howard and I to figur out how CPU accounting is going to be done. Discussion of the properties and problems of CPU accounting is in order.

GLOBAL INTERRUPT INHIBIT

Solid proposals and decisions for implementing the GIIB are in order. Also, the configuration of the user's process tree vis-a-vis interrupt handling could use some clarification.

20 april '70
2

DIRECT ACCESS REVISITED

At the 17 April disk system meeting, the old headache of DAE's was discussed again. It was realized that we were into giant headaches and going around in various size circles, so we backed off and started all over again. What the hell are DAE's good for anyway? The only use that the participants could suggest and defend was to give the user access to a large,fast address space. (Proponents of other uses, please step forward at once.)

On the basis of the "large block" theory, the following decisions were tentatively made:

1) At the ECS level, blocks have to be created nudged, as opposed to created first and then nudged later. This avoids fairly unpleasant problems encountered when trying to find space to relocate a large ~~xx~~ block.
2) At the disk level, big 0-level files and only big 0-level files are always created nudged. This is fairly restrictive, but it is adequate to the only use so far proposed for DAE's. (A side kludge is that big directories will be implemented as multi-level files unless we want them nudged, but that seems OK.) "Big" remains to be defined exactly.

REALLOCATION

Currently, when an object is being reallocated, if it can't be expanded in place, it is briefly charged to it's father allocation block twice (while another place for it is being found and it is being relocated). This has at least two bad consequences:

1) It makes it slightly difficult for you to control accurately the space used by some untrustworthy subprocess.
2) For every user in the system, the disk system has to either give him space for 2 process descriptors or be very tricky about allocation/changes to the process descriptor.

The reason behind the double ~~allo~~ charge is to avoid locking up ECS 'cause of space problems. ~~It~~ Mechanisms for avoiding the double charge and not causing a disaster are under consideration. There seem to be only 2½ ~~x~~ kinds of ~~objects~~ things which get reallocated

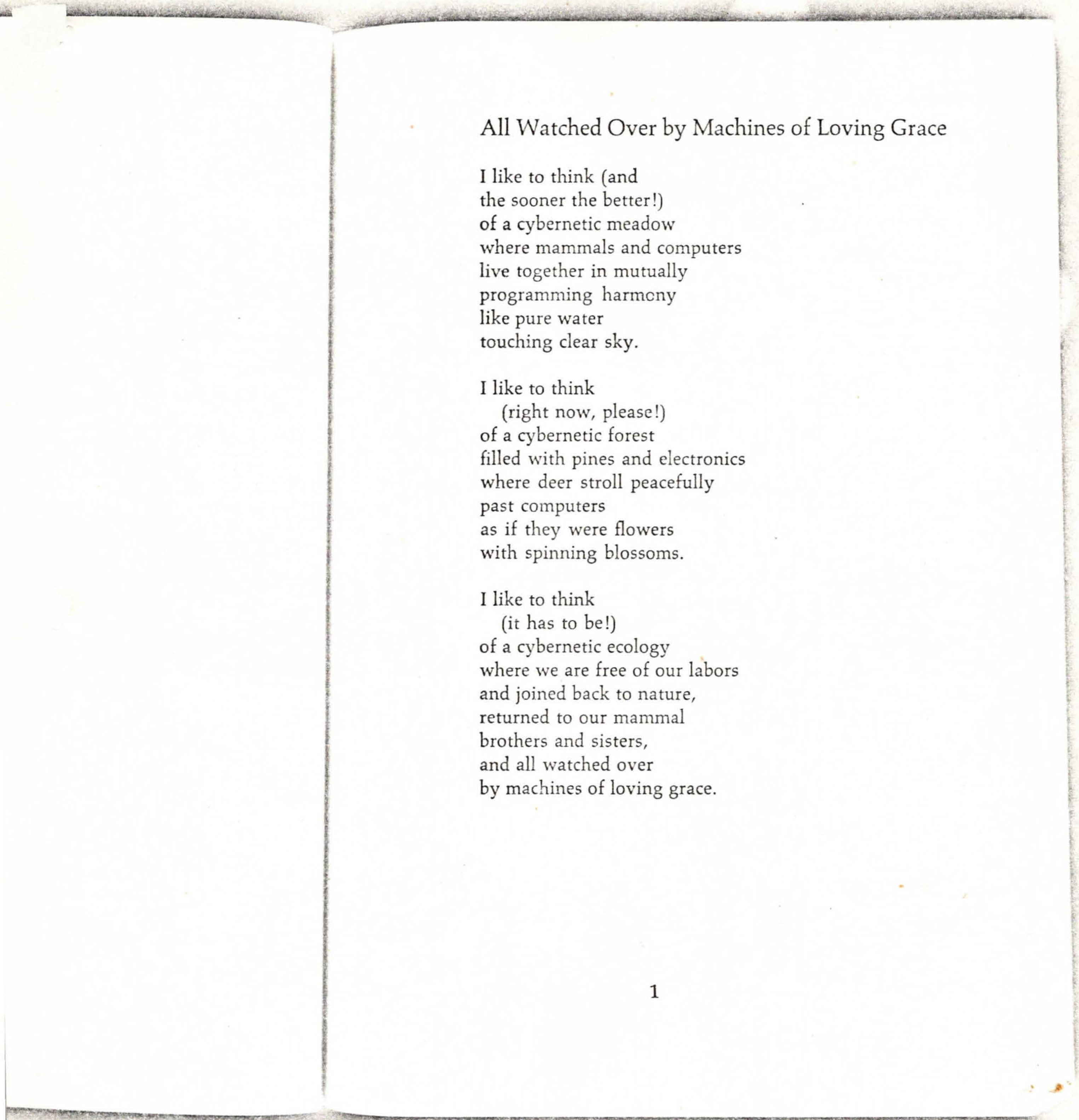
1) Process descriptors
2) Operations

2½) Maybe nudged blocks, pending the outcome of the DAE debate.

Since 1 & 2 are small, they could usually be handled by "hidihg" some number of cells of ECS and counting on these for the relocation of small objects; larger objects could be doubly charged as at present, or handled out of some kind of system space pool (with the possibility of failure, since there may not be enough system space).

Anyone have a nice solid idea?

27 April '70
1



All Watched Over by Machines of Loving Grace

I like to think (and
the sooner the better!)
of a cybernetic meadow
where mammals and computers
live together in mutually
programming harmony
like pure water
touching clear sky.

I like to think
   (right now, please!)
of a cybernetic forest
filled with pines and electronics
where deer stroll peacefully
past computers
as if they were flowers
with spinning blossoms.

I like to think
   (it has to be!)
of a cybernetic ecology
where we are free of our labors
and joined back to nature,
returned to our mammal
brothers and sisters,
and all watched over
by machines of loving grace.

Richard Brautigan

1 June '70

RECONSTITUTED LIST OF THINGS TO BE DONE ON THE ECS LEVEL OF CAL TSS

$tuff needed for operation ~~for~~ of the "Septmeber System"

1) Allocator-compactifier (Vance, endof June)
2) Implementation of block parameters and return parameters (Bruce, BNG*)
3) Indirect C-list stuff (Bruce, done and tested)
4) Set temporary part of class code (easy) - Bruce
5) Return capability of specified type
** 6) Change map compiler to do error instead of DISASTER in case of missing map block (Paul, ?)
7) Get option bits into the operations (Vance, easy)
8) Implement the error return operation (Bruce, BNG)
9) Date and real time (Keith, ?)
10) Find descendent of subprocess (Dave, written but not in)
11) Fix n'th parameter of an operation (Paul)
12) Dirty bit stuff (Paul, written but not debugged)

Stuff needed for the "Real System"

1) DAE entry stuff (just won't be available in Sept)
2) Operation to turn on/off map entries for a subprocess (no subprocess descriptors in the Sept directory system)
3) Change to change unique name oper vis-a-vis option bits (NA in Sept)
4) Move system code out to ECS (Vance, in progress)
5) Message channels

Other stuff

1) Make PUTACT and PUTECS check the lenght of ACTIONL
2) Accounting of CPU time
3) Operation to reset end of path to self
4) Fast actions
5) Fix to CCCLOA (this is easy and a lot of code will be affected by it, so I'll get it done)
6) General destroy operation
7) Send interrupt to pseudo-process (written, but not in)
8) M ove CLASSCNT to ECS
9) Check GARBCNT at point of doing DAE when establishing subprocess environment
10) Fix error returns from OPINTR
11) F-return when subprocess to be deleted isn't a leaf (Bruce, ?)
12) Display process descriptor and subprocess descriptor operations
13) In process and subprocess creation, test for lower limit of entry point correctly
14) Incremental map compliation (Paul, ?)
15) Move object (Allocation block) from one allocation block to another

*BNG = before national guard

** was this going to be postponed until NEWUN is released?

1 Feb '71

Questions on which you are invited to expressopinions in the next few days:

1) Should external interrupts do a tree search similar to internal interrupts? That is, if the target subprocess of an interrupt has interrupt disarmed, should the interrupt be sent to its nearest ancestor with interrupts armed?

2) Should error numbers (or input parameters) be relocated to avoid the current conflict on subprocess calls?

3) Should the order of parameters be inverted when doing a subprocess call from a high level of a multi-level operation? That is, if we are processing the third order of an operation, the parameters of order 3 would be put in subprocess core first, followed by the parameters of order 2 and order 1.

4) Should facility for ECS actions to return parameters be provided?

5) Should the number of error classes be increased (from 32?) or made variable?

6) Does anyone have any thoughts on stack full errors? (Like the disk folks.)

The 6,1,0 error that can occur during disk loading or recovery is caused by the disk not destroying an empty file before it tries to bring it in.

ARCHIVE PROPOSAL

On the first of each month, a system tape and a disk dump tape will be made and put in the vault. The three most recent dump tapes will be kept; only the most recent system tape will be kept.

ECS modification list

Appended is a list of ECS modifications from last year with the completed stuff indicated and the stuff that seems to be no longer important also indicated (but with a different mark!). Comments are appropriate.

✓ means done
I.P. " in progress
NCD " not completely designed
NS " not scheduled
½ " a partial solution/code is done
? " ?
U.S " now seems irrelevant

(2)

1 Feb '71

Map error logic

I. Separation of change unique name count from compaction count. It has been suggested that two counts be kept on the compiled maps. The compaction count acts the same way the current count does, i.e. if the count isn't as large as the current number of compactions, the map is recompiled.

The new count, which might be called the "map invalidation count", or somesuch descriptive name, would be compared against a count which is maintianed by the system and incremented by one each time the unique name of a file which has a block in a map is changed. Whenever the map code finds the local count on a map to be behind the global count, it checks the logical map to see if all the files still exist. If so, it resets the local count to the global count. If not, it recompiles the map and flags the subprocess for a map error.

II. Handling of the map error. Because swapout of a subprocess may occur asynchronously with respect to execution of the subprocess, it is deemed unsuitable to signal the error to the process at the time that it is discovered. Rather, the error is remembered and not signaled until the subprocess in question becomes part of the full map again. (This means right away if it is becoming part of the full path when the error is detected of course.)

Appendix

1 June '70 (3)

RECONSTITUTED LIST OF THINGS TO BE DONE ON THE ECS
LEVEL OF CAL TSS

Stuff needed for operation ~~for~~ of the "Septmeber System"

✓1) Allocator-compactifier (Vance, endof June)
✓2) Implementati n of block parameters and return parameters (Bruce, BNG*
✓3) Indirect C-list stuff (Bruce, done and tested)
✓4) Set temporary part of class code (easy) - Bruce
✓5) Return capability of specified type
I.P. **6) Change map compiler to do error instead of DISASTER in case of missing map block (Paul, ?)
✓7) Get option bits into the operations (Vance, easy)
✓8) Implement the error return operation (Bruce, BNG)
✓9) Date and real time (Keith, ?)
✓10) Find descendent of subprocess (Dave, written but not in)
✓11) Fix n'th parameter of an operation (Paul)
✓12) Dirty bit stuff (Paul, written but not debugged)

Stuff needed for the "Real System"

NCD 1) DAE entry stuff (just won't be available in Sept)
NS 2) Operation to turn on/off map entries for a subprocess (no subprocess descriptors in the Sept directory system)
✓3) Change to change unique name oper vis-a-vis option bits (NA in Sept)
IP 4) Move system code out to ECS (Vance, in progress)
NS 5) Message channels

Other stuff

½ 1) Make PUTACT and PUTECS check the lenght of ACTIONL
IP 2) Accounting of CPU time
?, NS 3) Operation to reset end of path to self
NS 4) Fast actions
✓ 5) Fix to CCCLOA (this is easy and a lot of code will be affected by it, so I'll get it done)
NS 6) General destroy operation
? 7) Send interrupt to pseudo-process (written, but not in)
✓ 8) M ove CLASSCNT to ECS
IP 9) Check GARBCNT at point of doing DAE when establishing subprocess environment
½ 10) Fix error returns from OPINTR
US 11) F-return when subprocess to be deleted isn't a leaf (Bruce, ?)
½ 12) Display process descriptor and subprocess descriptor operations
IP 13) In process and subprocess creation, test for lower limit of entry point correctly
NS 14) Incremental map compliation (Paul, ?)
NS 15) Move object (Allocation block) from one allocation block to another

*BNG = before national guard

** was this going to be postponed until NEWUN is released?

22 Feb '71

NEW STACK LOGIC

The manipulation of the call stack is being extensively revised. The most significant changes are:

1) When a subprocess does a system call, the p-counter in the stack will point at ~~to~~ the XJ, not one beyond it;

2) A p-counter qualifier will indicate whether the subprocess was

   a) about to execute the inst at p-counter

   b) in the middle of the inst at p-counter (presumably an XJ)

   c) almost finished with the inst at p-counter "

3) A return ~~instruction~~ action which will modify the p-counter qualifier of the ~~stack entry returned to~~ previous stack entry as part of the action will be provided.

4) The interrupt inhibit bit ~~in the top o~~ will always be set when a new top of stack is formed, so that the current, running subprocess will automatically have interrupts inhibited. An operation to explicit~~e~~ly set* and clear the bit will also be provided.

5) The forced f-return and interrupt flags will disappear. I would like to move the interrupt inhibit bit from its present position if no one objects.

A complete description of the display stack operation and an "internals" specification for the new stack should be available soon.

Is there any enthusiasm for an action to display stack entries from some other process?

*Oh my God, a split infinitive!

THE LATEST WORD ON THE EXCITING STRUGGLE TO OVERCOME THE ELUSIVE BLOCK*GONE*FROM*FILE*IN*MAP/CHANGE*UNIQUE*NAME*OF*FILE PROBLEM (CONSIDERED IN CONJUNCTION WITH TURNING MAPS ON AND OFF)

Maps now have two counts on the compiled part and a new flag on the logical part

1) a local BADMAP count

2) a local COMPACTION count

3) a map on/off flag.

There is in addition a new flag on subprocesses, but it is very elusive. It says ~~whether~~ or not the subprocess is suffering from a pending map error. whether

To begin with the map on/off flag

1) An action to turn off the map of a specified subprocess will be produced in due course. It will decrement the map count on all ~~blocks~~ file blocks used by the map and set the bit to off. (One gets an error for trying to turn off the map of a subprocess currently in the full path. Is that OK with everybody?)

2) ~~trying to~~ doing anything that might cause a subprocess with its map turned off to swap in will cause an~~d~~ error, as discussed below.

3) The operation to turn the map back on will be fraught with all so rts of hazards stemming from missing blocks and files, but if one is lucky, it will find everything present that is necessary and increment the map count on all the relevant file blocks and turn the bit off.

When one changes unique names on a file, if the file has a block in a map, the map count on the block is cleard and a global BADMAP count is incremented. This leaves some map, somewhere, sitting around with one of its files ripped off. This may later lead to an error as discussed below.

I regret that I must also mention that some careless code may callously destroy a c-list that is the local c-list of some innocent subprocess, thereby causing said innocent subprocess grave embarrasment. (When the current swapper tries to bring in such a subprocess, it destroys the process!) But have no fear, relief is at hand.

How is one to see one's way out of this quagmire? Well, let's start with the hard-working swapping code, MAPOUT/MAPIN, which do the bulk of the systems swapping work. Here comes this subprocess to be swapped out/in. If the two local counts on the compiled map are up-to-date, and the map is on, the swap proceeds. But, if a count is off, further action is taken

0) if the map is off, an error is signaled to the caller (no swapping or compiling)
1) if the COMPACTION count is off, the map is recompiled

2) if the COMPACTION count is OK, but the BADMAP count is off, a check is made to see if all files in the logical map are still present; if so, the count is updated and the swap proceeds, but if not, the map is recompiled.

Whenever the map compiler encounters a missing file in a logical map, it zeros the map entry and proceeds with the compilation. It later exits with a signal if a file was gone. MAPOUT/MAPIN return this signal to whomever called them. The map is then swapped (with a possibly newly zeroed entry).

Now, if we're swapping, we're either swapping in or out, if you see what I mean. So, suppose we're swapping out and we get one of these funny errors from ~~MAPXXX~~ MAPOUT, what the hell do we do with it?
Remember that the subprocess we just swapped out may not even be part of the full path currently, for reasons that are classified (the president knows best though, you may be sure). Well, we

1) ~~mapofxisxaxDISASTER~~ ignore a mapoff error. Because you can only turn off the map of a subprocess that is out. This means that the mapoff condition was detected on swapin and the appropriate error generated as describled later.

2) If a file was gone, since the entry was zeroed, the subprocess will swap back in later with nary a whimper, so we flag the subprocess at this point for a pending map error.

And that about covers swapout.

But what about swapping in? Here we can encounter three different hassles while ~~ixxx~~ just doing our job and minding our own buisness.

1) If the local c-list of the currnet subprocess has been ripped off, we generate the appropriate error right away.

2) If the logical map of one of the subprocesses in the full path is turned off, we generate the appropriate error.

3) If MAPIN reports that the logical map of one of the subprocesses in the full path has had a file ripped off, we also generate an error. [redacted]

4) Last, if we see that one of the subprocesses that we are bringing in has a pending map error condition, we again generate the file-ripped-off error. <u>The flag is turned off.</u>

But what if they all happen at once? Only one gets reported, namely

1) the c-list error if it occurred

2) failing that, the type of map error occurring on the subprocess

closest to the current running subprocess is reported.

Well, I sure am glad to have that off my mind. Oh yes, I forgot to mention that when the map compiler encounters a missing block when it's compiling a logical map entry, it is still a DISASTER.

CONTROL OF CPU TIME

Two new features are being implemented for processes, a timer and an associated message mechanism. Time may be moved between the CPU time field of an allocation block and the timer of any of its owned proceses. When a process is swapped out, its timer is decremented by the ~~total~~ time it just used, and if the result is negative, the process is descheduled; the negative residual sits in the timer.

The message mechanism, which is set by a separate system action ~~efxit~~, consists of an event channel and an event. When a process is descheduled, if the message mechanism is set, the swapper sends the event on the event channel (any errors, such as event channel gone or full, are ignored). If the message mechanism is not present, nothing further is done.

The operation which moves time into a process timer increments the current timer. If the ~~timer~~ process is descheduled and the timer goes positive, the process is rescheduled.

I heavily favor creating processes descheduled, but it is not too late to argue for an additional parameter on process creation to initialize the timer (that is the only alternative that I can see). A graceful phase-in will beprovided in any case.