

Sept 2
~~May 27~~, 1969

System entry/exit

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, to collect and check the parameters needed for the action, to transfer control to the proper system action routine, and to handle the return to the user after the system action is completed.

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 1). 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 IPO) is interpreted as a C-list index and 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 see that it is a capability for an operation; if it is not, an error is generated. The parameter specifications of the operation are interpreted by OPINTER and an actual parameter list is formed in the process descriptor starting at P.PARAM.

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, data parameters or C-list indices. User supplied capabilities are checked for the correct type and required options unless the parameter specification is "any capability". All capability indices are checked to be sure they fall in the range of the full C-list. If an "any" 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 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, and the bit mask(s) are copied from the operation into the cells preceding P.PARAMC-1.

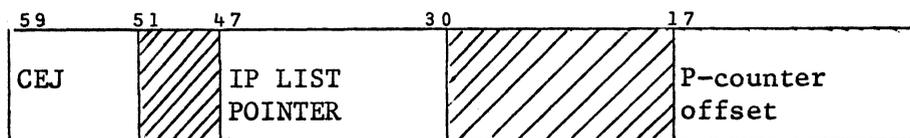
Finally, the ECS action number is extracted from the operation; it is used as an index to jump into the ECS action jump table starting at ACTIONL where there will be a jump to the proper entry point for the desired ECS action.

Upon 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 exception to this is the case in which the user process has blocked on an event channel, in which case the event channel routine exits to the swapper.

There are three points to which ECS action routines may return. The normal return is to 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. The system time clocks at S.SYSTM in system core and P.SYSTM in the process data area are updated, and a check is made 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 is executed to return control to the user.

The second return is to TOUSER and is 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 and the subprocess return routines. The third return is at S.RETU and simply does the CEJ to the user. It is used by the swapper to transfer control to the user.

Figure 1
SYSTEM CALL



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.

Why no unique name provided

The Master Object Table (MOT) is located in low ECS and contains an entry for each object in ECS. Each entry occupies one cell and contains a pointer to the object as well as a "unique name" associated with the object. Except for the special case of "direct access" 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. Furthermore, the MOT facilitates object relocation.

The unused entries in the MOT compose 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. System disasters occur when the MOT free 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 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 are data blocks and the other nodes are pointer blocks, 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 continuous 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.

ALLOCATION BLOCKS

The allocation block is the object which relates ECS allocation to funding. 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 object is associated with an allocation block; these objects are linked in a two-way circular list. The allocation block has pointers to this list, and each object has a backpointer to its father allocation block. The objects of ECS, therefore, form a tree, all but the leaf nodes of which are allocation blocks. 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 daughter processes, and will be charged rent on the ECS space occupied by its daughter objects. FUND is the routine which charges this rent and must be called whenever the size of a daughter 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).

As of this writing for rent on ECS space

FUND is called with an allocation block, and an increment to ECS space. It compares the master 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" are updated. "ECS in use" cannot exceed "Allocated ECS" and "\$ used" cannot exceed "\$."

FUND has three entry points:

FUND - B2 ... 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
 XØ ... ECS address of alloc bk
 X7 ... increment to ECS space

BLOCK MANIPULATION

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

On entry: B2 - size of block
 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)

when used? — 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 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 disaster 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 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 compact the block structure.

OBJECT CREATION AND DESTRUCTION

MAKEOBJ creates an object

FUND is called; an MOT entry 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
 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 DESTRUCTION

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

ECSINIT initializes ECS appropriately

Four ECS actions:

NEWUN changes a unique name

This is the system "Indian-giver" —

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.

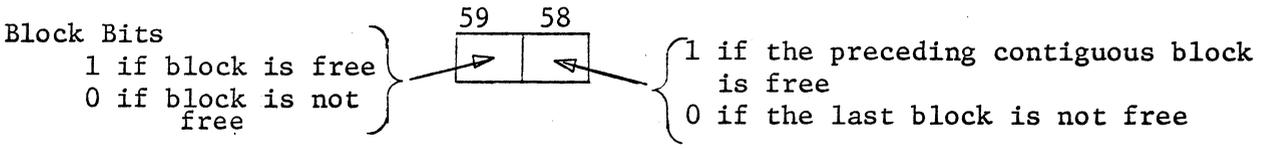
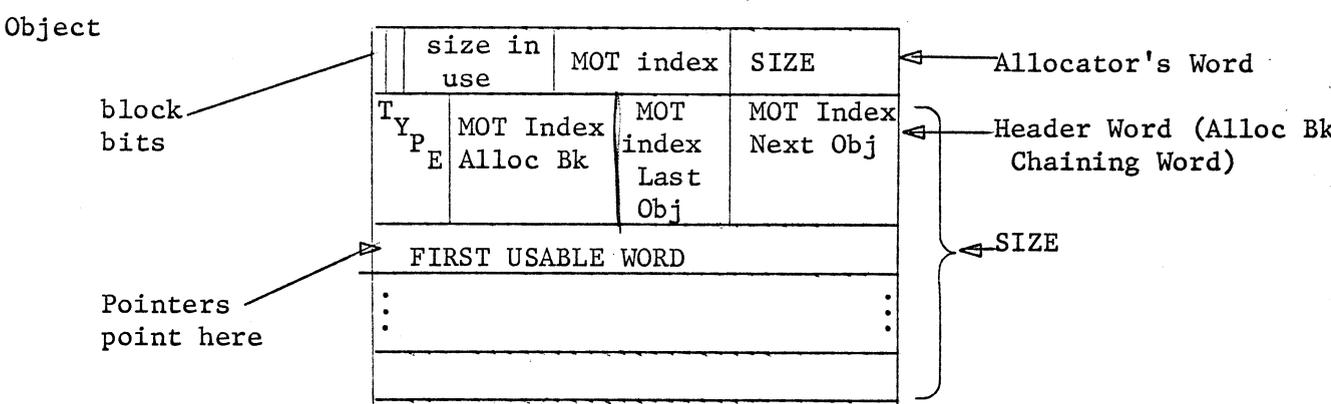
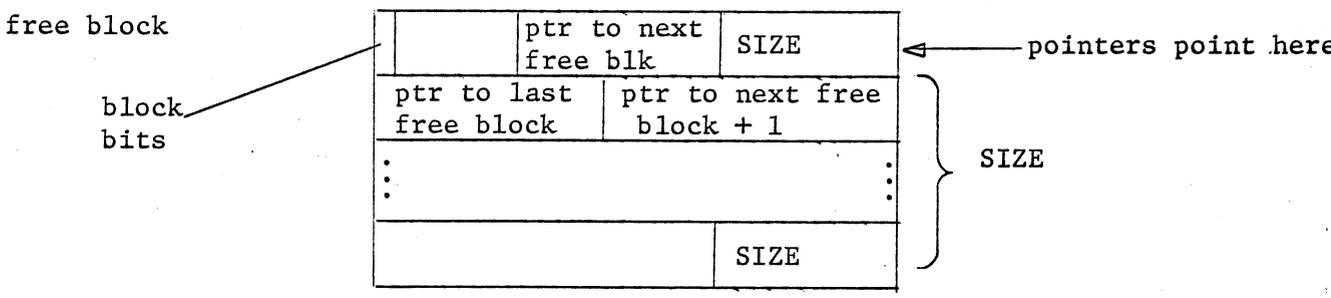
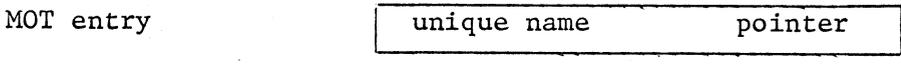
AP1 = C : Allocation block
 AP2 = D : Index for new capability

DONATE transfers space and money from one alloc bk to another

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

*i.e. initialize all available
 capabilities for the object*

PICTURES



Allocation Block

Alloc Word (see object)	
Header Word (see object)	
Allocated ECS	ECS in use
pointers to AB - chain	
time of last bill	\$ \$ \$
\$ used for CPU	\$ used for rent

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. 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 (MOT) 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.

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 data area) 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

OPTION MASK		TYPE	1 st WORD
UNIQUE NAME		MØT INDEX	

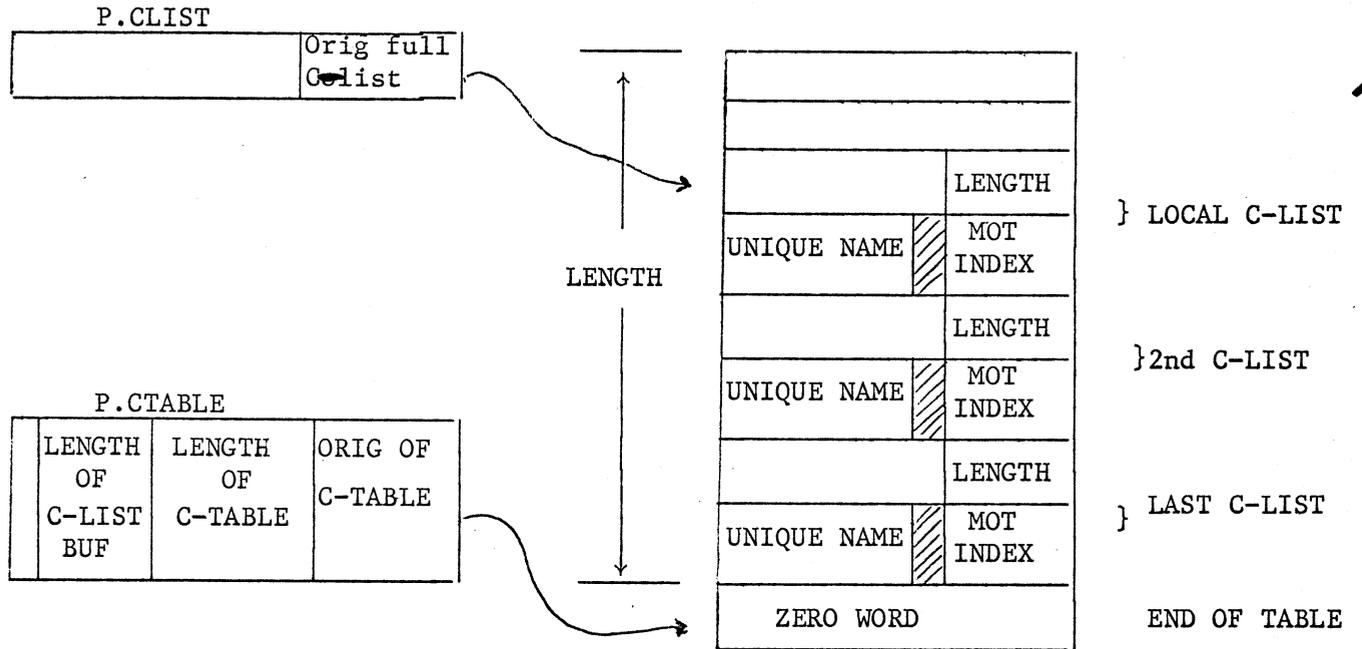
Figure 2

CAPABILITY LIST

		LENGTH	Number of Capabilities in C-list
OPTIONS		TYPE	} Capability (Index 0)
UNIQUE NAME		MØT	
OPTIONS		TYPE	} Capability (Index 1)
UNIQUE NAME		MØT	
OPTIONS		TYPE	} Capability (Index = LENGTH-1)
UNIQUE NAME		MØT	

Figure 3

FULL C-LIST TABLE



SUBPROCESS DESCRIPTOR (C-LIST DATA)

0			
1			
2			
3			
4			C-LIST LENGTH
5	C-LIST UNIQUE NAME	MOT INDEX	
6			
7			

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_0, S_1, \dots, S_n) $n \geq 0$, which describe the shape of the file. Each S_i , for $0 \leq i < n$, is the number of branches in the file tree at nodes of level i (the root of the tree is at level 0; all nodes connected to the root are at level 1; etc.). Each S_i for $i > 0$, must be an integral power of 2 (note: this does not apply to the first shape number S_0). 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 $L = \prod_{i=0}^n S_i$. 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, ~~which is stored in the file descriptor (see Fig. 2).~~ The file descriptor (see fig 2) is pointed to from the master object table (MOT) and contains the dope vector, the length of the file, a pointer to either a pointer block or a data block (zero 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 ($n > 0$). 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.

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 $\neq 0$).

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.

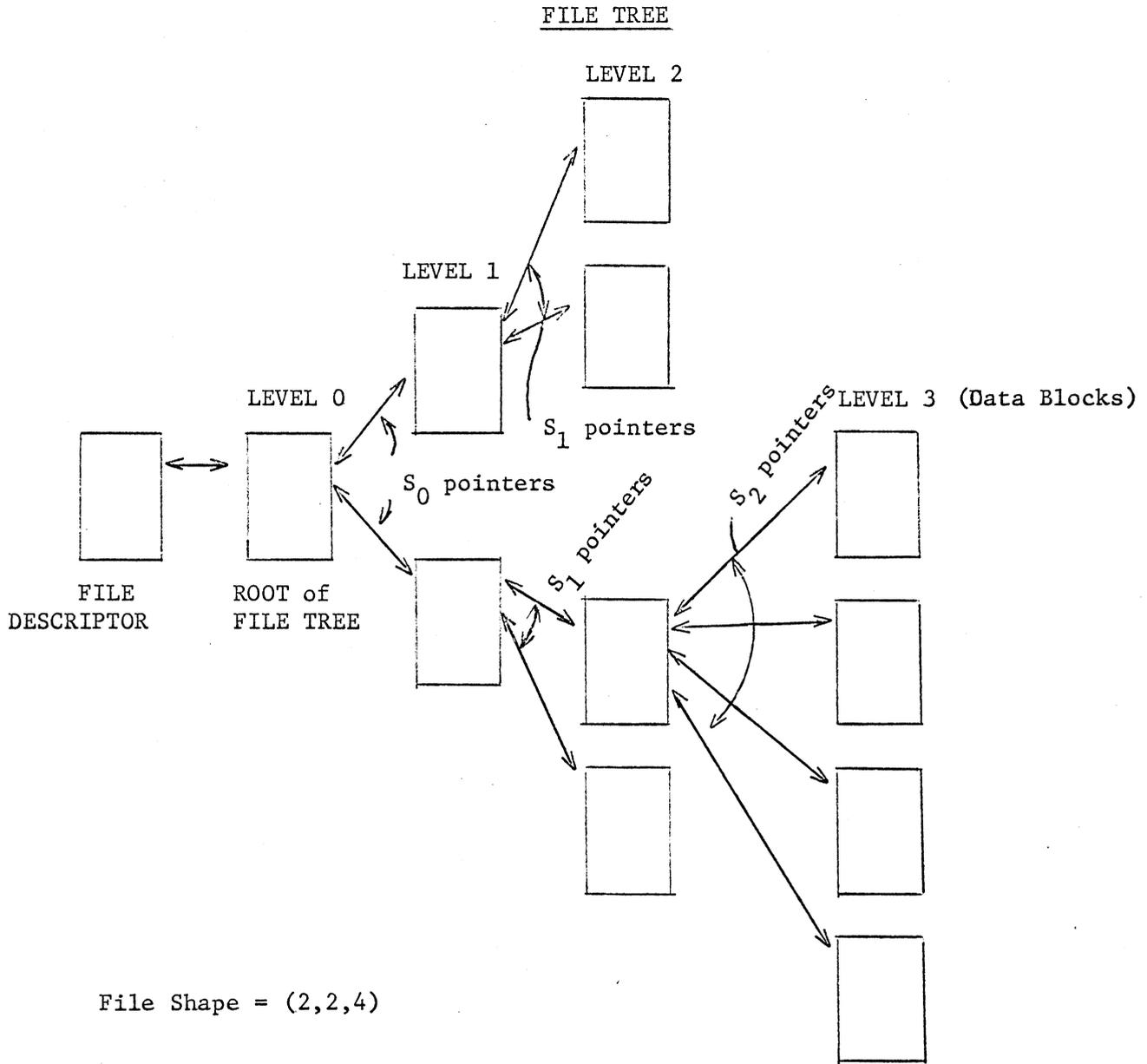
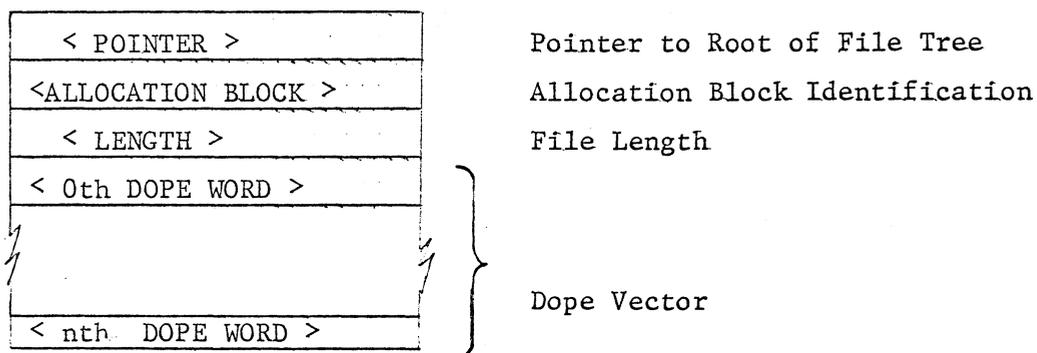


Figure 1

FILE DESCRIPTOR

$$\text{SHAPE} = (S_0, S_1, \dots, S_n)$$

< POINTER > ::=

+ 0

 If root doesn't exist

or

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

 If root is pointer block ($n > 0$)

or

12 1777	6 8 0	18 1	3 0	21 ABS ECS ADDR
------------	----------	---------	--------	-----------------------

 If root is Data Block ($n=0$)

< ALLOCATION BLOCK > ::=

39 Unique Name	3 MOT	18 Index
-------------------	----------	-------------

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

< 0th Dope Word > ::=

AX6	ℓ	MXO	0	JP	B7
-----	--------	-----	---	----	----

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

< 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
-----	---	----	------

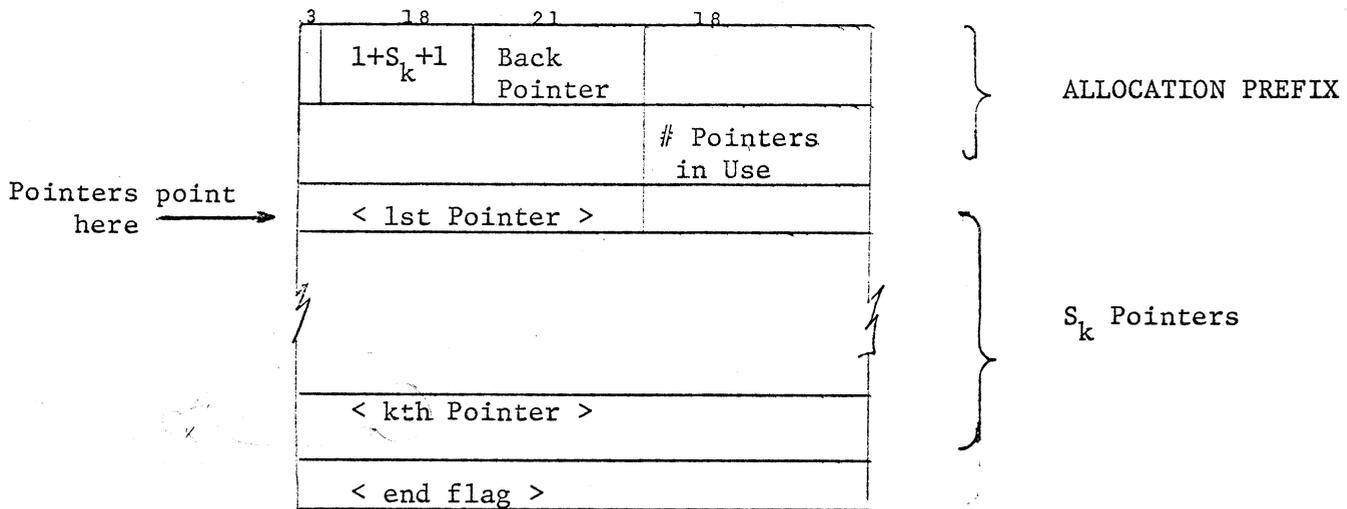
 $S = S_{n-1} \quad (n > 0)$

or

SB5	S	JP	B7+5
-----	---	----	------

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

Figure 2

POINTER BLOCKPointer block at level kShape = (S₀, S₁, ..., S_k, ..., S_n)

< jth pointer > ::= (j ≤ 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

Figure 3

DATA BLOCK

$$\text{Shape} = (S_0, S_1, \dots, S_n)$$

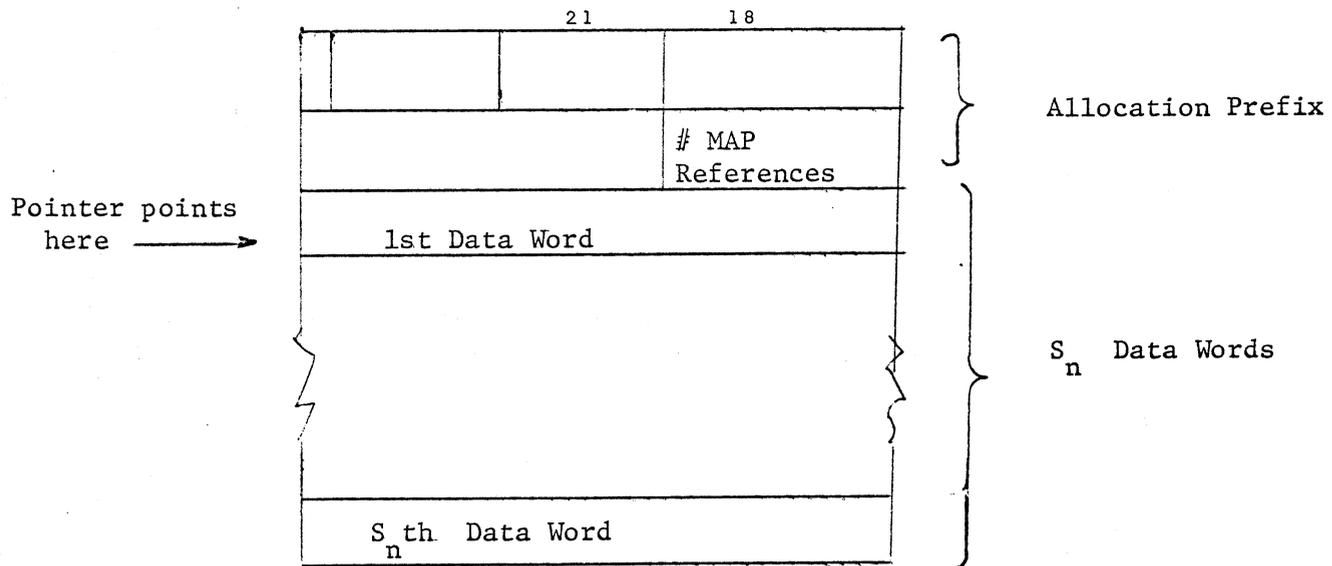


Figure 4

~~August 18,~~ ^{Sept 2} 1969

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, a copy of 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.USRBI 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)

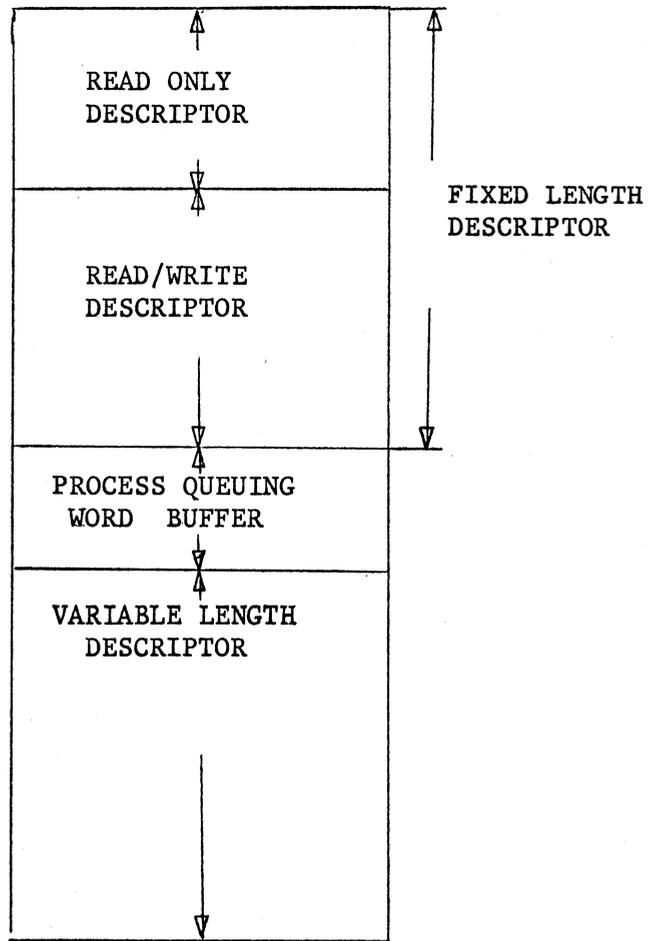


Figure 1

PROCESS DESCRIPTOR IN CORE

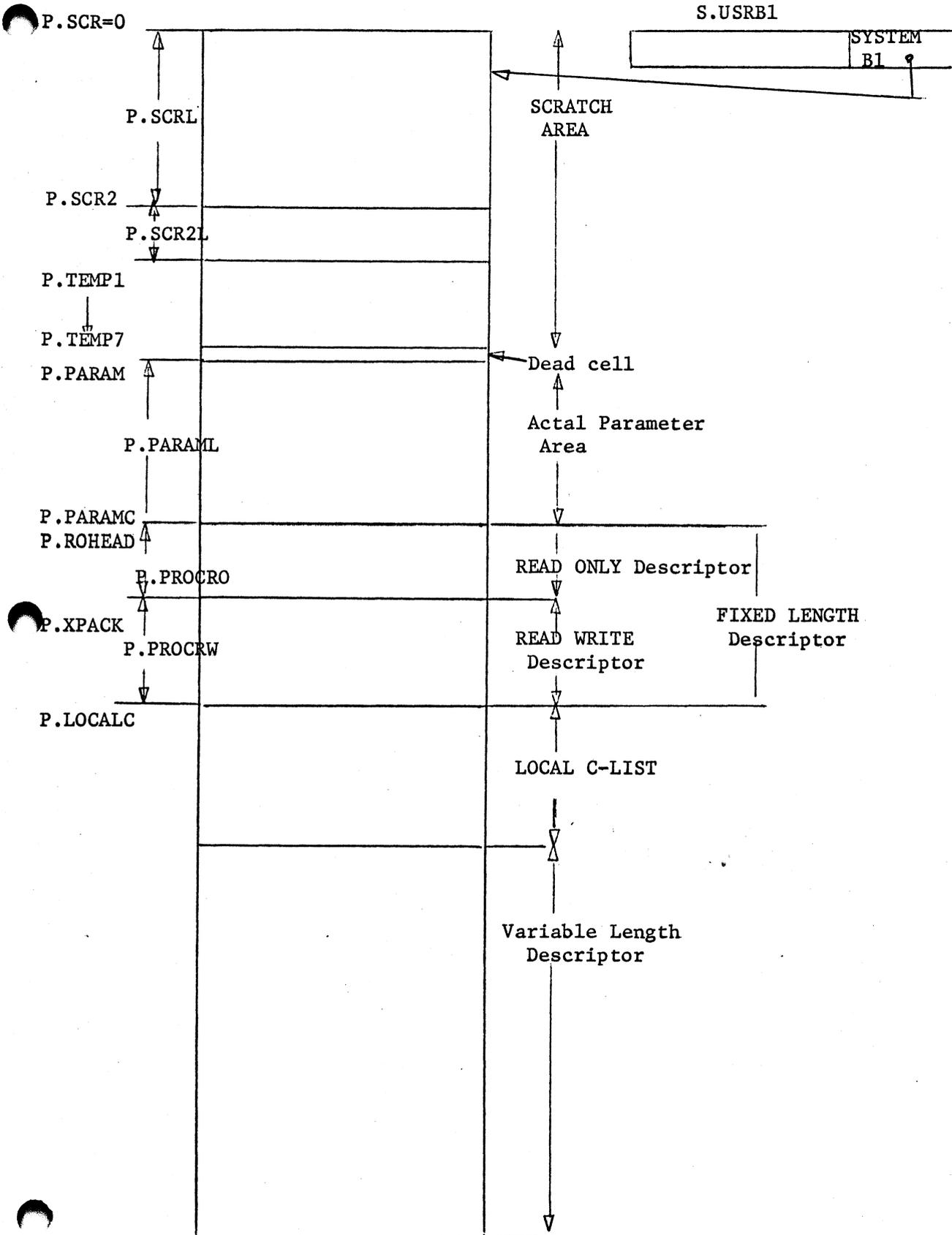
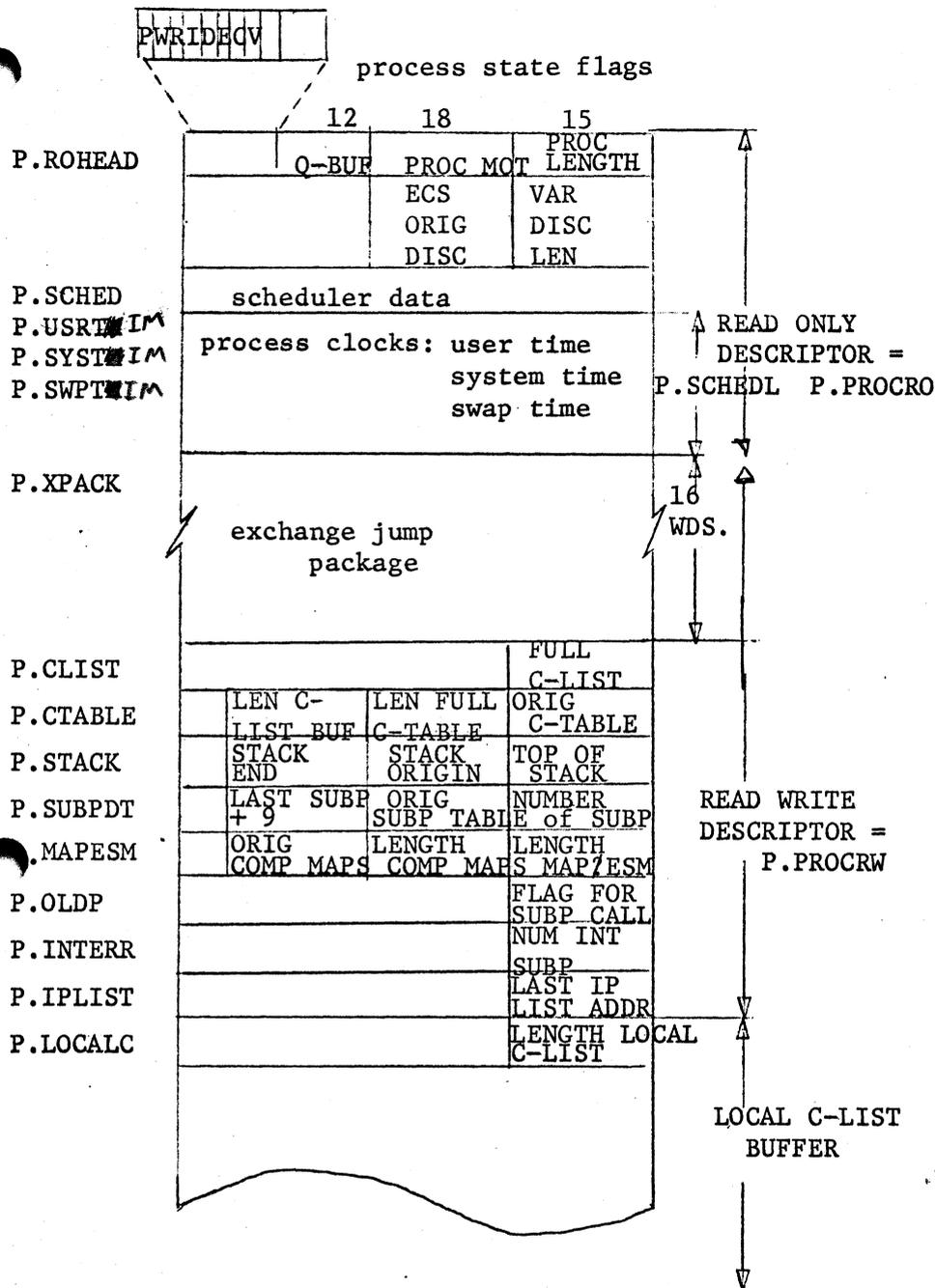


Figure 2

FIXED LENGTH DESCRIPTORState flags

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

W = "wake-up waiting"

R = "running"

I = "interrupt"

D = "destroy"

E = 0 ⇒ ECS process
1 ⇒ pseudo-process

C = process "in core"

V = "event"

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

<PROC MOT> ::= MOT index of process

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

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

<ECS ORIG DISC> ::= origin relative P.ROHEAD in ECS of variable length descriptor = Q-BUF + P.PROCRO + P.PROCRW

Figure 3

VARIABLE LENGTH DESCRIPTOR

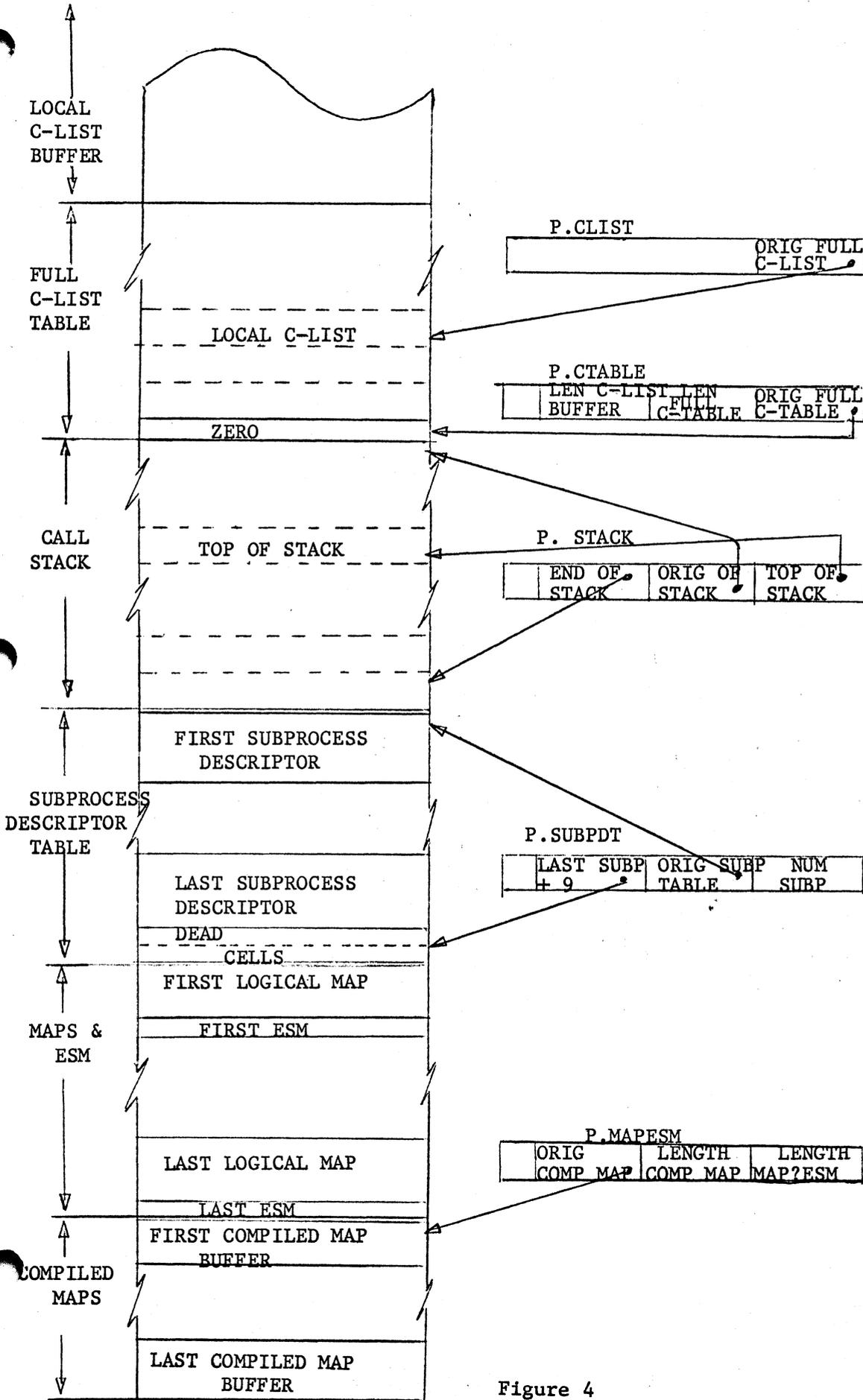


Figure 4

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.

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.

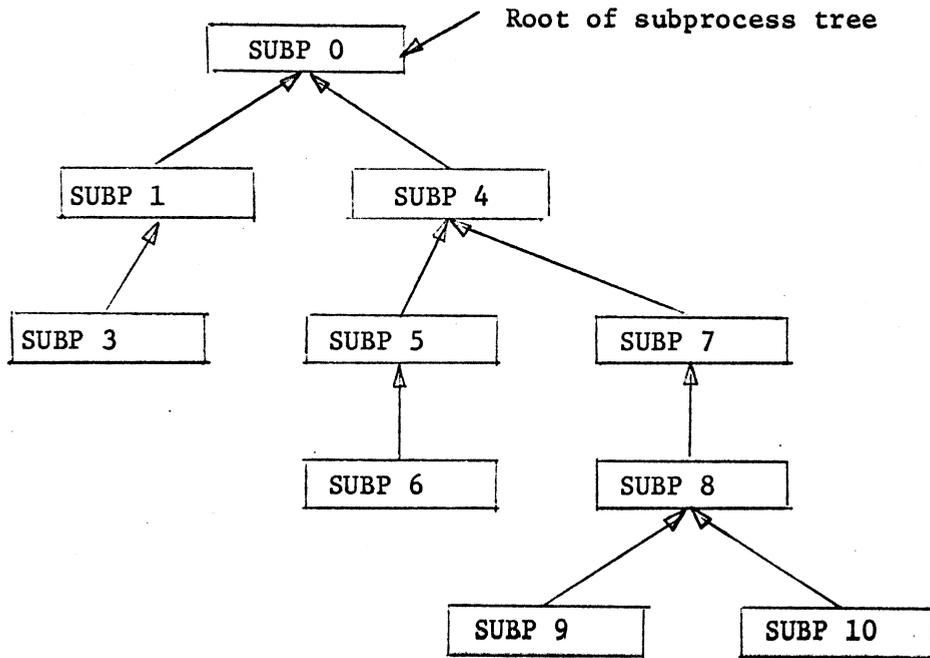
SUBPROCESS TREE

Figure 5

FULL PATH EXAMPLE

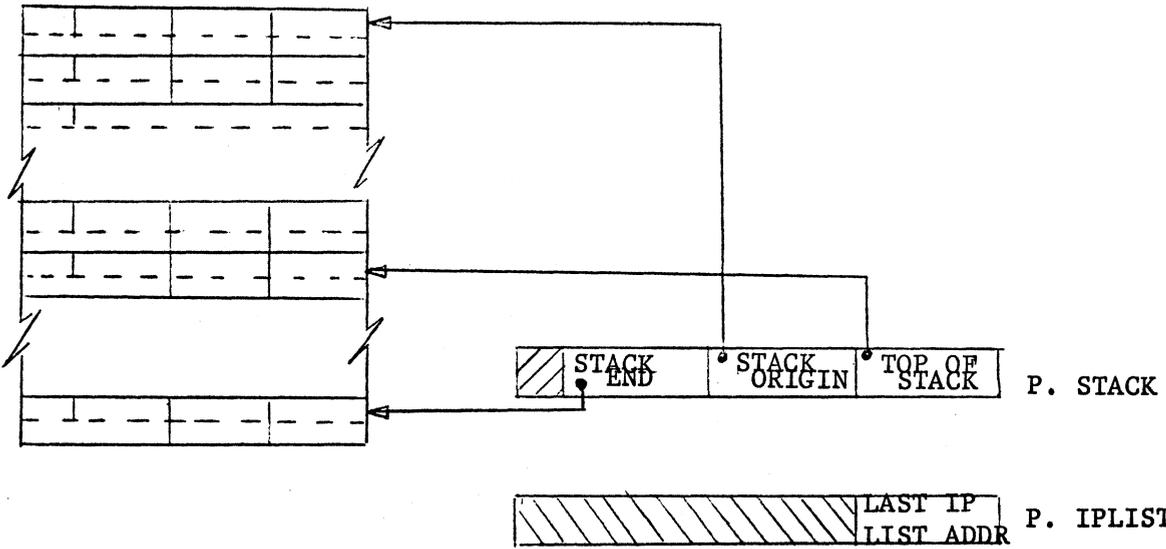
<u>CALLING SEQUENCE</u>	<u>CURRENT SUBP</u>	<u>END-OF-PATH SUBP</u>	<u>FULL PATH</u>
	SUBP0	SUBP0	SUBP0
SUBP0 calls SUBP9	SUBP9	SUBP9	SUBP9
SUBP9 calls SUBP6	SUBP6	SUBP6	SUBP6
SUBP6 calls SUBP4	SUBP4	SUBP6	SUBP6,5,4
SUBP4 calls SUBP0	SUBP0	SUBP6	SUBP6,5,4,0
SUBP0 calls SUBP5	SUBP5	SUBP6	SUBP6,5
SUBP5 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 (see Operations) 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.

CALL STACK



Forced Freturn
 Interrupted
 Interrupt Inhibit

STACK ENTRY

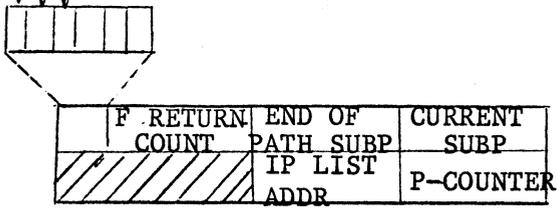


Figure 6

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)

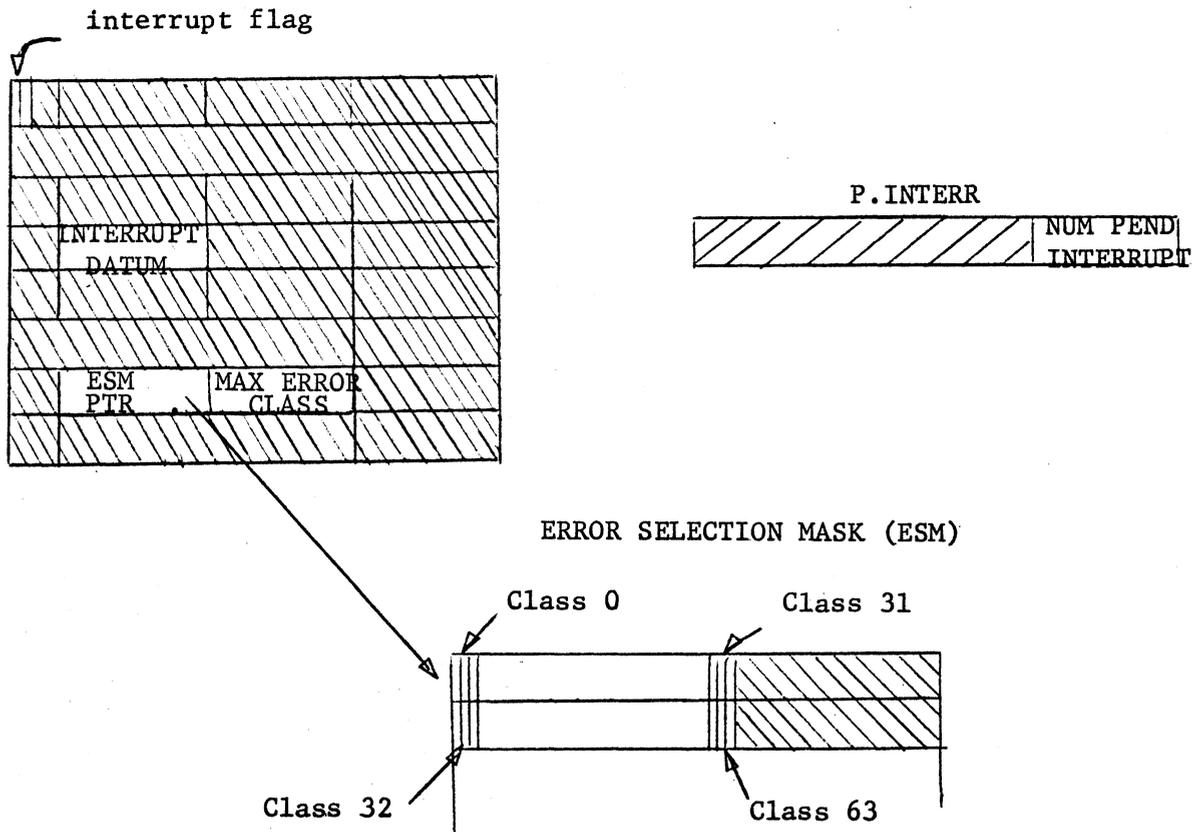


Figure 7

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 interrupt process). Thus one process can force a second process to enter a specified subprocess. Furthermore, the interrupt 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 ^{sub}process 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) ^{sub}process.

If the interrupt process is "hung" when a process interrupt is initiated, the "ancestors" of the current subprocess (of the interrupt 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 interrupt process and, if necessary, the process is scheduled to run.

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

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).

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 the 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		
WORD 0	RA + FL	RA	PTR TO FATHER
WORD 1	SUBPROCESS NAME		
WORD 2	ENTRY POINT	MAP ORIGIN	C-LIST ORIGIN
WORD 3	COMP BUF SIZE	LOGICAL MAP PTR	COMPILED MAP PTR
WORD 4	INTERRUPT DATUM	# LOGICAL MAP ENTRIES	C-LIST LENGTH LENGTH
WORD 5	C-LIST UNIQUE NAME		C-LIST MOT
WORD 6	ESM POINTER	MAX ERROR CLASS	MAX STACK POINTER
WORD 7	-0-	# CAPAB PARAMETERS	MAPIN LIST LINK

- 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)
 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 permissible 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 re-computed, 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 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 Operations).

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 local environment) of the address space, C-list, and map of the calling subprocess are computed and stored in cells 2, 3, and 4 of the full address space. If the calling subprocess is not a member of the new full-path

(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.PARAM - 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 transferred, 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 5 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.

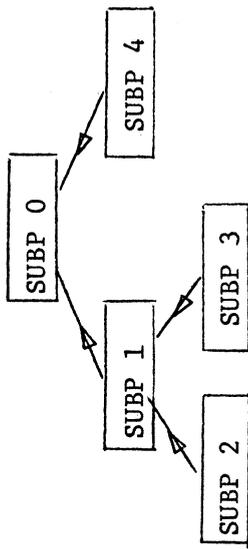
An error subprocess call is initiated by the ECS system or by user request. An error subprocess call passes as its parameters the error class and error number which describe the error causing the call. Also, the bit in the ESM of the error subprocess corresponding to the error class must be reset to 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 Operations).

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	6	10B
map length	4	5	10B	6	3
map origin	0	4	11B	11B	4



Subprocess Tree

DYNAMIC STRUCTURE

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

Subprocess Calling Example

Figure 2

Class Codes

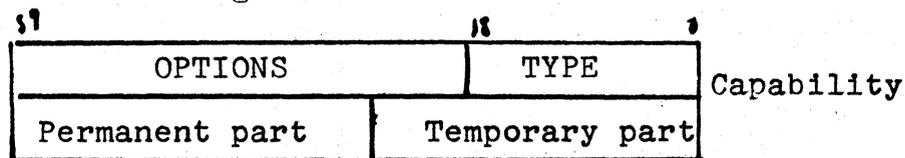
A Class code is a protected 60-bit datum by which a user may identify himself or some ECS system object. Within the ECS system; class codes are used as the names of sub-processes (See SUBPROCESSES); in the future they will be used to identify users within the disk system and will be called access keys.

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 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 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



Sept 4,
~~July 22,~~ 1969

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 ^{maximum} number of words in ~~to be added to~~ the compiled map are declared when the subprocess is created and may not be altered. (i.e. SIZE of compiled map buffer)

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 map 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. X

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 whole map is not swapped. 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.

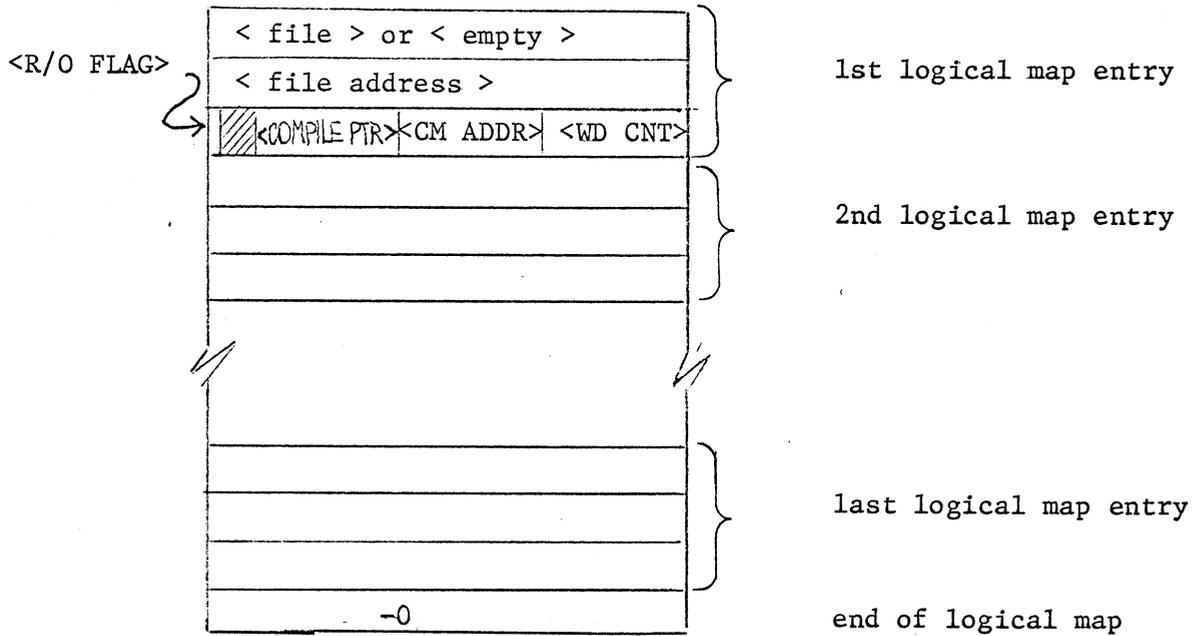
the corresponding file block

the new block

address space

X

LOGICAL MAP



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

< file > ::=

39 UNIQUE NAME	18 MOT INDEX
-------------------	-----------------

 file identification

< file address > ::= $0 \rightarrow 2^{60} - 1$

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

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

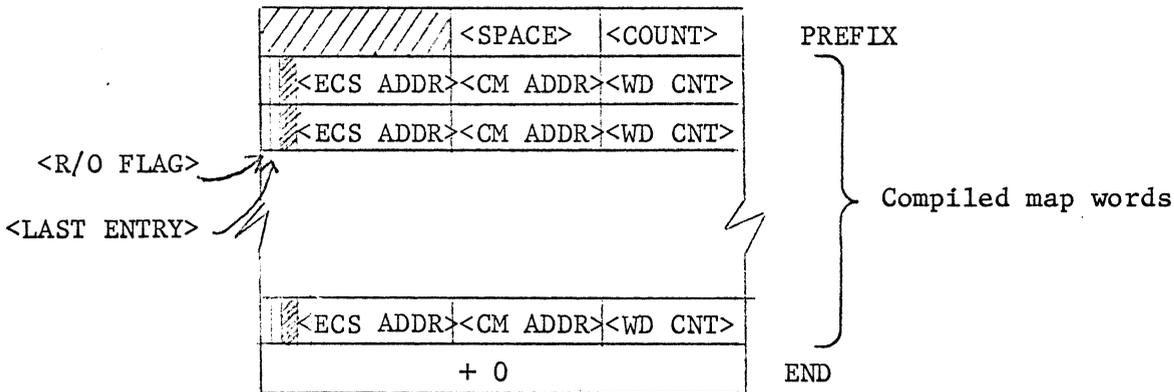
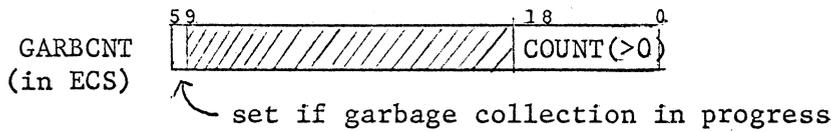
< CM ADDR > ::= CM address within subprocess local address space

< WD CNT > ::= word count

Note: $\langle \text{CM ADDR} \rangle + \langle \text{WD CNT} \rangle \leq$ length of subprocess local address space

Figure 1

COMPILED MAP



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

< 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

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

Figure 2

SUBPROCESS DESCRIPTOR (MAP DATA)

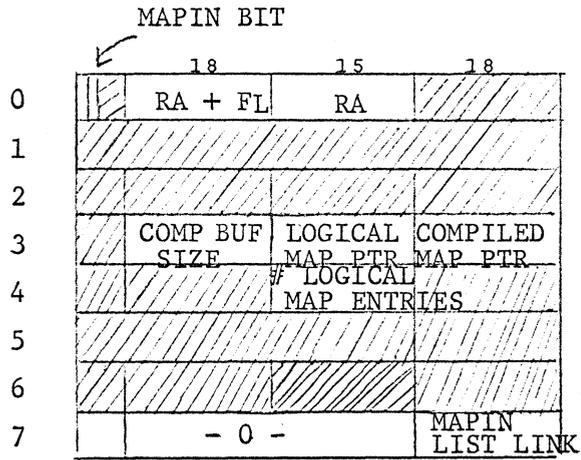


Figure 3

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~~ ^{two-word slots} 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. X

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. X

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) non-zero. 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 the MOT index of the process which is sending the event. 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.

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.

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 conditions, 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. X If the process is not running ("running" flag) the scheduler is called to schedule the process to run. X 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. unclea

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).

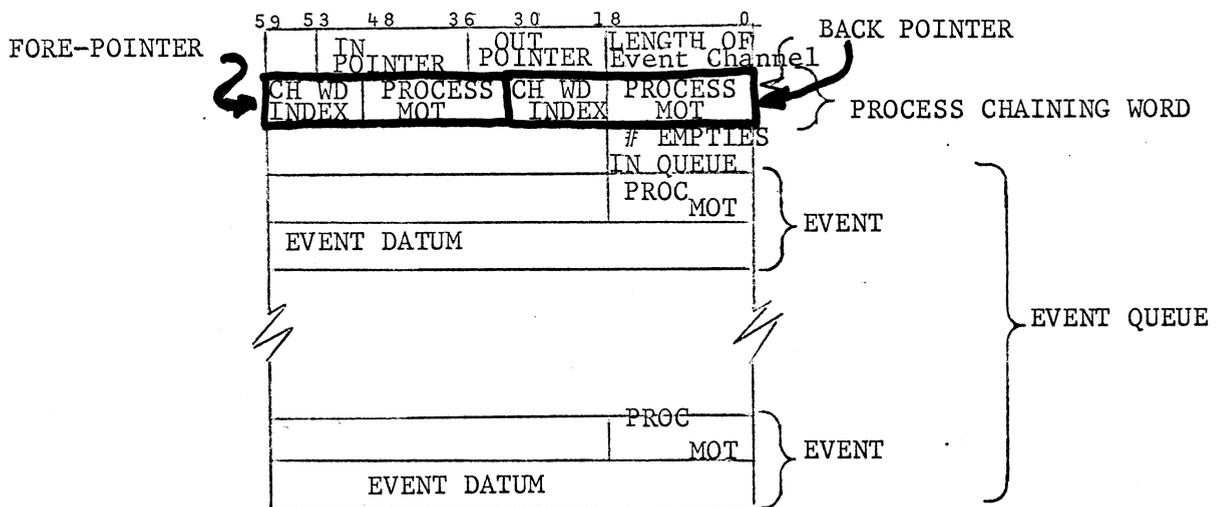
If the event queue is empty the process must be added to the queue of waiting processes (process queue) using a process queueing word in the ECS image of the process. The "running" flag in the process is cleared and the process is removed from the scheduling queue (de-scheduled). Next, the P-counter 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



PROCESS QUEUE EXAMPLE

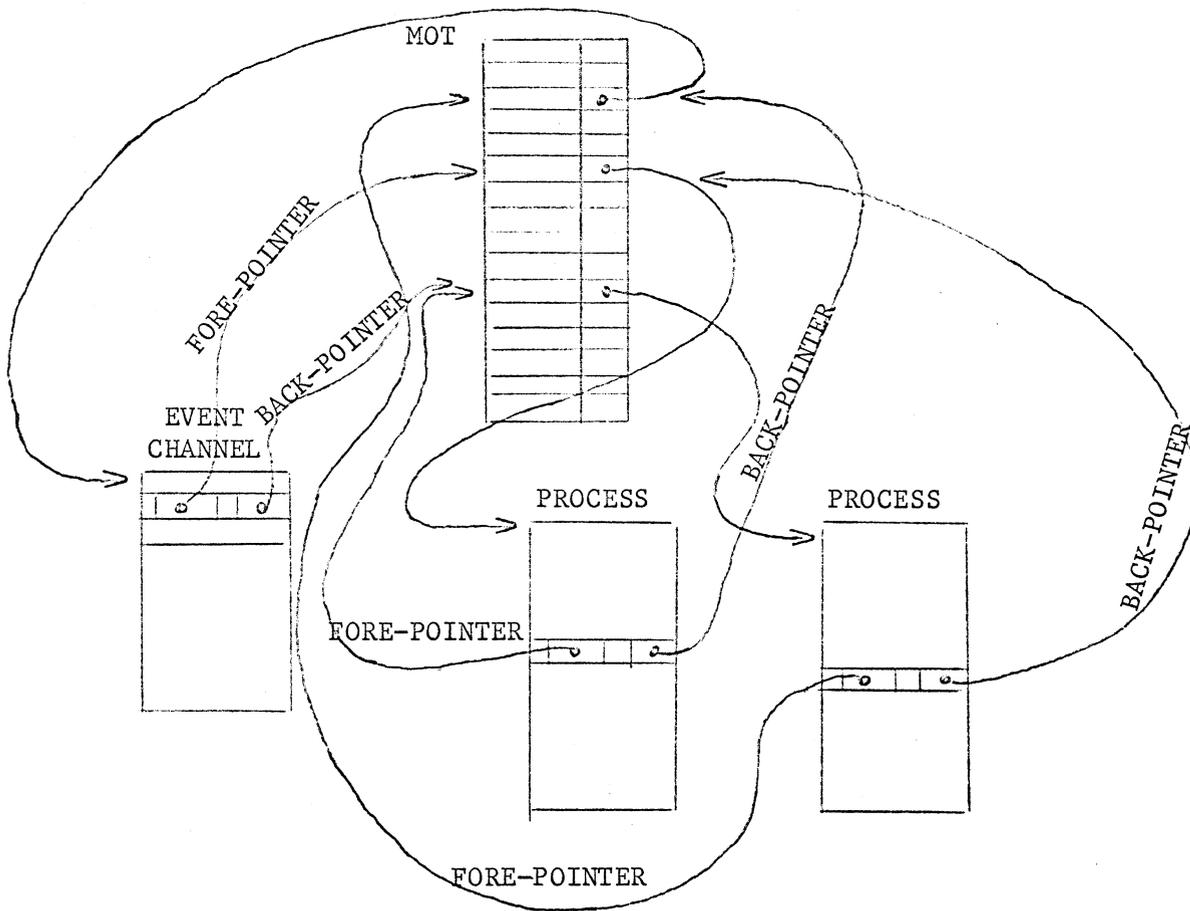


Figure 3

RESPONSES TO EVENT SENDER

<u>CONDITION</u>	<u>RESPONSE</u> (X6 for user call)
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 ($= 155_8$). 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.

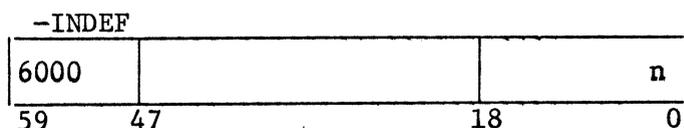
Character Representation

The internal ASCII code used in System Standard Text is the external ASCII + $140_8 \pmod{200_8}$. The conversion is performed by the system I/O routines (see

Section). 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

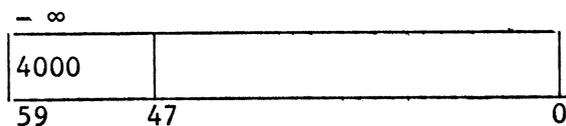
A sloppy segment in the Systext file is a group of n words ($0 < n < 2^{18}$) that are to be ignored. The first word of such a segment is of the form:



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.

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

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

Table 2

Non-Graphic TTY Character Representation

<u>Character</u>	<u>Internal ASCII Representation</u>	<u>Key Combination Systext Representation</u>	<u>Function</u>
NUL	140	% @	
SOH	141	% A	
STX	142	% B	
ETX	143	% C	
EOF	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*:

<u>Operation</u>	<u>Control Characters</u> *	<u>Action</u>
Accept	(CR)	The current new line is accepted as is.
Type State	(LF)	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 <u>up to</u> the current position of the cursor. e.g.: remainder of old line current new line ↑ (New cursor)
Concatenate and Accept	(CTRL) (@P)	Concatenates the remainder of old line onto the current new line and accept.
Concatenate, Print and Accept	(CTRL) (←0)	Concatenates the rest of the old line onto the new line, prints it out, and accepts.
Tab Set/Release	(CTRL) (Ik)	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.

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

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

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.

<u>Operation</u>	<u>Control Characters</u>	<u>Action</u>
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 RUF

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 BELG

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 IH

The remainder of the old line is appended to the new line and printed.

Note that CTRL H CR is equi-

valent 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 EORB

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.

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 > .

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 133₈ 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 string descriptor. 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 = 23₈) 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 = 101₈). 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 = 76₈) 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 anomaly 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_g) 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.

<u>Table Number</u>	<u>Characters which signal a break</u>
0	none
1	any character
2	non-graphics
3	non-alphanumerics
4	non-numeric