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 slave (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 data area has been picked up in B1 by the exchange jump. The origin of the process data area 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 data area.

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 data area at P.IPLIST in case it is needed to form a stack entry for a subprocess call. Also the stack manupulation 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 (user RA + IP list pointer) 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 data area 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 capacility 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. If so, the subprocess name is copied from the operation to P.PARAMC in the process data area and the bit mask(s) are copied into the cells preceding P.PARAMC.

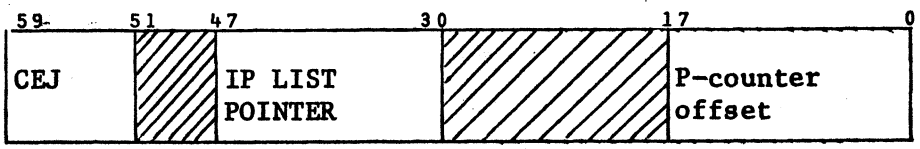
Finally, the ECS action number is extracted from the operation; it is used as an index to jump into the ECS action jump table starting at ACTIONL where there will be a jump to the proper entry point for the desired ECS action.

Upon completion of an ECS action, the ECS action routine normally returns to the system entry/exit routins 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.SYSTIM 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



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. Thus, the user can transfer the right to access an oabject 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). It uses the low order 18 bits of the second word, which contain the master object table (MØT) index, and the high order 39 bits, which contain the unique name of the object. The remaining 3 bits of the second word are unused.

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

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. 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 data area (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 data area 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 data area, holds a pointer to the end of the full C-list table (the zero word), the number of entries allowed in the table (maximum length of the full path), and the size of the core buffer for the local C-list (maximum local C-list size).

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

Figure 1

CAPABILITY

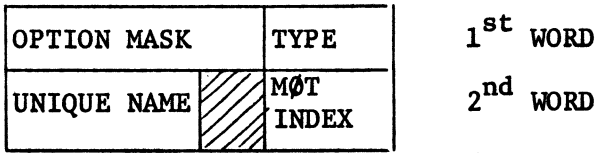


Figure 2

CAPABILITY LIST

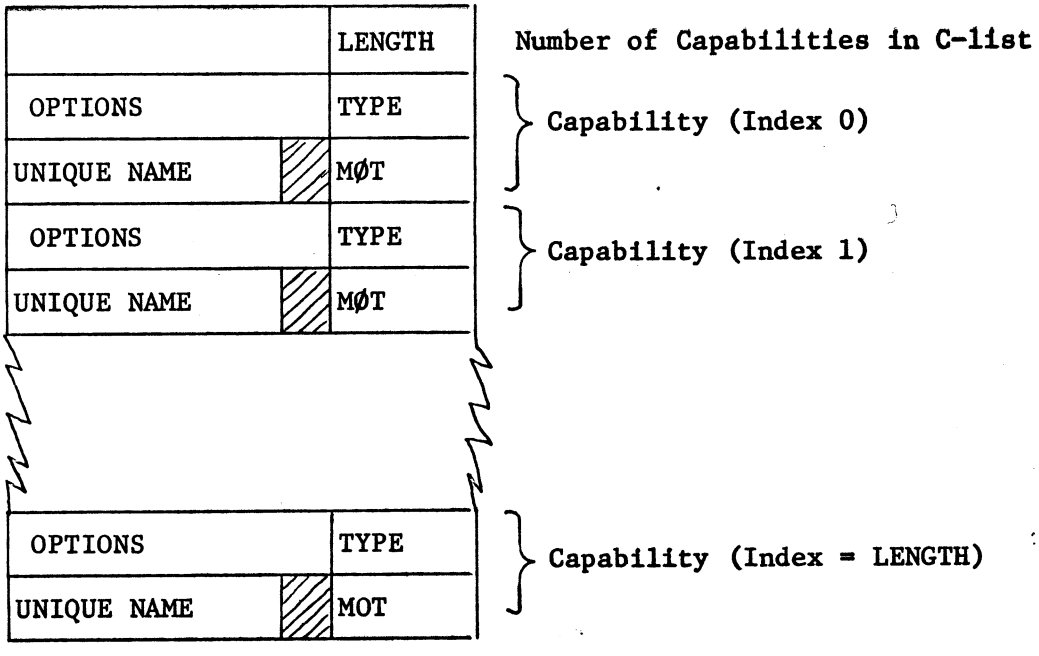
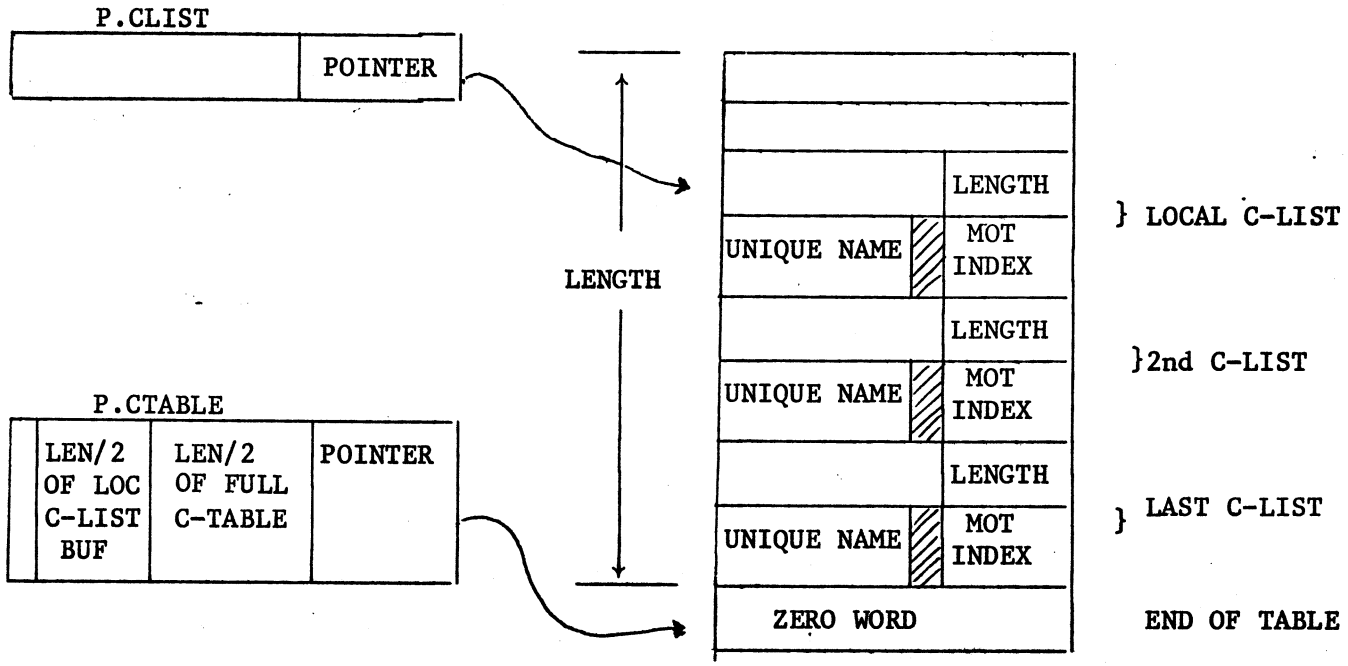


Figure 3

FULL C-LIST TABLE



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 (see Fig. 1) are called data blocks (see Fig. 2) 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, $(S_0, S_1, \ldots, S_n)$ $n \geq 1$, 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 and is stored in the file descriptor (see Fig. 4). The file descriptor is pointed to from the master object table (MOT). It contains the dope vector, the length of the file, a pointer to either a pointer block or a data block (zero 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 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 contain the addressable words of the file. Again, the backpointer in the allocation prefix is made use of, and 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 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.

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

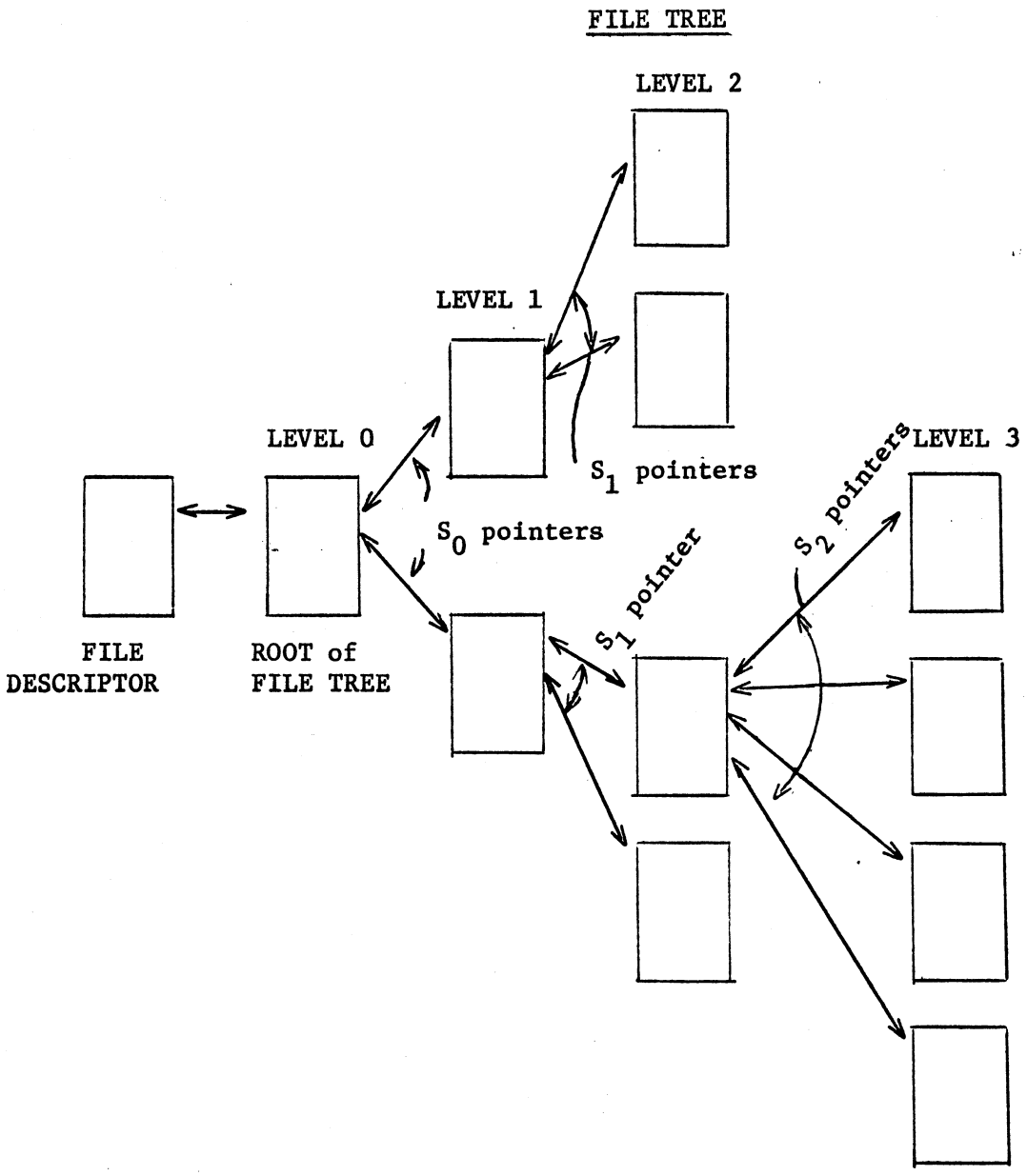

FILE TREE
LEVEL 2
LEVEL 1
LEVEL 0
LEVEL 3
S_1 pointers
S_0 pointers
S_1 pointer
S_2 pointers
FILE DESCRIPTOR
ROOT of FILE TREE

Figure 1

DATA BLOCK

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

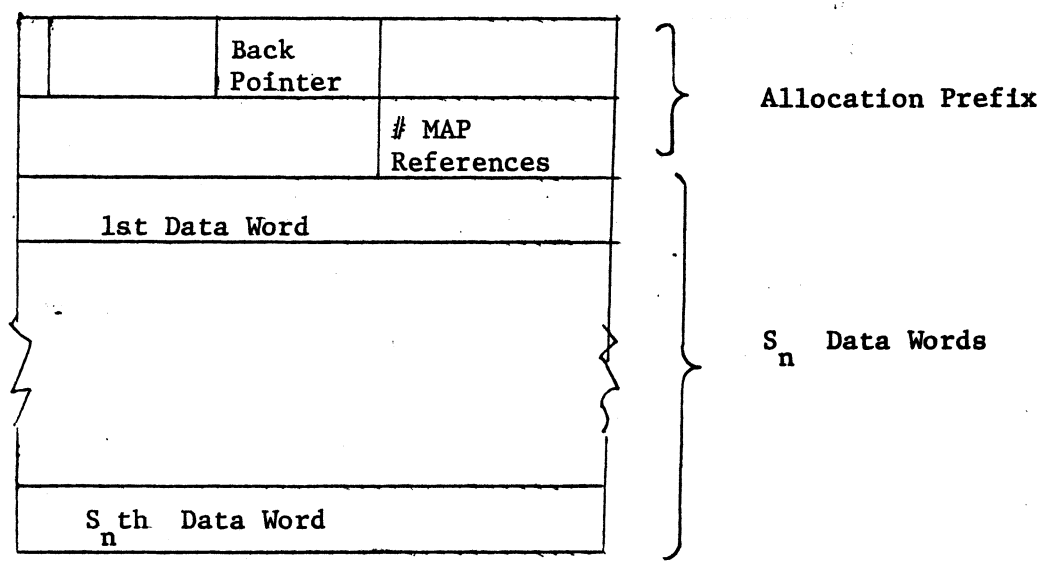
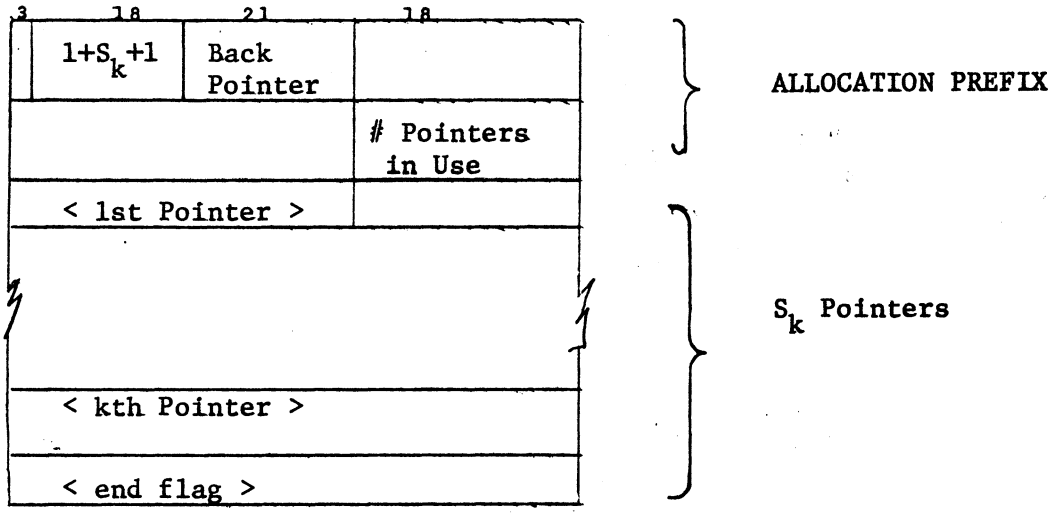


Figure 2

POINTER BLOCK

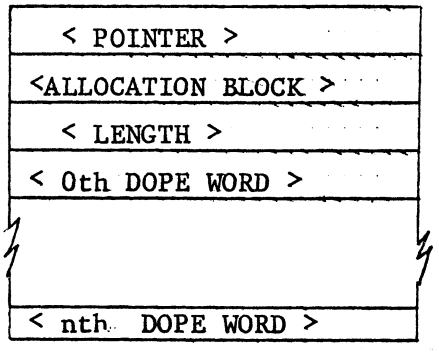
Pointer block at level k



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

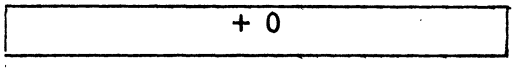
< jth pointer > ::= (j ≤ k)	+ 0		Corresponding pointer or data block doesn't exist
	12 / 6 / 18 / 3 / 21		
or	0000 / 0 / j / 0 / ABS ECS POINTER		Corresponding pointer block (k < n-1)
	12 / 6 / 18 / 3 / 21		
or	1777_8 / 0 / j / 0 / ABS ECS POINTER		Corresponding data block (k = n-1)
< END FLAG >	$-(S_k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

FILE DESCRIPTOR



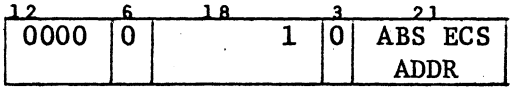
SHAPE = $(S_0, S_1, \ldots, S_n)$

< POINTER > :: =



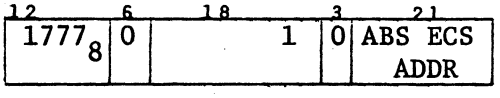
If root doesn't exist

or



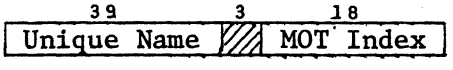
If root is pointer block (m > 0)

or



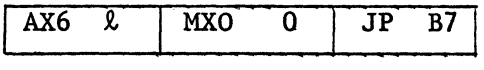
If root is Data Block (m=0)

< ALLOCATION BLOCK> ::=



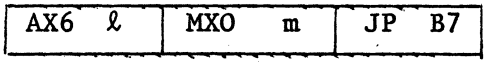
< LENGTH > ::= (maximum file address) + 1 = $\prod_{i=0}^{n} S_i$

< 0th Dope Word > ::=



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

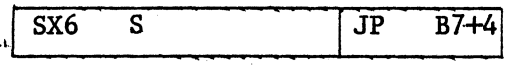
< Jth Dope Word > ::=



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

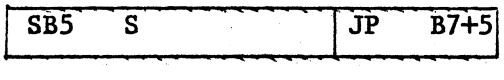
$m = 60 - \log_2 (S_j)$

< nth Dope Word > ::=



$S = S_n \quad (m < 0)$

or



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

Figure 4

June 5, 1969

Event Channels

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

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

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

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

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 relax 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 receive 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 queues and the pointers 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 and the event is copied to the process data area in ECS. Note that the testing and setting of the "wake-up waiting" flag must not be interrupted by any other access to this flag. If the process is not running ("running" flag) the scheduler is called to schedule the process to run. If the first process without "wake-up waiting" is a pseudo process, it is removed from the process queue; otherwise, it is not removed until the process is swapped in to run.

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 oen 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.

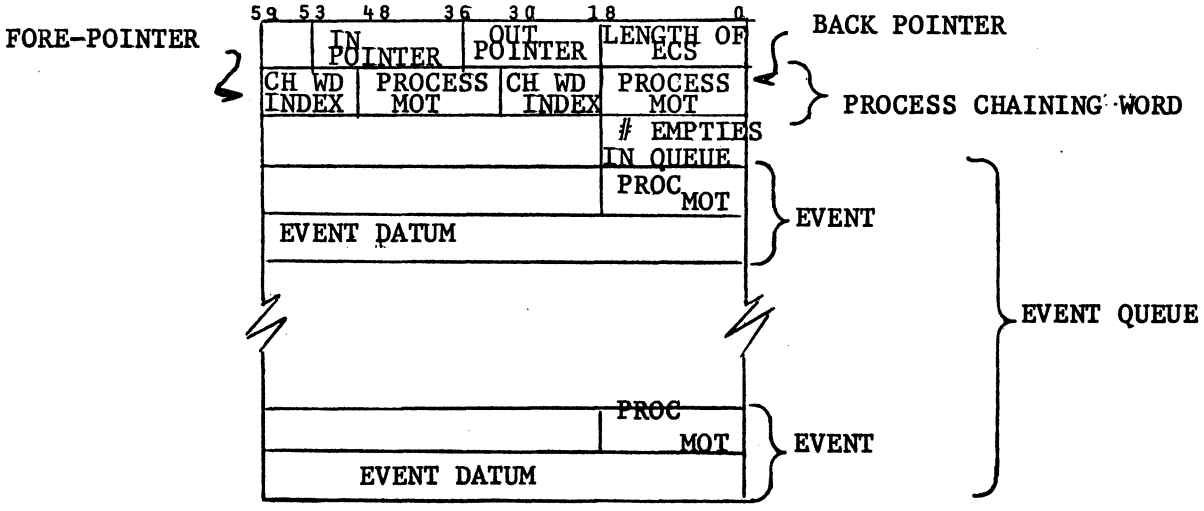
When an event arrives for a process which is hung on an event channel, the event sending mechanism will set the appropriate flags and schedule the process to run as described above. The swapper will detect the "event" flag and return through the event channel routines instead of directly to the system entry/exit routines. The swapper will have already removed the process from any process queues on which it had been hung. After restoring the P-counter of the process and copying the event to registers X6 and X7 of the process, the event channel routines return through SYSRET in the system entry/exit routines.

* why doesn't this end up hanging the process on the event channel two or more times? Does "TEST-EVENT-OR-BLOCK" check the process queue of the channel to avoid hanging it on twice?

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 must be removed from all the process queues on which it is already chained. In addition, the event is copied to X6 and X7 of the process, 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

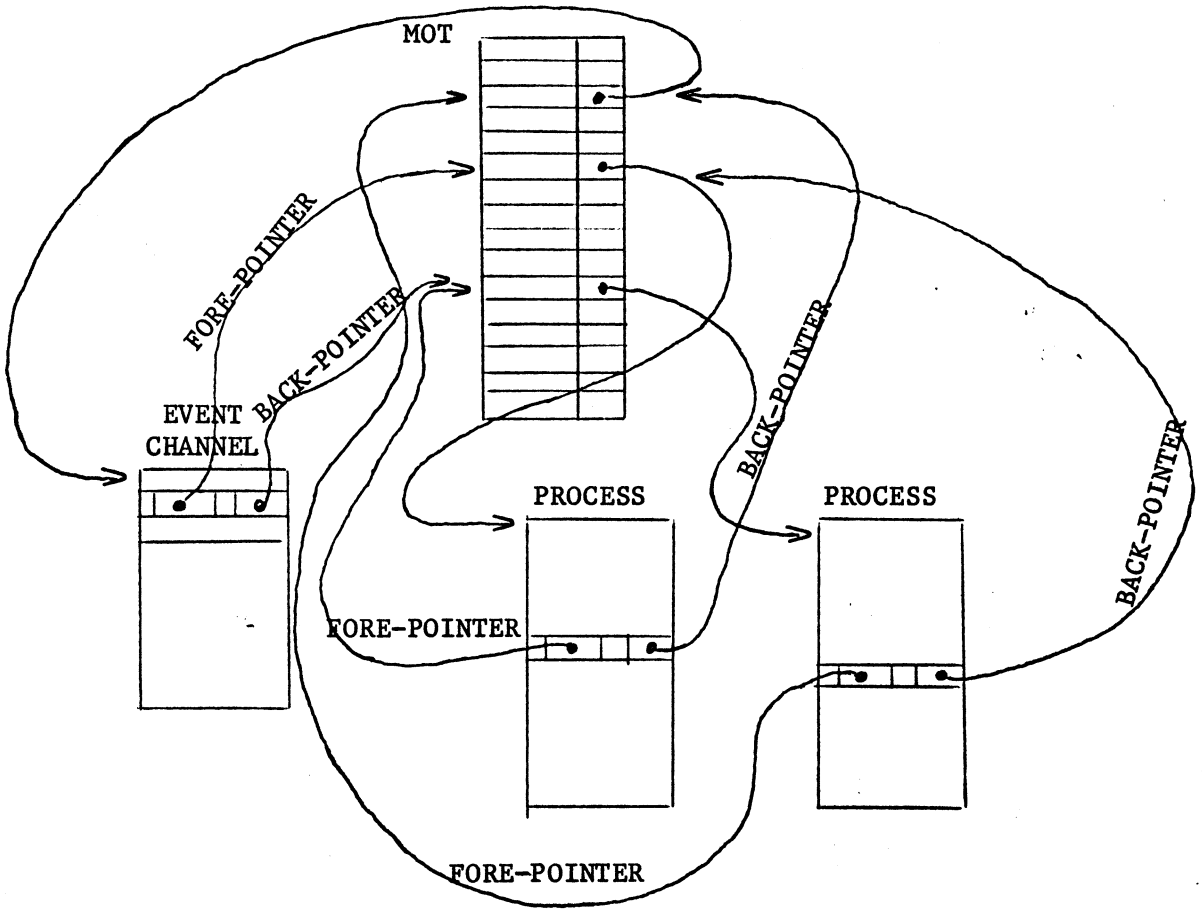


Figure 3

RESPONSES TO EVENT SENDER

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

Time Sharing System Text Standard

The System Standard Text (Systext) is the standard method of storing information for the Time Sharing System. Information in Systext format exists as a file (a semi-infinite array of 60-bit words) 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 ignored, 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 right-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:

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' ($0 < n < 128_{10}$) . On output these two characters are replaced by n blank characters.

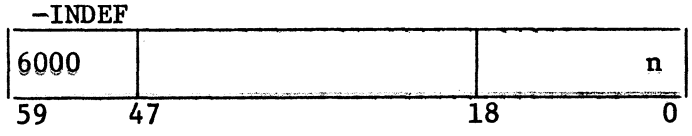
Character Representation

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

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

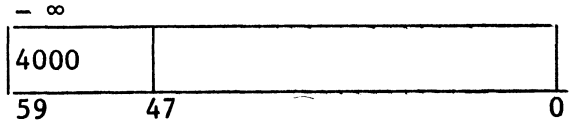
A sloppy segment in the Systext file is a group of n words ($0 < n < 2^{18}$) that are to be ignored. The first word of such a segment is of the form:



where n is the count of words in the segment. The system ignored 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

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

Table 2

Non-Graphic TTY Character Representation

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

The Line Collector

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

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

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

Operation	Control Characters*	Action
Accept	CR	The current new line is accepted as is.
Type State	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 up to 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 [k	Sets (releases) a tab stop at the current position of the cursor in the new line if entered an odd (even) number of times.
Tab	CTRL TAB I	Inserts blanks up (both cursors advance) to the next tab stop.

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.

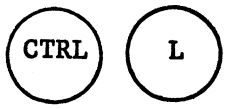
line of the next character typed in after the control characters. If at any time an edit request is made which cannot be fulfilled, the line collector echoes a bell instead of the graphic specified.

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

* The old cursor moves simultaneously with the new cursor.

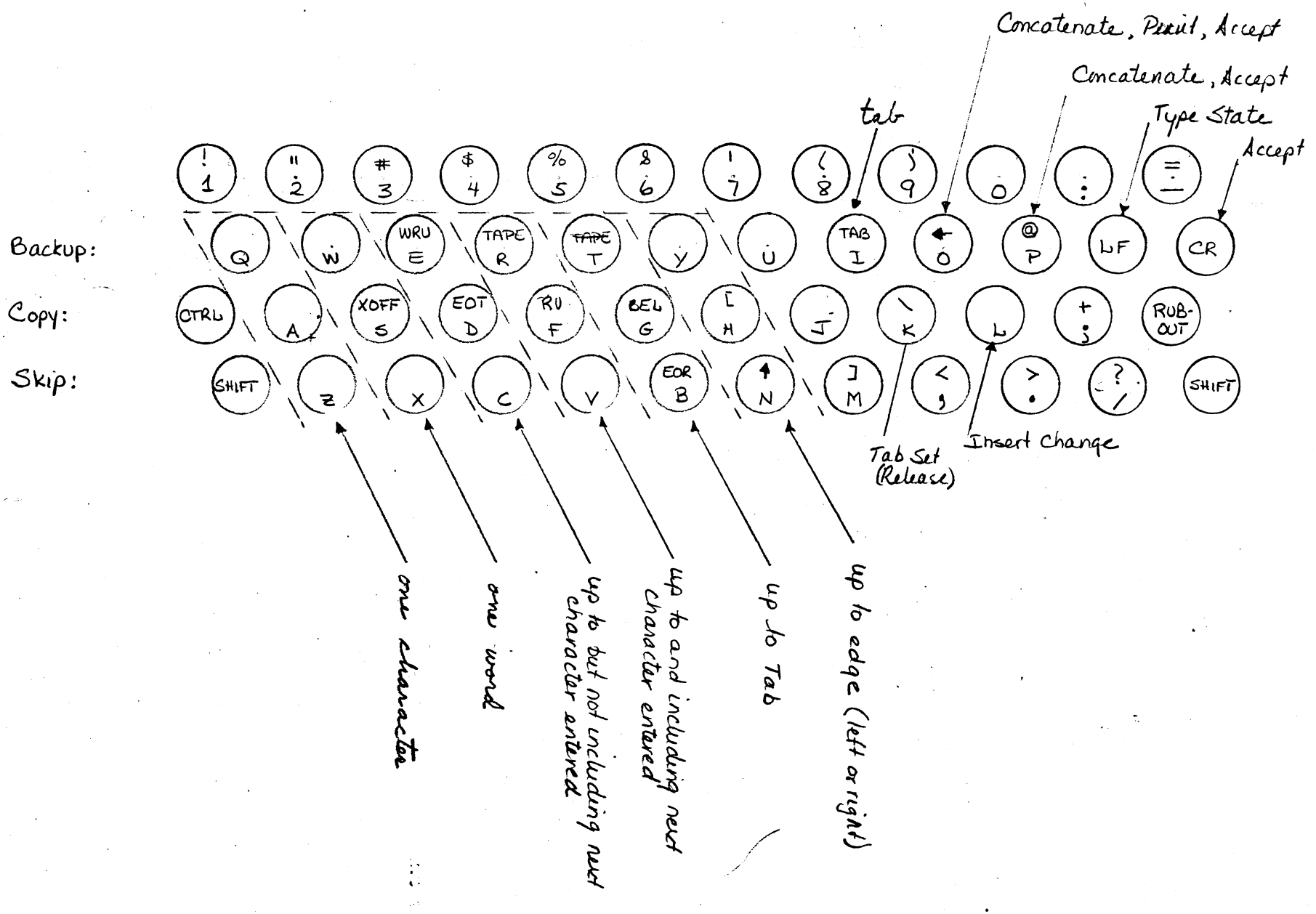
Copy up to and including next character entered	CTRL RU F	Characters in the old line up to and including the next character entered are appended to the new line and printed.
Copy to tab	CTRL BEL G	Characters in the old line up to the next tab setting are appended to the new line and printed.
Copy rest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

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

Insert Change: 

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 > .

Table 3
(33/35) Teletype Keyboard and Control Characters



Teletype I/O Functions

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

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

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

X7 is set to the return address in calling program.

I/O operations are performed upon strings or lines where a string is a sequence of characters and a line is a string terminated by a CR character. Every string or line is quantified by a two word entity called a string descriptor. The first word of a string descriptor points to the 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, DECS
            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 are output by .PUTOUT.

NOTE: If the flag at TTYBUFF + FORCE (FORCE = 23_8) in the TTY data area is on, the TTY buffer will be flushed (PP is notified that there is something in the buffer) each time .PUTOUT finishes a line. This kind of line-by-line 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 and/or when the last line is entered, whichever occurs first.

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

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

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

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

Input

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

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

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

The following macro call to INGET, enables the user to implement 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 use % 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 strinc descriptor DESC.

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

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

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

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

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

GETCTTY does not echo the retrieved character even if the SOFTECHO ($= 21_8$) flag in TTYBUFF is on. (The SOFTECHO flag signals that the PP has not been able to echo a character and therefore that GETS should.) The retrieved character is returned in X1 .

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

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

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

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