[redacted]

ETS SYSTEM USER'S GUIDE

by

Donald Wayne Oxley

May, 1972

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN · URBANA, ILLINOIS

UIUCDCS-R-72-523

ETS SYSTEM USER'S GUIDE

by

Donald Wayne Oxley

May, 1972

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

*This work was supported in part by the National Science Foundation under Grant No. US NSF GJ-812.

Warning to the Unwary

This manual is intended for use by students or staff faced with the problem of writing packages which must be run under the Educational Timesharing System - ETS. This manual assumes a reasonable idea of ETS and a good familiarity with the PAL assembly language. It is designed more as a reference manual than guide to ETS.

Any suggestions for improvement will be sincerely appreciated.

Table of Contents

List of Figures

List of Examples

ETS System User's Guide

I. Introduction

A. Welcome to the ETS System User's group. In this paper we hope you will find anything you need to know about programming under ETS. However, before we begin to explain the art of programming for ETS we would like to bring up a few items of ETS' philosophy.

1. ETS is designed for use by students who are not acquainted with the use of a computer. Consequently, we must assume that every student input is possibly in error and check it accordingly.
2. Again because we assume that students know nothing about the system, we want to keep the required responses very short and as clear as possible.
3. In the interest of compatibility with DOS we will use names and input responses as much as possible like those used by DOS.

B. The system code which you write for ETS will be run without the use of an interpreter. Since the PDP-11 has no protection it is necessary that all system code be bug free.

C. Unlike operating systems on machines which have hardware memory protection and hardware relocation, ETS is a "handshake" monitor which runs in conjunction with the system programs. This means that system programs must to some extent be conscious of the ETS monitor and its needs. Similarly, the ETS monitor will provide services which will make it easier for the system program to run without the presence of hardware memory relocation.

II. Basic Job Structure

A. Dynamic Relocatability

All jobs in ETS are considered as blocks of dynamically relocatable code (see section III.A) with an addressing space, A, $0 \leq A \leq 4000_8 n$ where n is the number of blocks of core assigned to the job.

B. Minimum Stack

All jobs will have a minimum stack size of 400_8 bytes beginning at user location 0 (see section III.B).

C. Control and Communication Structure

In line with the "handshake" principle of operation, ETS permits jobs to examine the control variables which it maintains, in order to know the state of the job at any given time. While the user may examine the control variables at will, he must not change them. ETS also provides some communication variables through which the user may communicate his wishes to ETS and vice versa. The first seven items in the following list are all "read only" control variables. The last three are communication variables.

1. JOB is a variable of length 1 byte which contains a user's job index or job number (note that all job numbers in ETS are even numbers).
2. JOBDA is a one word pointer which points to the beginning of a 16 word job control block called a job data block (JDB). The job data block is the basic structure which ETS uses to define the state of a user job at any given time (see figure II.1).
3. The first word in the JDB points to the input-output control block (IOB). The IOB is an eight word block which contains pointers to control blocks for each of the files or devices currently in use by a given user. Each of the eight possible pointers is referred to as a channel. Any channel which contains

JDIOB - Pointer to I/O block		0
JDFLG - Job status flags		2
JDIOST - I/O error status		4
JDSP - Relative user stack		6
JDCPU - CPU time used		10
JDKCT - KILO-CORE-ticks used		12
JDDEV - Device time used		14
JDSIZ1 - Next size in K	JDSIZ0 - Present size in K	16
JDUFD - Start block of job's UFD		20
JDPPN - Proj-Prog number		22
JDSGNW - Unused		24
JDSGMX - Unused		26
JDUFLG - User comm. flags		30
JDNAME - User Prog Identifier		32
JDANCR - Tie User Job to Specific Core Location		34
Unused		36

Figure II.1 Job Data Block

a non-zero pointer is currently in use; thus, a user may check for open files or devices by interrogating the channels and checking for non-zero entries (see example II.1). Note that all channels are referred to in ETS as channel indexes which are even numbers and may be used as an index. Thus, channel number one has a channel index of two.

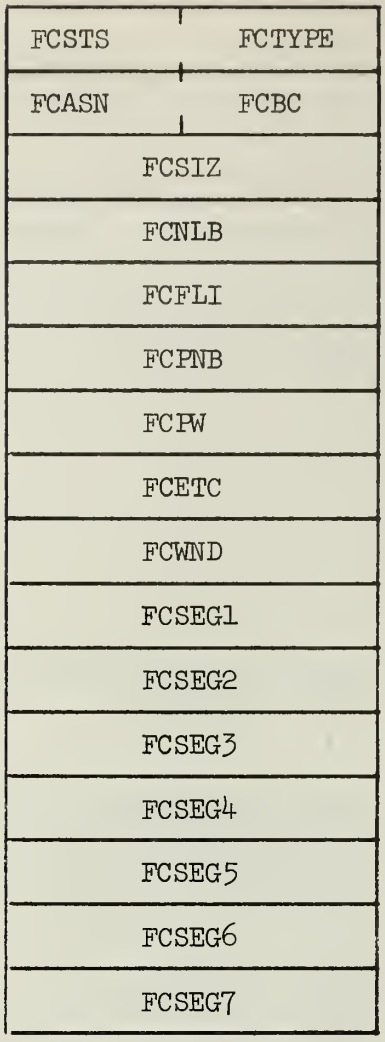
4. FCB/DDB

Each open file is controlled by a 16 word block called a file control block (FCB - see figure II.2). A file control block is pointed to by an entry in the IOB and contains sufficient information to allow the system to transfer data to/from a file without excessive disk accesses. Associated with each device is a device data block (DDB - see figure II.3) which allows the system to control the physical device. DDB's are created at system initialization and remain uniquely associated with the device from then on. An FCB is created only when a file is opened and is destroyed again when the file is closed. There may be more than one file control block for a given file assuming that more than one user has opened a particular file. However, there can be only one device data block for a device at any given time. The first byte of an FCB/DDB contains a handler index which may be examined to find the device associated with the block (see example II.2).

5. The first word of the IOB always points to the DDB for the user's job teletype. Thus, channel 0 is designed to be the control teletype for the user job. Note that channel 0 can neither be opened or closed by the user.

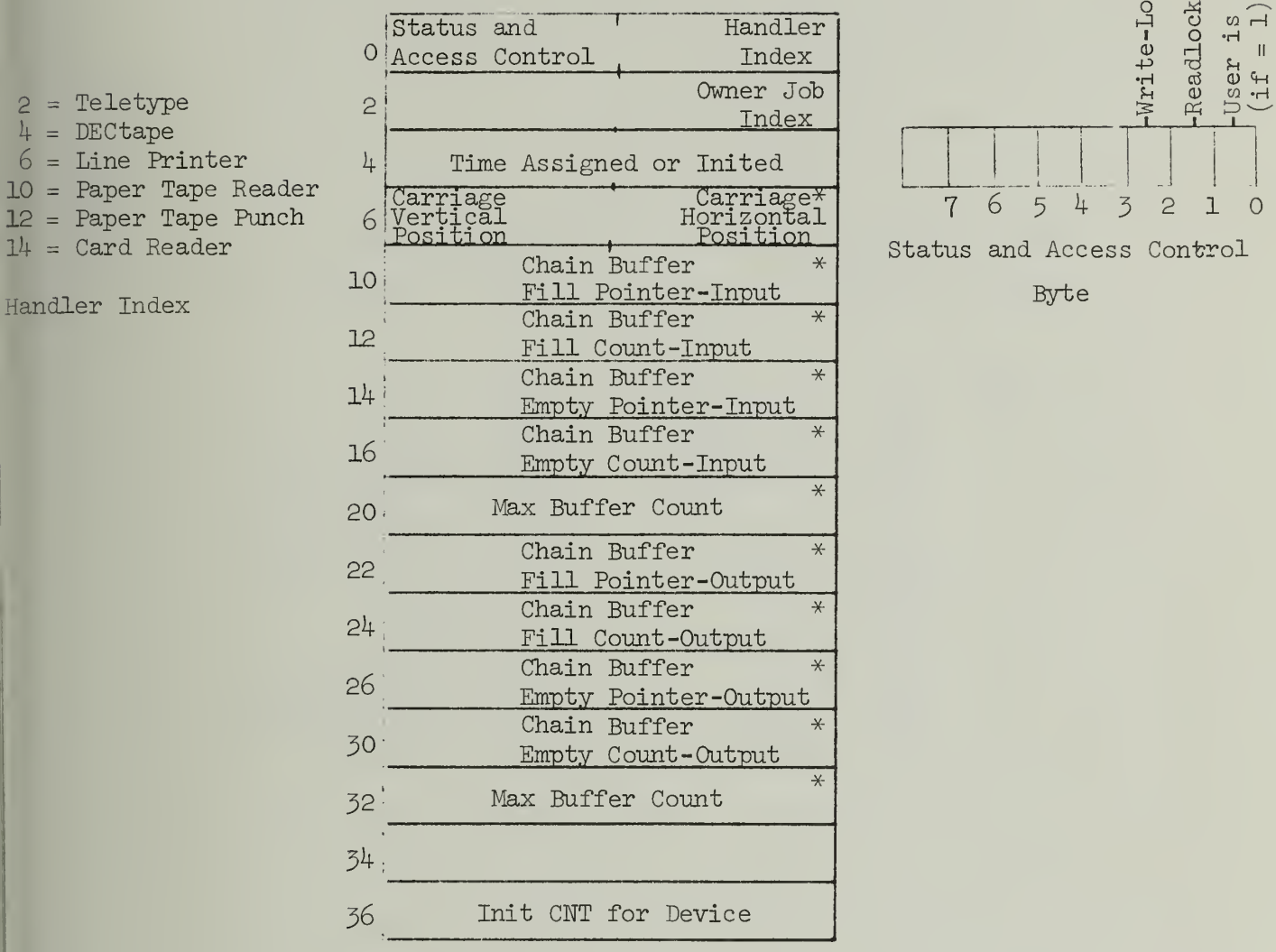
```
        JOBDA    = xxx          ;System variable containing pointer
            .                   ;to job data block
            .
            .
BEGIN:  MOV      @#JOBDA,R0     ;Pick up pointer to JDB--
                                ;use absolute addressing to
                                ;system variables
        MOV      (R0),R0        ;Point to IOB--address is multiple of 40(8)
        CLR      R1
   B1:  TST      (R0)+          ;Is this channel open?
        BEQ      B2             ;No, don't count it
        INC      R1             ;Yes, count it
   B2:  BIT      #17,R0         ;Have we checked 8 channels?
        BNE      B1             ;No, keep trying
            .
            .
            .
```

Example II.1 A Routine to Count the Number of Active Files/Devices




```
FCTYPE   =   0       ;I/O handler index - disk = 0
FCSTS    =   1       ;Status bits for file
                     ;Bit 1 = 1  User is not owner of file
                     ;Bit 2 = 1  User may not read file
                     ;Bit 3 = 1  User may not write file
FCBC     =   2       ;Block count for current transfer
                     ;(He can only get 1 block at a time
                     ;unless he does a direct access read/write)
FCASN    =   3       ;Location of file name block in UFD (word offset)
FCSIZ    =   4       ;# of segments in file
FCNLB    =   6       ;Logical block number of next block to read/write
FCFLI    =  10       ;First logical block in window
FCPNB    =  12       ;Disk physical segment containing name block
FCPW     =  14       ;Disk physical segment containing current window
FCETC    =  16       ;Miscellaneous word.  Users doing a direct
                     ;read/write put current memory address here
FCWND    =  20       ;UFD address of next retrieval window
FCSEG1   =  22       ;Physical segment corresponding to FCFLI
FCSEGi   =  20+2i    ;Physical segment corresponding to FCFLI+i
```

Figure II.2 File Control Block



*Indicates Device Dependent Assignment

Figure II.3 Teletype Device Data Block

```
        JOBDA   xxx             ;Pointer to JDB
CHNNDX: .WORD   x               ;Channel index set by user
          .
          .
          .
BEGIN:  MOV     @#JOBDA,R0      ;Point to JDB
        MOV     (R0),R0         ;Point to IOB
        ADD     CHNNDX,R0       ;Point to channel desired
        MOVB    @(R0)+,R0       ;Pick up handler index--
                                ;use sign extension
        ADD     R0,PC           ;Branch to desired handler
        BR      DISK            ;0 => disk (was an FCB)
        BR      TTY             ;2 => teletype
        BR      DTA             ;4 => DECtape
        BR      LPT             ;6 => line printer
        BR      PTR             ;10 => paper tape reader
        BR      PTP             ;12 => paper tape punch
        BR      CDR             ;14 => card reader
          .
          .
          .
```

Example II.2 Routine to Extract Device Type
and Branch to Special Routine

6. JBSTAT/JBWAIT

In order to implement timesharing it is not sufficient to be able to start, stop and switch jobs. Jobs will often generate conditions which require them to wait for some later event (e.g., completion of a line of TTY input). It is very inefficient to keep restarting a job to allow it to check for the occurrence of a given event. A better solution is to set some type of flag and let the system watch for the event, then wake the job when the event has occurred. In ETS there are a number of specific events on which a job might wait (in particular, each device has a unique event associated with it as does the file processor and the timer service). When a job is created it is assigned 2 status words (JBSTAT/JBWAIT) to indicate blockages that may occur. When a job is running both JBSTAT=1 and JBWAIT=1. When a job requests a service for which it must be blocked, JBWAIT is cleared and the bit associated with that event is set by the system event routine in JBSTAT. The condition that a job be able to run is then JBSTAT $\wedge$ JBWAIT $\neq$ 0. The system then can check JBSTAT $\wedge$ JBWAIT and schedule a job to run only when it is not blocked. Note that a job will then run if any one of a number of blockages are relieved; it will not wait until they are all relieved.

7. IOSTS is a one word pointer to a job's I/O status which is kept in the second word in the job data block. This word contains the result of the last I/O or file operation performed by the job (see figure II.4 for a list of error codes).

8. REGREL identifies six bytes which provide the register relocation controls (see section III.B).

ERROR NAME	ERROR NUMBER	MEANING
BADCMD	1	Invalid command
BADDIR	2	Directory screwed up
BADNAM	4	Badly formed file name
NOTFND	6	File not found
INUSE	10	File of same name is in use
NOROOM	12	Directory is full or disk is full
NOSUCH	14	File or UFD does not exist
NOTCLS	16	Can't open already open channel
NOTAVL	20	Device not available to user
NOTOPN	22	Trying to operate on unopened channel
PRVIOL	24	Protection violation
EOF	26	End of file on read/write
ABORT	30	Operation aborted
DATERR	32	Data error on device
HNGDEV	34	Hung device for job
NOLOAD	35	No load address specified on object file
CKSMER	36	Checksum error in loader
OUTRNG	37	Address requested out of user area

Figure II.4 FIP or I/O Error Codes

9. COMWRD identifies four words which are reserved for communications between the system and the user job. These words presently have no designated meaning in the system and may be used at will by the user job.

10. USRFLG is a communication word in which the system sets flags to indicate that the user has typed a control character which should be meaningful to the user job (see section VII.B).

III. Code Description

A. Dynamic Relocatability

Since the PDP-11 has no relocation hardware and ETS intends to be a timesharing monitor, the code in ETS must be relocatable. This means that ETS must be able to stop the execution of the user job at any time in order to swap it out to allow another process to run. When ETS determines a job is to be swapped in again from disk, it does not necessarily return the job to the same position in memory. A job does, however, always remain in a single contiguous block so relative distances within the job itself do not change.

1. Dynamic relocatability requires the use of strictly relative addressing modes with the user's program variables.
2. When referencing system variables such as REGREL or USRFLG the user must use strictly absolute addressing (i.e., INCB @#REGREL).
3. With the exceptions noted here, the user must not develop absolute addresses pointing anywhere in the user area.
4. ETS does provide limited ability to develop absolute pointers into the user area. The user job may inform ETS of its intentions to place an absolute address in a register or it may place absolute addresses on the stack or it may place them in reserved pointers in system area (RELPTR).

B. Register Relocatability

1. When a user wishes to place an absolute pointer in a register he may tell ETS of his intentions by putting a non-zero quantity in the register relocation byte (REGREL) corresponding to that register. Six REGREL bytes are assigned to the registers R0 to

R5 in order beginning with R0. Any non-zero quantity in a register relocation byte indicates that a user may have absolute pointers in the corresponding register (see example III.1).

2. Conditional Relocation

When ETS finds a non-zero REGREL byte, it assumes that the corresponding register may contain an absolute pointer into the user area. However, should ETS need to relocate that register, it will check the contents of the register, to see if it actually contains a value which would be a valid pointer into user area. If it does not contain a possible absolute address for the user (a value which lies between 30000_8 and 60000_8), ETS will not relocate the register.

3. Conditional relocation allows the user to conveniently construct absolute pointers in registers without having critical segments of code (code which must not be interrupted during execution). The following example is not a critical section because of conditional relocation:

```
INCB  @#REGREL+1     ;Mark R1 as absolute
MOV   #X-.-6, R1     ;Load offset to desired variable
ADD   PC, R1         ;Make it an absolute pointer
```

If the job is interrupted and swapped between the MOV and ADD instructions, it will try to relocate R1 but will not find a value in R1 which does not point to user core (X is a local variable) and will not change it. Without conditional relocation, the system would modify R1 which, at the time of the interrupt, contained a relative offset rather than an absolute pointer.

```
REGREL   =  xxx          ;Address of register relocation
   .                     ;Byte zero
   .
   .
   .

INCB     @#REGREL+3      ;Register 3 is to be relocated
INCB     @#REGREL+5      ;Register 5 is to be relocated
```

Example III.1 Marking Registers Which May Contain Absolute Pointers

4. The stack in ETS is a conditionally relocated area (see III.D), and consequently the user must be careful about saving registers on the stack. In general, registers which are relocatable may be placed on the stack and registers which contain arbitrary data should not be pushed on the stack.

C. COMWRD and RELPTR

1. There are four reserved words, beginning at COMWRD, in the system area which have been set aside as communication words between the system and the user. None of these four words are presently in use; hence, the user may store anything he wishes in this area.
2. There is an additional communication word by the name of USRFLG whose usage has already been defined (see section II.C.10).
3. There are seven words beginning at RELPTR which are kept as automatically relocated user offsets. Each of these seven words is changed by an amount equal to the distance which a user is moved anytime the user is swapped. Thus, the user may construct an offset to an absolute pointer which does not point directly into user core and still have the system relocate it each time he is moved. When a user job begins execution each of these seven "relative pointers" will point absolutely to the first location in the user area. Therefore, the user does not need to construct a program counter relative offset in these pointers. He may simply add the offset from the beginning of his own area to the present contents of the pointer and construct his absolute pointer (see example III.2).

D. Stack Conventions

1. The user stack is also treated as a conditionally relocated area. Each time a user's job is swapped out and brought in at a different

```
RELPTR   =   xxx          ;Address of relocating pointers
PTRONE   =   RELPTR
PTRTWO   =   RELPTR+2
     .
     .
     .
     .

ADD    #XRB,@#PTRONE      ;PTRONE now contains an absolute
     .                    ;pointer to XRB - the system initially loads
     .                    ;PTRONE with the user job origin
     .
     .

INCB  @#REGREL+1          ;Mark R1 as absolute
MOV   @#PTRONE,R1         ;Pick up pointer in R1
```

Example III.2 Establishing and Referencing Absolute Locations using RELPTR

location the system examines all words between the current top of stack and the stack base, and relocates any of those which point absolutely into the user area. Note that the JSR instruction and the processor trap instructions also cause the current value of the program counter to be pushed onto the stack. The trap instructions also cause the processor status word to be pushed onto the stack. Since the stack is a conditionally relocated area the system will modify the program counter values when necessary; however, the processor status word is not a possible pointer into the user area and the system will not change it.

2. When a user job begins execution it has the minimal stack size of 400_8 bytes. The user may increase this if need be, however, he may not decrease it. When calculating the amount of stack necessary for his job, the user should determine the maximum number of bytes that he might need at any time and then add 200_8 bytes to that for system use and for a protection buffer. Notice that there is no form of protection for user stack overflow. Thus, care must be taken to insure that no user stack overflows occur.

3. Should a user need to increase his stack size, he simply loads his desired stack size into a system variable called STKBAS. (He cannot just add an increment.) The user can use this facility should he need to create a table of pointers into his user area. By telling the system that he has increased his stack size but not actually doing so, the user can create a table which will be conditionally relocated. If he loads the absolute values of the pointers he wants into this table, he has a table of relocated values which may be used as a dispatch table (see example III.3).

```
        STKBAS  =  xxx                          ;Address of user stack size variable
        JOBORG  =  xxx                          ;System pointer to beginning of user job
        TBLSIZ  =  16                           ;Our jump table will have 16 entries
             .
             .
             .
             .
        MOV  #TBLSIZ+TBLSIZ+400,@#STKBAS        ;Set our new "stack" size
        MOV  #TBLSIZ, R0                        ;Set up TBLSIZ entries
        ADD  #401, @#REGREL+2                   ;Mark R2 & R3 as absolute address regs.
        MOV  @#JOBORG, R2                       ;Pick up start of our job
        MOV  R2, R3                             ;and replicate
        ADD  #400, R2                           ;Point just above stack base
B1:     MOV  R3, (R2)+                          ;Fill table with our own job origin
        DEC  R0                                 ;Count this entry
        BGT  B1                                 ;And get them all
        ADD  #400, R3                           ;Now use R3 to finish construction
                                                ;of table
        ADD  #SUB1, (R3)+                       ;Abs. jump address to SUB1
        ADD  #SUB2, (R3)+                       ;Abs. jump address to SUB2
             .
             .
             .
        ADD  #SUB16 (R3)+                       ;Abs. jump address to SUB16
             .
             .
```

Example III.3 Construction of a Dispatch Table Containing Absolute Addresses at Base of Stack

4. The user should be careful about using instructions of the form JSR R5,X where R5 may not be a relocatable register since the system pushes the contents of the register onto the stack. If R5 is an absolute piece of data which looks like a pointer into the user area the system will change the data in R5 if the user is swapped. A useful technique for the user who needs a stack which may contain arbitrary data is to create a separate data stack below the user's real stack and use another register such as R5 as a stack pointer. Note that R5 must be indicated as a relocatable register. Also note that R5 does not behave exactly like SP in doing byte operations in auto increment and auto decrement mode.

E. Processor Status Word Manipulation

1. In general the user should avoid modifying the processor priority. However, he may examine and modify the condition codes in the processor status word at will. There may be times when it is necessary for the user to execute short sections of code without interruption. This may be done by running at system priority seven. The user may either set the priority to seven in the processor status word himself or he may use system routines to do so. The system provides two routines: IOF and ION to raise the priority to seven and lower it to its original priority respectively. These routines are called by JSR PC, @#IOF and JSR PC, @#ION respectively. These two routines must be executed as a pair and the user must take caution to avoid executing two consecutive IOF calls. (This will cause the system to be locked at priority seven.)

2. In general the user does not need to run at priority seven; however, he may wish to execute sections of code without being swapped out. He can do this by running at system priority three. This allows the system to perform normal I/O but will not allow the system to switch to another job. The user need not worry about informing the system about absolute pointers which exist only during PR3 execution, as the user cannot be swapped.

   a. Like the use of priority seven, the use of priority three should be restricted to short segments of code. Normally 50 to 100 instructions or less.

   b. Should the user need to execute large segments of code at priority three, he must inform the system when it is alright for him to be swapped. In general, if more than 1-2% of a user's code is to be executed at priority three he should attempt to inform the system each time he returns to priority 0 that he is available for swapping. This is done by simply executing an EMT .BREAK. If the system has another job requesting the use of the CPU it will switch to that job; if not, it will return immediately to the user job.

   c. Even though the user does not intend to execute his code at priority 3, he may cause the same effect if he does a great deal of disk I/O. When the user executes a disk I/O transfer, he is locked into his core. While the processor may switch tasks if there is another runnable task which is core resident, the job may not be swapped out. Consequently, programs doing many disk transfers should execute an EMT .BREAK periodically (~3-5 transfers) to allow the system to swap him out.

F. System Buffer Allocation

1. Some other operations require the use of buffers which remain in core even though the user may be swapped out. For this purpose the system maintains a pool of approximately 100 buffers which may be allocated to the user for short periods of time. All system buffers are 20_8 words in length and are allocated by calls to the system buffer allocation routine (see section IV.H). All buffers are allocated on 40_8 address boundaries.
2. When the system allocates a buffer to the user job, it loses all record of that buffer and depends on the user to return it when he has completed his work with it. Since it is possible for the student to cause the user job to be killed by typing a CNTL C (↑C), users must be careful about the allocation and deallocation of system buffers. Special care should be taken to see that user jobs do not have system buffers allocated during I/O operations.

G. Anchor Mode

If for some reason it is impossible or impractical to write code which is dynamically relocatable, a user may specify that his code be run in 'Anchor Mode' which means that the system will swap him back to the same place each time he is brought into core.

In order to run in Anchor Mode, the user must first execute an EMT .ANCHR (see section VI.A) and then cause himself to be swapped out so the system can position him (see example III.4).

When a user no longer requires that he be anchored he should execute a .HOIST to release himself. Since the use of anchor mode works quite a hardship on the system, it should be avoided whenever possible.

```
;This routine will anchor a 2K program at 30000.
;Since the program may have already been loaded
;elsewhere it is necessary to force the program
;to be swapped to the proper location.  This can
;be done by increasing the size of the program
;and then decreasing it again.

        .ANCHR = 104112
        .CORE  = 104056
BEGIN:  MOV #30000,R1     ;Anchor at 30000
        .ANCHR            ;
        MOV #2,R0         ;We should be at size two.
        .CORE             ;Make sure we are.
        MOV #3,R0         ;Now increase to size three
        .CORE             ;forcing a swap.
        MOV #2,R0         ;Now decrease to size
        .CORE             ;two again.
        .
        .
        .
        .
```

Example III.4 Anchoring a Program

H. Named Code

In order to allow the system to provide special assistance to selected programs, it accepts a name for a program which it may use to identify the type of job in progress. When a program begins execution, it may execute an EMT .NAME (see section VI.). When the program executes a .EXIT, the name will be removed. This facility is useful when the system must service a user who may have terminated execution. By checking the existence of the proper name, the system can avoid damaging or unnecessarily servicing users.

It is necessary that any new names be unique among the names known to the system. However, this is not a hardship since names are useful only when special service routines are included in the system and the names may be selected when the routines are written. Currently only the name DTADO=1 is in use and is reserved for routines using DECtape service.

IV. File Handling

A. File Operation

All operations which concern the handling of data which is stored on the disk are file operations. In addition, ETS supports a number of non-resident functions which are also treated as file operations though they may actually have little or nothing to do with the filing system. The ETS filing system consists of a core-resident and a disk-resident portion. On disk are stored a number of special functions which may be called into core when needed. All file operations are carried out by allocating a system buffer, called a FIRQB, and placing the number of the requested function as well as any necessary parameters in this buffer. The user then places the address of the buffer in R4 and executes an EMT CALFIP.

B. File Request Queue Block (FIRQB)

1. When performing file operations, the user must request a system buffer and load it with the proper parameters (see figure IV.1). This FIRQB requires the user's job number, FQJOB, and a function code, FQFUN (the system provides a routine to allocate the buffer and load these two words - see section IX.C). In addition, the user must supply any special parameters relative to the process he is using (see section IV.C). Generally, the system will return the FIRQB to the user on completion of an operation. The user may get any results which were returned in the FIRQB and should then return the FIRQB to the system. See example IV.1.

C. Available File Operations

1. EXTFQ is a routine to extend the length of a disk file. The user loads in FQFIL the channel index of the file which he wishes extended. In FQSIZ he supplies the number of 256_{10} word blocks to extend the file. If the disk is full, or no disk blocks remain,

Name	Value	Length (bytes)	Purpose
FQQQ	0	2	Queue word for chaining requests
FQJOB	2	1	Job # issuing request (*2)
FQFUN	3	1	Function requested
FQFIL	4	1	Channel index
FQERNO	4	2	Error or message # requested from file
FQSIZ	5	1	Size of file to be created or number of segments to extend file
FQPROT	5	1	New protection code
FQPPN1	6	2	Project-programmer number of UFD desired
FQNAM1	10	6	FILENAME.EXT in RAD50
FQPPN2	16	2	Project-programmer number of second UFD
FQNAM2	20	6	FILENAME.EXT of second file
FQBLCK	22	2	Number of absolute loader blocks loaded
FQCKSM	24	2	Number of check sum errors encountered.
FQSTRT	26	2	Offset from beginning of user area to "relative 0" of file to be loaded
FQDEV	30	2	RAD50 device name
FQDEVN	32	2	RAD50 device number
FQLEN	34	2	Length of string to be parsed
FQPTR	36	2	Absolute pointer to buffer containing source string to be parsed--string is in chained set of system buffers for SWIFQ

(Note that FQJOB and FQFUN must always be supplied--all other parameters need to be supplied when actually needed.)

Figure IV.1 FIRQB Structure

```
            BUFFER = 4
            GETSML = 200
            RETSML = 0
            FQJOB = 2
            FQFUN = 3
            FQFIL = 4
            FQLEN = 34
            FQPTR = 36
            OPNFQ = 10
            CSIFQ = 46
            CALFIP = 104050
            REGREL = xx
            JOBORG = xx
            IOSTS = xx
            JOB = xx
                .
                .
                .
STRING:     .ASCII /FILEAB.EXT/          ;Name of file to be opened
STREND:     .BYTE 0
CHNNDX:     .BYTE 4                      ;Channel index - even #
            .EVEN
                .
                .
                .
            INCB    @#REGREL+0           ;R0 is relocatable
            BUFFER                       ;Request a 16 word buffer
            GETSML                       ;Pointed to by R4
            MOV     #STRING,R0           ;Point to file name relative offset
            ADD     @#JOBORG,R0          ;Make an absolute pointer
            MOVB    (R0)+,(R4)+          ;Store string in buffer
            BNE     .-2                  ;a byte at a time
            BIC     #37,R4               ;Make R4 point to beginning of buffer
            MOV     R4,-(SP)             ;Save R4
            BUFFER                       ;Get a FIRQB
            GETSML                       ;
            MOV     (SP)+,FQPTR(R4)      ;Store pointer to file string in FIRQB
            MOV     #STREND-STRING,FQLEN(R4) ;Store length of file string
            MOVB    #CSIFQ,FQFUN(R4)     ;Load function desired
            MOVB    @#JOB,FQJOB(R4)      ;Load our job index
            CALFIP                       ;Call the file processor
            MOV     @#IOSTS,R1           ;Get pointer to our I/O status
            MOV     FQPTR(R4),-(SP)      ;Save pointer to file string buffer
                                         ;for later return
            TST     (R1),                ;Operation successful?
            BNE     ERROR                ;No
            MOVB    CHNNDX,FQFIL(R4)     ;Yes - load channel index on which
                                         ;file is to be opened
            MOVB    #OPNFQ,FQFUN(R4)     ;Set open function
            CALFIP                       ;Call file processor again
            TST     (R1)                 ;Successful
            BEQ     OKAY                 ;Yes - FIRQB was not returned
ERROR:      BUFFER                       ;Failed - return FIRQB
            RETSML
OKAY:       MOV     (SP)+,R4             ;Get pointer to file string buffer
            BUFFER                       ;and return it
            RETSML
```

Example IV.1 Routine to Pack a File String into a FIRQB and Open the File on Channel 2

an error code 12 (NOROOM) is returned in IOSTS.

2. DELFQ is a routine to delete an open file. The user supplies the channel index of the file in FQFIL. The user must be write enabled on the file in order to delete it.

3. CLSFQ is a routine to close an open file. The user supplies in FQFIL the channel index of the file he desires closed.

4. OPENFQ is a routine to open a file. The user must supply in FQFIL a presently unused channel index. In FQNAM1 he must supply a three word file name which is packed RAD50. If the user wishes to open a file which belongs to a different project-programmer number he supplies the project-programmer which owns the file in FQPPN1. In order to open a file a user must be either read enabled, write enabled or the owner of the file. If the open operation is successful the FIRQB will not be returned to the user.

5. CREFQ is a routine to create a file. The user supplies in FQNAM1 a file name of the form FILNAM.EXT packed RAD50. In FQSIZ he specifies the length of the file which he wants created. A length of zero or one is created as a file of length one block. If a file of this name already exists in the user's UFD and the user is write enabled on that file, the system will delete that file and recreate it at its new length.

6. RENFQ is a routine to rename an existing file. The user supplies in FQNAM1 the name of the file which he wishes to rename. He supplies in FQNAM2 the new name of the file. The user must be write enabled on a file before he can change its name. If the file does not belong to the user's project-programmer number he specifies a project-programmer number of the file in both FQPPN1 and FQPPN2.

7. BYEFQ is a routine to log out a user from the system. No parameters other than the user's job number need be supplied. Before logging out a user, the system will close all his open files. BYEFQ does not return!!

8. DIRFQ is a routine to return directory information on a file entry. The user supplies in FQPPN1 a project-programmer number of the directory with which he is concerned. All files in a given directory (UFD) are numbered sequentially beginning with zero. The user supplies in FQFIL the number (this is not a channel index) of the file on which he wishes information. If there exists a file corresponding to the number specified, the system will return the name of the file in the three words beginning at FQNAM1. The system will return the protection code associated with the file in FQPPN2. It will return the date of the last access at FQNAM2, which will be immediately followed by the size of the file (the number of 256_{10} word blocks), followed by the creation date of the file, followed by the time the file was created.

9. PROFQ is a routine to change the protection code associated with the file. The user supplies in FQFIL the channel index on which the file is open. He supplies in FQPROT the new protection code to be assigned to the file. The user may not change the protection of a file unless he is either write enabled or the owner of the file.

10. ERRFQ is a routine to return an error message from the system error file. The user supplies in FQERNO the error number which he would like to have returned. On return the system will pack the error message into the FIRQB beginning at FQERNO.

11. LOAFQ is the system relocating loader. The user supplies in FQFIL the channel index of the file to be loaded. He supplies in FQSTRT an offset from the beginning of the user area to the zero of the load file (a zero here causes the file to be loaded with the load "0" corresponding to the beginning of the user area). On return the system supplies in FQBLCK the number of loader blocks loaded. This includes the six word start block. The system also returns in FQCKSM the number of checksum errors occurring during the load.

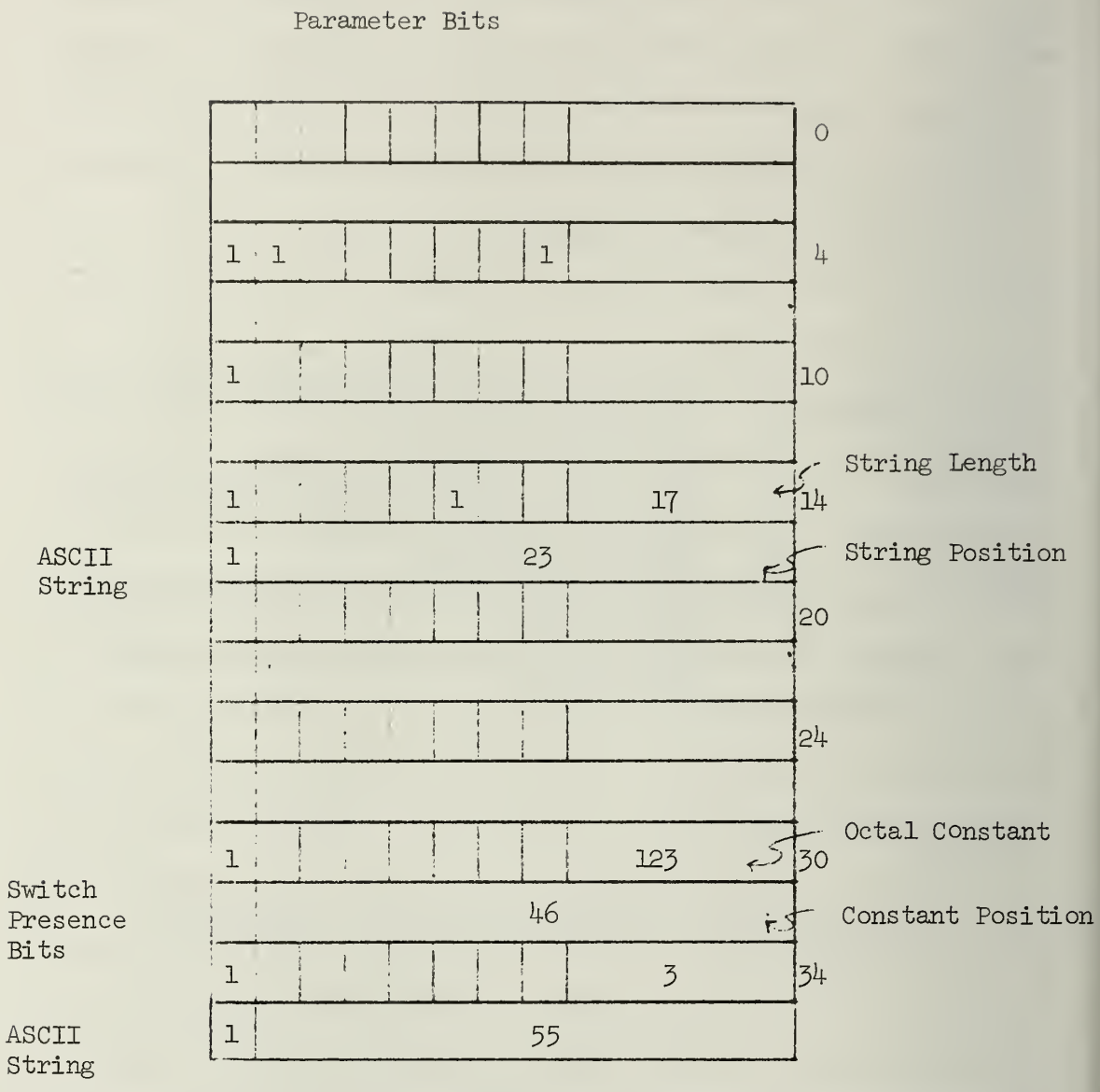
12. RSTFQ is a routine to close all I/O channels except channel zero. The user needs no additional parameters.

13. CSIFQ is a routine to parse DEV:FILE.EXT[X,X]. The user must allocate a system buffer and pack the string to be parsed into the buffer. He then places the address of this buffer at FQPTR and places the length of the string to be parsed at FQLEN. The system returns the RAD50 pack of any device name at FQDEV. The RAD50 pack of any device number is returned at FQDEVN. Independent of the presence or absence of a device name and device number, the system packs a string of up to six characters as a file name. They are packed RAD50 and placed beginning at FQNAM1. If there are less than six characters, the name is considered to be extended on the right with blanks to form a six character name. A file name may be followed by a three character extension to indicate its type. The extension is packed and stored at FQNAM1+4. It too is extended with blanks. Additionally, the user may specify a project-programmer number by enclosing it in square brackets ([4,5]). The two numbers are considered as octal and must be separated by a comma. They will be stored in the two bytes of FQPPN1--the first digit goes into the high order byte.

14. SWIFQ is a routine to parse a list of switches. The user must pack the list of switches into system buffers which may be chained together. He supplies a pointer to the first buffer at FQPTR. The first word of each buffer is then a pointer to the next buffer in the chain. The last buffer in the chain has a zero in its first word. The user supplies the length of the string in FQLEN. See figure IV.2 for a description of a possible switch list. On return, the system returns the FIRQB in addition to the first buffer in the chain. All other buffers are returned to the system buffer pool. The buffer pointed to by FQPTR will contain the parse information for the switch list. Each of the possible switches is associated with two words in the buffer (beginning with the third word of the buffer, in order, as given in <switch> in figure IV.2). In the low order byte of the first word of any switch which occurs is stored an octal constant or the length of an arbitrary string which was the last parameter to the switch. Bits 8 through 14 indicate the presence of the parameters numbered 1 through 7 (in order as given by <parm> in figure IV.2). Bit 15 of the first word is set to indicate that this particular switch occurred in the list of switches. The second word of the switch list is a pointer, relative to the beginning of the string, to the beginning of an ASCII string which was a parameter to that switch. In case of an error return all buffers except the FIRQB will be returned to the system. Figure IV.3 is a diagram of the buffer returned for a sample switch string.

```
<switch list>     ::=   <switch string>|<switch string>;<switch list>
<switch string>   ::=   <switch>|<switch>:<parm list>
<switch>          ::=   OP|CL|RE|PR|DE|S6|S7
<parm list>       ::=   <parm string>|<parm string>:<octal string>|
                        <parm string>:<ASCII string>
<parm string>     ::=   <parm>|<parm>:<parm string>
<parm>            ::=   CN|EE|P3|P4|P5|P6|P7
<octal string>    ::=   <string of ASCII characters which represent
                        octal digits>
<ASCII string>    ::=   <string of ASCII characters which cannot be an
                        octal string>
```

Figure IV.2 The BNF Definition of a Switch List



<switch list> = QP:CN:P6;CL;RE:P3:FILABC.EXT[1,1];S6:123;S7:129*

buffer position 4 10 14 30 34

*129 is not a valid octal constant, hence it is returned as a string length

Figure IV.3 The Buffer Returned on Parsing a Switch List

15. FNDFQ is a routine to check if a given file or device is open and if so to return the channel index on which it is open. It is called by placing the name of the file at FQNAM1 or the device name at FQDEV with any device number at FQDEVN. On return FQFIL will contain the channel index of the channel on which the given file or device is open. If the file or device is not open it will return an error 14 (NOSUCH). If the file or device does not exist it returns an error 22 (NOTOPN).

D. Error Return

When a specified file request fails to function properly it will return in the second word of the user's job data block an error code. The system variable IOSTS contains a pointer to this I/O status word. Figure II.3 contains a list of possible error codes and an indication of their meaning.

V. Input-Output

A. General Principles

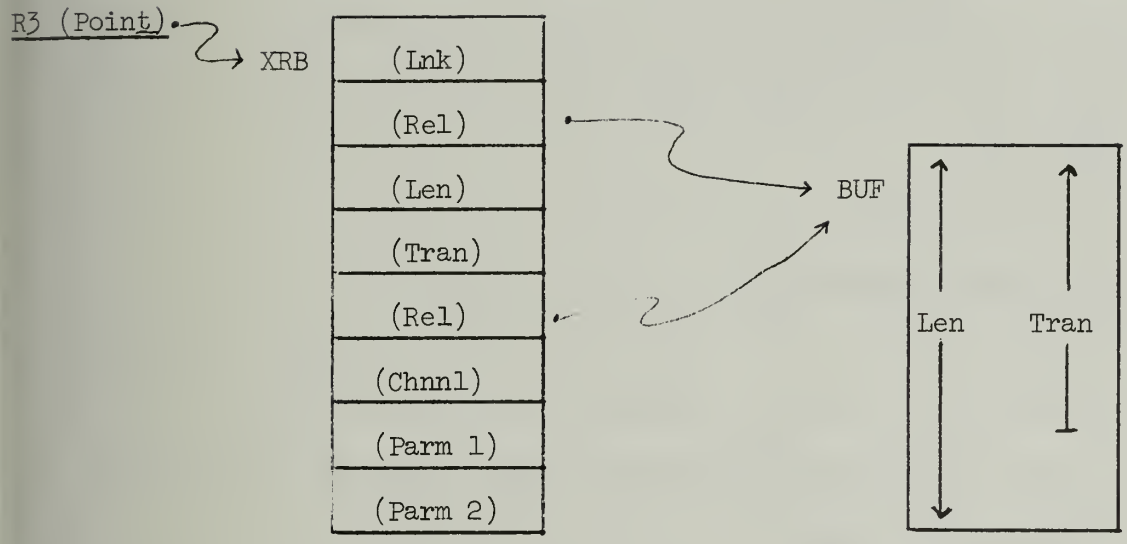
1. All device I/O in ETS is controlled by constructing a transfer control block (XRB) which contains all the necessary parameters for the desired operations (see figure V.1). There are four basic items which must be supplied:

   a. Buffer Address. The user places in the XRB in two places (XRPTR and XRLOC) the address of the buffer to be used in the transfer. The address is in the form of the relative distance from the buffer to the XRB (#XRB-BUF). The buffer must be in the user area proper.

   b. Transfer Length. On a read operation, the user places a zero in XRBC and the maximum length of transfer (in bytes) in XRLEN. On return, the system will put the number of bytes actually transferred in XRBC. On a write operation, the user places the actual write count in XRBC.

   c. Channel. All read/write operations must take place to a file/device which the user has successfully opened (channel zero is automatically open). The user places the channel index in XRCI. Before allowing the operations to proceed, the system checks the protection associated with a channel and aborts the operation if necessary.

   d. Parameters. Reading from the card reader requires that the user supply parameters to allow the service routine to perform the proper character translation. Parameters are supplies in XRPAR and XRPAR2 (see V.B.3).



Point	=	Relative distance from start of job to XRB
Lnk	=	Monitor link word
Rel	=	Relative distance (XRB-BUF) from XRB to I/O buffer
Len	=	Maximum length of I/O buffer
Tran	=	Actual transfer count
Chnnl	=	I/O channel for operation
Parm 1	=	Parameter word
Parm 2	=	Parameter word

Figure V.1 Transfer Control Block Structure

2. Once the user has set up his XRB, he places its relative address in R3 (distance from user "0" to the XRB) and executes EMT 52 (.READ) or EMT 54 (.WRITE).

3. When doing I/O to a device other than disk, it is possible that all data is not immediately available; hence, the system will swap the user out of core to await physical I/O. This is not true for disk operations.

B. I/O to a Serial Device

1. Line - Block Oriented Read

Normally, on read operations, the system will attempt to supply the maximum possible number of characters, with two exceptions.

a. Any read operation is terminated when an EOF is encountered on the source device.

b. Line oriented devices will terminate a read after transferring a line delimiter (provided the delimiter is encountered before the maximum transfer is completed.

2. Papertape Reader - Papertape Punch

The papertape reader and papertape punch do strictly block oriented transfers of binary data. All operations will terminate with either the maximum transfer length fulfilled or an EOF or EOM (End of Medium). Since leader and trailer tape look like perfectly valid binary data, the user must be careful when he wants to avoid processing undesirable 'nulls'.

3. Card Reader

There are two types of transfers from the card reader:

a. ASCII Source Data. The user may specify parameters to the card reader driver which causes the first n columns to be ignored, or he may ignore the last m columns, or he may request that trailing blanks be suppressed, or any combination of the three. Figure V.2 details the parameters and their meaning.

b. Binary Data. The card reader looks like the papertape reader where the top eight rows of the card correspond to the columns of a papertape (row 5 = bit 1). The card reader will try to fulfill the read request before terminating the transfer.

4. Line Printer

The line printer accepts block transfers on output. No parameters are necessary where outputting to the line printer; however, it does do some special processing on output characters in order to provide formatting ability.

a. A form-feed (ASCII code = 14) is echoed as a page eject.

b. A carriage return is ignored.

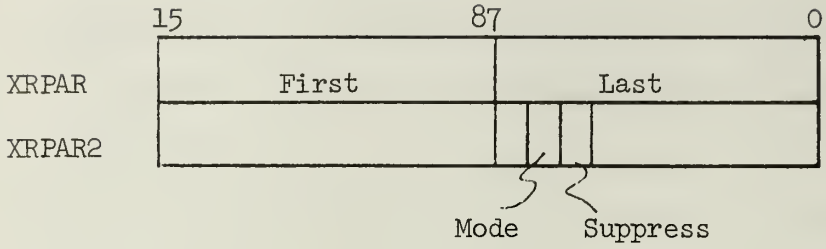
c. A line feed is echoed as a line feed and a carriage return (the hardware does this).

d. A tab (ASCII code = 11) prints as spaces up to the next eight space boundary (column eight is the first such boundary).

e. Rubouts (ASCII code = 177) are ignored.

f. Vertical tabs (ASCII code = 13) are ignored.

g. All other characters print as themselves if they exist on the print drum. If they do not exist on the drum, the hardware



First = logical first column of card -1
Last = logical last column on card
Mode = 0 =>EBCDIC
1 =>binary data in top eight rows
Suppress = 0 =>suppress trailing blanks if in EBCDIC mode
1 =>do not suppress trailing blanks

Figure V.2 XRB Parameters for Using the Card Reader

ignores then causing the counted line position to be one greater than the real line position. (Figure V.3 is a table of characters on the print drum.)

h. The maximum line length is 128 characters. Attempts to print more than this number of characters will result in the 129th character being deleted and replaced with a line feed.

5. Teletype

The teletype is a line oriented input device, but a block oriented output device. Because of the use of the teletype as a job control device, many input characters cause special processing to occur. With the exceptions in the special characters noted below, all characters cause their correct 7 bit ASCII code to be transferred to the user buffer. On output, characters for which there is no output character will echo as nulls.

a. A carriage return echoes as a carriage return, line feed and inserts both a carriage return and a line feed into the user's input buffer. The carriage return is a delimiting character and will cause the job to be started if it is blocked awaiting a delimiter. On output, a carriage return does not echo a line feed.

b. A line feed echoes as a carriage return, line feed but inserts only the line feed into the user's buffer. The line feed is a delimiter. On output, the line feed prints as a line feed only.

c. CNTL C echoes as a ↑C and causes the user job to be aborted. The teletype buffers are flushed. The user will be returned to monitor mode.

b7 b6 b5	0 0 0	0 0 1	0 1 1	1 0 0	1 0 1
b4 b3 b2 b1					
0 0 0 0		Space	∅	@	P
0 0 0 1		!	1	A	Q
0 0 1 0		"	2	B	R
0 0 1 1		#	3	C	S
0 1 0 0		$	4	D	T
0 1 0 1		%	5	E	U
0 1 1 0		&	6	F	V
0 1 1 1		'	7	G	W
1 0 0 0		(	8	H	X
1 0 0 1		)	9	I	Y
1 0 1 0	LF	*	:	J	Z
1 0 1 1		+	;	K	[
1 1 0 0	FF	,	<	L	◇
1 1 0 1		-	=	M	]
1 1 1 0		.	>	N	^
1 1 1 1		/	?	O	♡

Figure V.3 Line Printer Character Set

d. CNTL B echoes as a ↑B and causes the termination of all echoes until the next ↑B is typed. No character is inserted into the buffer.

e. CNTL O echoes as a ↑O and causes all programmed output to be discarded as well as any input which is not a delimiter. The effect of a ↑O is terminated when the delimiter is read out of the input buffer by the user program.

f. CNTL Q echoes as ↑QUIT and places a <CR> in the user buffer. Sets bit 15 of USRFLG.

g. CNTL U echoes as a ↑U and causes the user input buffer to be flushed (throwing away the current line of input).

h. CNTL Z echoes as a ↑Z and is given to the user as an EOF 'error'.

i. A CNTL V echoes as a ↑V and puts a <CR> in the user input buffer; however, it sets bit zero of USRFLG.

j. Altmode echoes as a $ and acts as a delimiter. No characters are placed in the input buffer.

k. Rubout operates in two modes:

   (1) In tape mode (which is set via an EMT .TTAPE) it is discarded. Tape mode is for use in reading from an ASR paper tape reader and can be terminated by an EMT .TTRST.

   (2) In normal mode, it causes the last character in the input buffer to be fetched and discarded. Characters discarded are echoed in reverse order between two back slashes.

l. A tab echoes as spaces up to the next eight character boundary. A tab (ASCII code = 11) is inserted into the user buffer.

m. A vertical tab echoes as eight line feeds and inserts a vertical tab (ASCII code = 13) into the buffer.

n. A form feed echoes as eight line feeds and places a form feed (ASCII code = 14) into the buffer.

Example V.1 gives a sample program to input a character string from a teletype. Example V.2 is a sample message output routine.

C. File I/O

File operations involve all transfers to and from the disk. All such transfers involve exactly 256_{10} words. Files in ETS are logically sequential, but they may physically reside anywhere on the disk. When a file is opened, a pointer (FCNLB) is set up to the first block (block zero) in the file. Each read/write operation causes this pointer to be advanced by one block. Before reading or writing to/from a file, the user must open the file (see section IV) on an available channel.

On the disk, an EOF is generated anytime the user attempts to operate on a disk block which is not a part of that file. If the operation was read, an EOF indicates the end of the data; however, if the operation was a write, the system provides a function which the user may call to extend the file (EXTFQ - section IV.C.1) so that the write operation may be repeated.

Although most normal disk usage is of a sequential nature, ETS provides facilities so that users may perform random block access operations. This requires that the user know the logical block number which he wishes to read/write.

To select a specific block, the user must place the block number desired in the first parameter word of the XRB and the channel on which the file is open at XRCI. He then places the relative address of the XRB in R3 and executes an EMT .FIND. If the request is for a non-existent block, the system will set bit 15 of the XRPAR on return. In such a case,

```
;Reader is a routine to read a string from the TTY into
;an 80 character input buffer
;Call
;          JSR  PC,READER
;          R0 relocatable

;Return
;          R0 points absolutely to input string
;          R3 contains byte count

READER:    MOV       #XRB,R3              ;Read a string

           .READ                          ;
           MOV       XRB+6,R3             ;Store byte count in R3

           MOV       #INBUF,R0            ;Get absolute address of INBUF

           ADD       @#JOBORG,R0          ;In 2 steps

           MOV       XRB+2,XRB+10         ;Restore XRB pointer

           CLR       XRB+6                ;Clear byte count

           RTS       PC                   ;Return

XRB:       .WORD     0,INBUF-XRB,120,0,INBUF-XRB,0


INBUF:     .WORD 0                        ;Input buffer
           .=.+120                        ;
```

Example V.1 A Routine to Read a String from the Teletype

```
;MESSAG is a routine to output a message on the TTY
;Call
;         JSR       R5,MESSAG
;         +         MSG
;  Where R5,R0 are relocatable
;         MSG is the assembly (0 relative) address of the message
;             to be output.  The first word contains the length
;             of the message (in bytes).
;Return
;         R0,R3 destroyed
;
;
MESSAG:  MOV       (R5)+,R0          ;Get pointer to the message
         ADD       R0,XRBW+2         ;Set XRPTR in write XRB

         ADD       R0,XRBW+10        ;Set XRLOC

         ADD       @#JOBORG,R0       ;Get absolute pointer to message

         MOV       (R0),XRBW+6       ;Load byte count

         MOV       #XRBW,R3          ;Get relative pointer to XRBW

         .WRITE                      ;Output message
         MOV       #2-XRBW,XRBW+2    ;Reset XRPTR

         MOV       #2-XRBW,XRBW+10   ;Reset XRLOC

         RTS       R5                ;Return

XRBW:    .WORD     0,2-XRBW,0,0,2-XRBW,0

MSG1:    MSG1ND-MSG1 - 1
         .ASCII /THIS IS THE FIRST MESSAGE/
MSG1ND:  .ASCII /./
         .EVEN

MSG2:    MSG2ND - MSG2 - 1
         .ASCII /ISN'T IT A BEAUTIFUL DAY/
MSG2ND:  .ASCII /?/
         .EVEN
```

Example V.2 A Routine to Output a Message on the Teletype

the present pointer to the file is unchanged. Example V.3 is a routine to read an arbitrary block of a disk file.

D. Errors

1. EOF. An end-of-file (Error #26) is returned when a user tries to read/write past the end of a file or from a device on which an end-of-file separator is encountered. All errors are returned in the second word of the job data block which is pointed to by the system variable IOSTS.
2. NOTOPN (Error #22) is returned when the user attempts to perform an I/O transfer to an unopened channel.
3. PRIVOL (Error #24) is returned when the user attempts to operate on a file/device which is protected from him. He may either be trying to access a file belonging to another UFD which is protected or doing something brilliant like reading from the line printer. A PRIVOL error is also returned when a user attempts to write to a file which another user currently has open.
4. NOTAVL (Error #20) is returned when the user attempts to get a device which is currently servicing another user.
5. DATERR (Error #32) is returned if there is a device malfunction or data error.
6. HNGDEV (Error #34) indicates that a device is not available to the system, e.g., it may be off-line.

```
XRB:     .WORD     0, BUF-XRB, 1000, 0, BUF-XRB
XRCI:    .WORD     0
XRPAR:   .WORD     0
BUF:     .=.+1000
          .
          .
          .

;This routine will read the disk block specified in
;the variable BLOCK from the file opened on the
;channel given in CHNNL.  Data will be placed in a buffer
;beginning at BUF.  In case of error, the return is to ERROR.

READER:  MOV       CHNNL, XRCI       ;Set channel index for operation
         MOV       BLOCK, XRPAR      ;This is the disk block
         MOV       #XRB, R3          ;Set relative offset from start of job to XRB
         .FIND                       ;Point to the block we want
         TST       XRPAR             ;Did we get it?
         BMI       ERROR             ;No--either not a valid file or no such block
         .READ                       ;Yes--read the block
         CLR       XRB+6             ;Reset byte count for later try
         MOV       XRB+2, XRB+10     ;Reset pointers for later use
         MOV       @#IOSTS, R1       ;Get our I/O status
         TST       (R1)              ;Any errors
         BNE       ERROR             ;Too bad
              .
              .
              .
```

Example V.3 Finding and Reading a Particular Disk Block

VI. Special EMT Functions

A. .ANCHR (EMT 112)

Anchor mode is used to allow programs to be attached to specific locations in core and to force the system to swap jobs into the same position in core each time (see section III.G).

In order to operate properly, after a user has been achored, he must force himself to be swapped out so that the system will have a chance to reposition him. This is best accomplished by increasing his core allocation. The user places the core address (relative to the beginning of the user area) in R1 and executes a .ANCHR. This will place the user in anchor mode, but will not reposition him if he is not already in the correct position (this is handled by the monitor swap routines); therefore, the user should increase his core size which will cause him to be swapped out and brought in at the correct place. It is the user's responsibility to see that his anchor request is a valid one (i.e., he does not want to anchor a 2K job in the top 1K of core). Also note that the necessity of increasing size prevents the user from anchoring a job to the highest core block. A user who wishes to anchor a 1K job may increase his size to 2 to cause the swap and then reduce his size to the desired 1K.

Again note that anchor mode should be avoided if at all possible.

B. .BREAK (EMT 66)

When users execute long sections of code which may cause the system to fail to act on clock interrupts (e.g., many disk accesses), the user should inform the system that he can be swapped if there is another job ready to run. In general, users should return to the system after 4 or 5 "consecutive" disk accesses. The user does this by simply executing

a .BREAK. No parameters are necessary. This will cause the system to save the job and run the scheduler. If no other jobs are pending, control will return immediately to the user.

C. .CORE (EMT 56)

This is a call to the system core allocation routine. It may be called any time the user needs additional core or has core to return to the system. The core allocater is called by executing a .CORE with the size which the user wishes to be in R0. On return, R0 will contain zero to indicate success. Requests for size zero are ignored. If the request is for more than the maximum possible size, the request is ignored. Size is counted in blocks of 4000_8 bytes.

When the user wishes to return core to the system, the core will be removed from the end (higher addresses) of his job. He will not be swapped, and will return to execution immediately.

When the user wishes to increase his size, the system will swap him out and the next time he runs, he will be brought in at his new size; thus, he must be swappable at the time of the EMT. Should he be locked in core at the time, results are unpredictable.

D. .DAR (EMT 62) and .DAW (EMT 64)

This is a routine to handle direct access disk requests. The user may read (write) to/from the disk without going through the normal read/write processor. To do this he must set up a pointer in R4 to the File Control Block of the file he wishes to use (the file must be open in order to operate on it). Before executing the .DAR (.DAW) the user must place the relative core address for the transfer at FCETC(=16) in the FCB; the number of blocks to transfer is placed at the byte FCBC(=2). FCNLB(=6) in the FCB points to the block number of the file for the operation

(the first block is block zero, each access causes FCNLB to be incremented by one). Before allowing the disk transfer to proceed, the system will still check to see that the user is permitted access to this file. (See example VI.1.)

E. .EXIT (EMT 76)

This routine causes the termination of the user job. No parameters are necessary. The system will simulate the typing of a ↑C by the user. A .EXIT causes all I/O buffers belonging to the user job to be flushed, thus, terminating any I/O in progress by the job. When a routine must wait on the completion of some I/O before exiting, it should execute a .STALL then a .EXIT.

F. .HOIST (EMT 114)

A user who has been executing in anchor mode and no longer needs this facility should execute an EMT .HOIST to terminate anchor mode. No parameters are necessary and no damage is done if the user is not really in anchor mode.

G. .LOCK (EMT 60) and .UNLCK (EMT 104)

.LOCK is used when a routine must not be swapped out of core for some reason (e.g., he wants to use the system loader). No parameters are necessary. The user should use .LOCK only when absolutely necessary and then must unlock himself as soon as possible. .UNLCK will free a user who is locked in core; it requires no parameters. Since the process jobs for lack of available core space, users are requested to execute a .BREAK after executing .UNLCK.

Execution of a .LOCK when a user is already locked in core or a .UNLCK when a user is not locked in core has no effect.

```
        IOSTS     = xx                ;System I/O status pointer
        JOBDA     = xx                ;System pointer to job data block
        FCETC     = 16                ;FCB offset to core address
        FCBC      = 2                 ;FCB offset to # blocks to transf
        CHNNL     = nn                ;Channel index on which file is o
        BLKS      = nn                ;Number of blocks to transfer
                  .
                  .
                  .
START:  MOV       @#JOBDA,R4          ;Get pointer to job data block
        MOV       (R4),R4             ;Get pointer to I/O block
        MOV       @CHNNL(R4),R4       ;Get pointer to FCB
        MOV       #CORBUF, FCETC(R4)  ;Load relative core address for
                                      ;transfer
        MOVB      #BLKS,FCBC(R4)      ;Load # of blocks to be transferr
        .DAR                          ;Read them in
        MOV       @#IOSTS,R5          ;Get pointer to I/O status
        TST       (R5)                ;Any errors
        BNE       ERROR               ;Yes
                                      ;No
                  .
                  .
                  .
CORBUF: .=.+      2000                ;Buffer for transfer
                                      ;Length = #BLKS*1000
```

Example VI.1 Use of Direct Access Read - DAR

H. .NAME (EMT 116)

A user may assign a name to a program by placing the name he wishes assigned in R1 and executing an EMT .NAME. When the user job terminated (either normally or via a system abort) the name will be removed. Names should only be used when informing the system that a user is executing a program which requires some special service on the part of the system.

I. .STALL (EMT 102)

It is sometimes a good idea for a routine to wait for a period of time before executing more code (e.g., it wishes to open the line printer which is currently in use). The routine may place a number of seconds to wait in R1 and execute a .STALL. This will cause the job to be blocked for the number of seconds specified.

The use of .STALL is much better than sitting in a null loop since the user is not using CPU time. .STALL should also be used when the user wants to close a device which may be in the process of physical output. Since the close routine will cause a device's output buffers to be flushed, it is possible that output will be lost; consequently, a job that wants to close a device to which he has just finished output should stall for a few seconds before executing the close. (Figure VI.1 gives suggested times for stalling for each device.)

J. .TTAPE (EMT 74) and .TTRST (EMT 72)

Sometimes it is necessary to read paper tape through the low speed reader on the ASR-33 teletype. When the user wishes to do this, he would usually like to suppress the echo and ignore any rubouts on the tape. He may also wish to do the same processing for the input from the teletype keyboard. This type of transfer is called tape mode and may be reached by executing a .TTAPE. No parameters are necessary. A .TTRST is used to get out of the mode. .TTRST can also be used to kill any

Card Reader	-	no stall necessary
Line Printer	-	3 secs.
Paper Tape Reader	-	no stall necessary
Paper Tape Punch	-	3 secs.
Keyboard	-	no stall necessary
Teletype Printer	-	1/10 sec/character /max number of output characters--to a maximum of 12 sec.

Figure VI.1 Suggested Stall Times for Various Devices

'no output flags' (e.g., ↑O typed on the keyboard - see section VIII.D). A .TTRST is useful when a program wishes to inform the user of some abnormal condition which may have arisen during a process which normally requires so much output to the teletype that a user may turn it off.

K. .WAIT (EMT 70)

A routine may sometimes desire to wait on the completion of some event (e.g., typing of a delimiter at a teletype). The system provides a .WAIT for this purpose. A .WAIT has the effect of causing the system to save the user's job and run the scheduler. If there is no other job ready to run, the effect of a .WAIT by itself is much like a .BREAK. However, the user may set a wait bit in the system control variable JBWAIT which identifies an event on which he wants to wait (see section II.C.6). If he then executes a .WAIT, the system will not restart him until the specified event has occurred (see example VI.2).

L. .FIND (EMT 100)

.FIND is used to position a disk file at a certain point (see section V.C).

M. .TREAD (EMT 122)

.TREAD is used to read from a teletype and if a response is not given in a specified time, the user program will be awakened. .TREAD is simply a .READ (pg 46) combined with a .STALL (pg 51). The user will be awakened when either is successfully completed. The user sets up the parameters as described for both a .READ and a .STALL and executes a .TREAD. When he is awakened, he may check the XRBC field of his XRB to determine whether he has received any input from the teletype. Note, that no input is transferred to the user program until a terminal character is typed on the teletype.

```
JOB             = xx                    ;Byte containing user job index

JSKEY           = 20000                 ;Teletype delimiter blockage bit
JBWAIT          = xx                    ;Wait table for user jobs
.
.
.
MOVB            @#JOB,R0                ;Get user job index - sign extend
MOV             #JSKEY,JBWAIT(R0)       ;Set bit to wait on appearance of
                                        ;delimiter or TTY - Clear any other
                                        ;unit conditions
BIC             #JSKEY,JBSTAT(R0)       ;Make sure we don't already have
                                        ;the bit set
.WAIT                                   ;Go into wait - we will be
                                        ;awakened when the user types a
                                        ;delimiter on his teletype
.
.
.
```

Example VI.2 Setting a Wait Condition

VII. Use of Interrupts and Special Functions

A. Interrupts

ETS provides the user with two interrupt vectors which he may use as his own. TRAP (vector = 34) and BPT (vector = 14) are maintained by ETS as relocated pointers into the user area. Each time a user routine is saved, the contents of these two vector locations are saved with the relevant job information to be restored and relocated when the job is restarted.

B. CNTL V (↑V)

Many times students will request the performance of special functions from a user job which will cause results they had not expected (e.g., lengthy interpretation). Since it is often desirable to stop the given process without killing the job (especially if the abnormal termination of the job can damage a file), ETS provides a character ↑V which can be used as a communication flag between the user and the running job. When a ↑V is typed at the user's teletype, ETS sets bit 0 in the variable USRFLG which the running job should interrogate periodically. The job is responsible for clearing the bit to indicate to ETS that it has noticed it. The action taken on the occurrence of a ↑V is generally an abort of some type. The job may either return to the system or ask for input from the user to determine the reason for the ↑V.

C. CNTL Q (↑QUIT)

With two exceptions, ↑Q works exactly like ↑V. ↑Q sets bit 15 of USRFLG, and by system convention, ↑Q is a request by the user for the program to terminate its processing and return to the monitor via a .EXIT.

VIII. System Service Routines

The following are routines which the system uses for some of its own processing, but which might be useful to users running under the system. The routines are compatible with a user's requirements of dynamic relocatability. In general, they may clobber one or more registers and will pass parameters in the other registers. It is the user's responsibility to save his own registers and to see that appropriate parameters are passed.

Since the position of these routines is dependent on the version of the system in use, users are cautioned against running programs on separte versions of the system without checking the new definitions of the routine names. Example VIII.1 uses many of these routines.

A. SETDDB

This is a routine to load the TTY Device Data Block address in R1 and the TTY line index in R0. While users may find little direct use for this routine, it is good for setting up parameters required by OCTIN and NAMEIN.

Call:

```
JSR   PC,SETDDB
;no parameters are necessary
```

Return:

```
;R0 contains the TTY Line Index
;R1 points to the TTY DDB
;no additional registers are clobbered
```

B. OCTOUT

This routine will output the octal number in R5 onto the teletype. The number will be followed by a <CR> <LF>. In order to output the characters

```
'OB         = xx                ;Byte containing job index
'BWAIT      = xx                ;Blockage table
'BSTAT      = xx                ;Unblock table
(EGREL      = xx                ;Register relocation
:ETDDB      = xx                ;System
IAMEIN      = xx                ;Routines
)CTIN       = xx                ;and
:OSTS       = xx                ;Variables
;IMES4      = xx                ;Entry point for system error message handler
'SKEY       = 20000             ;TTY delimiter
)PNFQ       = 10                ;Open function
'QFIL       = 4                 ;Byte offset in FIRAB to channel index
;ADCMD      = 1                 ;Invalid command error message
'ALFIP      = 104050            ;EMT to call file processor
;UFFER      = 4                 ;IOT to buffer allocator
(ETSML      = 0                 ;Buffer return parameter


!OVB        @#JOB,R0            ;Get job index - sign extend
!OV         #JSKEY,JBWAIT(R0)   ;Set blockage - we wait for the user to type a line
.IC         #JSKEY,JBSTAT(R0)   ;And make sure we wait
 WAIT                           ;Begin wait
 NCB        @#REGREL+5          ;Mark R5 as relocatable
 SR         PC,@#SETDDB         ;Load pointers to TTY DDB
 SR         R5,@#SETFQ          ;Get a FIRQB → R4 and load
            OPNFQ               ;OPEN function
 SR         PC,@#NAMEIN         ;Get a name and pack in FIRQB
 VS         ERR                 ;Didn't get a good one
 SR         PC,@#OCTIN          ;Get a channel number
 VS         ERR                 ;Didn't make it
 SL         R2                  ;Convert to channel index
 IC         #-7-1,R2            ;Make it legal (0-7)
 OVB        R2,FQFIL(R4)        ;Store it in FIRQB
 ALFIP                          ;Try the open
 OV         @#IOSTS,R2          ;Get pointer to I/O status
 OV         (R2),R2             ;Get error code
 NE         ERR1                ;Error - too bad
                                ;No error, FIRQB was not returned


 OV         #BADCMD,R2          ;Choose our own error code
 SR         PC,@#SIMES4         ;Output error message - R2 contains #
 UFFER                          ;Return the FIRQB
 ETSML
```

Example VIII.1 Accepting a File Name and Channel From the Teletype and Trying to Open it

OCTOUT needs parameters set up and calls SETDDB to do it.

Call:

```
JSR    PC,OCTOUT
;R5 contains the number to be output
```

Return:

```
;R0 contains the TTY Line Index
;R1 points to the TTY DDB
;R2,R5 clobbered
```

C. SETFQ

SETFQ is a routine to set up a FIRQB for use in file processor calls. SETFQ allocates a system buffer and stores in it the user's job number and the function requested.

Call:

```
JSR    R5,SETFQ
+      FUNFQ
;FUNFQ is the FIP function desired
;R5 should be marked relocatable
```

Return:

```
;R4 will point absolutely to the FIRQB
;R0 clobbered
```

When the user has finished with the FIRQB, he must return it to the system.

D. SIMESS

SIMESS is a routine to retrieve a message from the system message file and output it on the teletype. There are two entry points to SIMESS, the first will request a message from the teletype for outputting. The second, and more useful, will accept a message code in R2 and output the corresponding message.

Call:

```
JSR    PC,SIMESS     or     JSR    PC,SIMES4
;a message code is supplied in R2 when using
;SIMES4
;a message code will be accepted from the teletype
;when using SIMESS
```

Return:

```
;R0,R1,R2,R4,R5 clobbered
```

E. ENDTST

ENDTST is a routine to determine if a character is a terminal. Terminal characters are <space>, <CR>, <LF>, <ALTMODE>, or <↑C>.

Call:

```
JSR    PC,ENDTST
;R2 contains the character to be tested
```

Return:

```
;Z=1 if R2 contains a terminal character.
```

F. OCTIN

OCTIN is a routine to input an octal character from the teletype. It will pick up characters from the teletype and skip over any initial "terminals." When it encounters a non-terminal, it will attempt to interpret it as an octal character. It will continue accepting characters from the teletype until it encounters another terminal or a comma. If it encounters a non-octal character, it will ignore it and continue processing; however, it will return a flag to indicate that the number was invalid. In case of overflow, high order bits are truncated. No flag is returned.

Call:

```
JSR    PC,OCTIN
;R1 contains the DDB address (use SETDDB)
```

Return:

```
;R2 contains the octal value
;R3,R5 clobbered
;V=1 if error or no characters were found
;C=1 if a bad character was encountered
```

Note that OCTIN does not issue a read from the teletype but accepts characters from the monitor chain buffers; consequently, the user program should set the delimiter wait bit and execute a .WAIT before calling OCTIN. If the buffers are empty, an error (V=1) is returned.

G. NAMEIN

NAMEIN is a similar routine to OCTIN but it is used to pick up a character string and pack it in a FIRQB. In picking up the name, NAMEIN skips any leading terminators and then stops on the first occurrence of any later terminator. Before calling NAMEIN, the user must allocate a FIRQB for use in a file call.

Call:

```
JSR    PC,NAMEIN
;R4 points to a FIRQB (a function need not be loaded)
;R0 points to a TTY Line Index (use SETDDB)
;R1 points to a TTY DDB (use SETDDB)
```

Return:

```
;V=1 if there was no name in the TTY buffer
;R2,R3,R5 clobbered
;File name is packed RAD50 in the FIRQB at the
;appropriate places - see CSIFQ (section IV.C.13)
```

H. BUFFER

BUFFER is a routine to allocate or deallocate a 16 word system buffer. The call to the buffer allocater is an IOT which requires one parameter.

Call:

```
BUFFER    (=IOT)
PARM
;PARM = GETSML = 200 for allocate
;PARM = RETSML = 000 for deallocate
;R4 points to the buffer on deallocation
```

Return:

```
;R4 points to the buffer on allocation
;V=1 if no buffers remain
;No registers clobbered
```

I. CLRBUF

CLRBUF is a useful routine for emptying the user's TTY chain buffers.

On call, all remaining characters in the input or output buffer will be

flushed.

Call:

```
JSR     R5,CLRBUF
+       X
;X = TTO+4 = 26 for output buffers
;X = TTI+4 = 14 for input buffers
;R1 points to the TTY DDB
;R5 should be marked relocatable
```

Return:

```
;R3, R5 clobbered
```

Index

BLIOGRAPHIC DATA EET	1. Report No. UIUCDCS-R-72-523	2.	3. Recipient's Accession No.
Title and Subtitle: ETS System User's Guide			5. Report Date: May, 1972
			6.
Author(s): Donald W. Oxley			8. Performing Organization Rept. No.
Performing Organization Name and Address: Department of Computer Science, University of Illinois, Urbana, Illinois 61801			10. Project/Task/Work Unit No.
			11. Contract/Grant No.: GJ-812
Sponsoring Organization Name and Address: National Science Foundation, Washington, D. C.			13. Type of Report & Period Covered: Research
			14.

Supplementary Notes

Abstracts

This manual is intended to serve as a reference to persons who wish to write programs for use under the Educational Timesharing System - ETS. It discusses in detail the conventions and facilities used or supported by ETS. It assumes some familiarity with ETS and a thorough understanding of the PAL-11 assembly language.

Key Words and Document Analysis. 17a. Descriptors

. Identifiers/Open-Ended Terms

. COSATI Field/Group

Availability Statement: Unlimited Release	19. Security Class (This Report): UNCLASSIFIED	21. No. of Pages: 63
	20. Security Class (This Page): UNCLASSIFIED	22. Price

M NTIS-35 (10-70)

USCOMM-DC 40329-P71