Status of the TENEX Memos 1 March 1972

The TENEX Memos were first issued as a complete set in January of 1970. They were intended as a comprehensive design specification for the TENEX operating system, on which, at that time, coding had not begun. The set of memos was the result of a design effort stretching over a two year period, and most of the memos had existed in several earlier versions which were revised and updated as the design was firmed up.

It has now been nearly two years since the first version of the TENEX monitor was placed in operation. During that time, these memos have served not only as a guide to implementation, but also as the main source of information on the system for new users and for other prospective TENEX installations. Until the publication of "TENEX - A Paged Time Sharing System for the PDP-1Ø" in late 1971, this set was the only document which could give the reader a feel for the "flavor" of the system. It is still substantially more detailed in most respects than the paper.

However, in the time since they were issued over two years ago, they have become steadily more obsolete. Partly, actual implementation of the system showed certain things to be impractical or less optimal than had been thought, and partly, some ideas were re-evaluated and modified or phased out. Some updating was done in late 1970, at a time when the system had been operating for several months, and the more important sections were brought into line with reality. An additional year has past since that time, and evolution of the system has caused some obsolescence in some of the sections, and rendered others totally useless.

As time permits, new documentation will be produced which is current and which dominates the TENEX memos in detail and function. In the interim, in order to continue to provide the function which these memos have served, we have undertaken a review. Rather than suffering the delay of revising and republishing the entire set, we have used the following procedure:

1. Memos which are mostly incorrect or irrelevant have been taken out of circulation;
2. A description of the discrepancies and some new information has been written for the remaining memos and is provided herewith.

Fortunately, the most fundamental and important documents in the set fall into the latter category. Therefore, the set of TENEX memos as now distributed should be quite useful in conveying the "flavor" of the system, and in serving as an introduction to its features and general structure.

The memos now being distributed are:

TENEX-3 - TENEX Job Structure
TENEX-4 - File System
TENEX-5 - Terminal Service
TENEX-6 - EXECUTIVE Technical Description
TENEX-7 - Fork Structure and Communication
TENEX-8 - Monitor Calls and Pseudo-Interrupts
TENEX-12 -Scheduling and Storage Management

The most important memos for learning about the general structure of the system are 3 and 4. The TENEX Executive Manual provides a much more detailed and user-oriented description of the Exec than TENEX-6, however the memo does contain documentation on most of the Exec's privileged (wheel) commands. Memos 5 and 7 provide some useful programming information, mostly for machine language programmers, and some implementation details. TENEX-8 is almost entirely implementation details, and TENEX-12 is a technical discussion, neither of which are of immediate value to the programmer.

Notes on the Obsolesence of Memos

As implementation on the system was begun, it was envisioned that there would be an initial version called the "Mini-System" which would be sufficient to meet our immediate needs, and that the so-called "full-blown" TENEX would follow at a later time. In fact, the system has evolved continually from the time it was first put into operation, and there has been no clear demarcation between a "Mini-System" and any other class of system, whatever its name. The system is now referred to as TENEX, and it is in the state where most of the original design goals have been met, but there remain a number of major and minor improvements which are planned for the next 6 to 12 months. Therefore, statements which occur throughout these documents to the effect that something is or is not in the "Mini-System" actually provide no information about what is implemented in the current system nor when a particular facility will be implemented. The following notes on each section will attempt to clarify what is now implemented, what is expected to be implemented, and what is an obsolete specification.

TENEX-3 - Job Structure

Pages 1-7 are generally quite accurate; the remainder (implementation details) contains some fairly unimportant inaccuracies. Everything described has been implemented, including expansion of JSB storage beyond one page. The diagram on page 11 contains an obvious error; in the current system, the 128K area from 64K to 196K (addresses 200000 to 600000 octal) contain the swappable monitor and all system-common swappable storage. See also the TENEX Monitor Manual which contains a very detailed description of the monitor virtual memory.

TENEX-4 - File System

The information herein is generally correct; page 7, 'indirect pointer', 'backed-up' file, and 'protection string' blocks are not implemented and this specification may soon be obsolete. Page 11, disc allocation specification does exist in the directory descriptor block but is not used by the system. The entire question of resource allocation and limitation is now under study. Page 20, subroutine files (4.2) have neither been implemented nor specified beyond this description. It is doubtful that exactly this type of facility will ever exist; under study now are "pipeline files" and "pseudo-teletype" files which would dominate most of the proposed uses of subroutine files plus be easier to implement and have various advantages of their own. Page 22, mailbox files (4.4) do not exist and are not planned. Page 25, "protected entry" not implemented, obsolete spec. Saved environments do not exist in the current system (core image saving has been adequate for most requirements), so various discussions in that connection are obsolete; e.g. page 32, there are no "retention counts" in the system. Page 35, special files (8.), no mountable directory devices which move information automatically into the disc directory currently exist; this specification is obsolete.

TENEX-5 Terminal Service

In general, service for half-duplex terminals has been implemented. However, the type of half-duplex connection implied by the third paragraph on page 10 (where the system receives an echo for each character sent) has never existed in our hardware, so the facility described has not been implemented. The system currently supports teletype models 33, 35, and 37, the TI, and the ARPA network "virtual terminal". Various others which have requirements similar to these have been used satisfactorily. Page 3, line editing is not now available as a monitor service but is a planned future addition. Page 5, no monitor facilities especially for terminal paper-tape devices have been implemented. User programs exist for copying files to and from these devices with the necessary formatting, and that will probably continue to be sufficient. Page 8, two additional data modes have been implemented, see recent programming documentation. Page 11, terminal linking is presently implemented, but the implementation is not quite what is implied by this discussion.

TENEX-6 - Exec Technical Description

In most cases, features listed as "not implemented yet" are presently not implemented, and the specification is probably obsolete. Page 16, DEFINE not implemented; PROTECTION is implemented. Page 21, ALPHABETIC, CHRONOLOGICAL, and REVERSE are implemented. Page 24, file group descriptors (*in file name) are implemented for DELETE, LIST/TYPE, DIRECTORY, and UNDELETE, and in COPY for the source file only. Page 30, TEN50 command obsolete; DDT command implemented and starts or resumes regular DDT. Page 36, linking commands are implemented. Page 39, !LOGOUT command is obsolete, regular LOGOUT command takes optional job number to log out another job, and is legal if other job is same user as this job, or if this job is enabled.

TENEX-7 - Fork Structure and Communication

Information herein is generally correct, some implementation details are incomplete. Page 11, implementation of terminal interrupts has been changed and improved substantially, but from the user point of view, the changes have been mostly additions. Page 13, a more up-to-date version of this table is available in the JSYS Manual.

TENEX-8 - Monitor Calls and Pseudo-Interrupts

All monitor UUO's available on the KA-1Ø (40-77 octal) have been reserved and used for TOPS-10 (DEC system) compatible operations. Only the JSYS instruction is used for TENEX monitor calls. The discussion is still accurate and should be useful for system programmers seeking to learn about monitor coding conventions or attempting to understand the PSI system.

TENEX-12 - Scheduling and Storage Management

This section is generally accurate in its description of the function of each of the various modules mentioned. There is however, no real-time scheduler module. The algorithms described for each of the modules are generally obsolete, however they may be of value in understanding the evolution which these modules have undergone and therefore in learning about the operation of the current system from the listings. Specifically, most of the goals and concepts given on pages 5-16 for the scheduler module are still valid, although the means for achieving them has been improved. The algorithms described in the "balance set" section were somewhat more tentative when written, and so are only about 50% indicative of the operation of the current system.

Other Memos

The remaining memos from the original set are no longer being circulated. For the most part, they were speculative design discussions and do not now represent current system implementation or current thought.

TENEX Job Structure

A user program running under TENEX operates on a virtual machine which looks something like a PDP-1∅ arithmetic processor with 256K of attached memory. The virtual APR does not make available to the user program the direct I/O instructions (CONO, DATAI, etc.), but has a large class of instructions (JSYS's and UUO's) which provide access to monitor routines performing user-oriented I/O and other operations.

The TENEX monitor and paging hardware create an illusion of memory (called the <u>virtual memory</u>) which can be treated as ordinary core, e.g. machine instructions can be executed which load and store randomly in the 256K space. Each of these references is interpreted by the paging hardware and translated from the user's virtual address to an "actual" core address before being sent to the memory. A reference may cause a not-in-core trap which stops execution of the running program and initiates a monitor routine which changes the contents of memory after which the reference is completed. Thus the illusion of 256K of memory can be created for the user even though there may be less than 256K of total core on the machine, and though there may be other user and monitor programs in the actual core.

The TENEX hardware and software do more than just simulate real core. The virtual machine has facilities that are considerably more powerful and sophisticated than typical hardware configurations used directly. This memo discusses the basic structure of the entities which are provided by TENEX.

Memory

The only "real", general purpose memory in TENEX is the file system. It is "real" in the sense that it has relatively fixed names attached (memory is always referenced by logical user-selected names, never by hardware location such as disc address). Also, all information of any sort (data, programs, etc.) resides in the file system when not being actively used.

The characteristics of the file system are discussed in memo TENEX-4. Generally, any word of information in the file system is identified by:

1. File name (including user/directory) within total file system
2. Page number within file (Ø To (512↑2-1))
3. Word within page (Ø To 511)

A concatanation of 2 and 3 (page number * 512 + word number; 9 bits of word number attached to the right end of up to 18 bits of page number) gives a logical identifier of any word within a file. A portion of the file system, called the <u>random file logic</u>, allows user programs to make single word random references into a file given the word address and the job file number (JFN, the identifier of an open file).

Frequently, it is convenient to deal with information in the file system by treating a page as a basic unit. For this purpose page number and job file number are used. To reference information in a file, the file must be open. The file opening procedure involves acquiring a handle on the file by associating a directory name and a small number called a Job File Number (JFN) then presenting this handle to the monitor call for opening a file. A page of any open file in a job can be identified by a single 36-bit word containing.

Left Half: Job File Number

Right Half: Page Number

Ø 17	18 35
JOB FILE NUMBER	PAGE NUMBER

The PDP-1Ø APR does not directly reference information in files. It does fetch instructions and instruction operands from the virtual memory. To the APR, the virtual memory consists of 256K 36-bit word addresses. To the user, the virtual memory is 512 consecutively numbered page addresses, each consisting of 512 consecutively numbered word addresses. The pages of the virtual memory are effectively slots into which are placed indentifiers of pages in the file system. At any given time none, some, or all of the slots may be filled. In general, a user program may place any page of information from the file system into any page of virtual memory by:

1. Opening the file (if not already open)
2. Executing a monitor call giving:
    a. The Job File Number and page number of the desired page of information in one word.
    b. The number of virtual memory page which is to receive the information in another word.

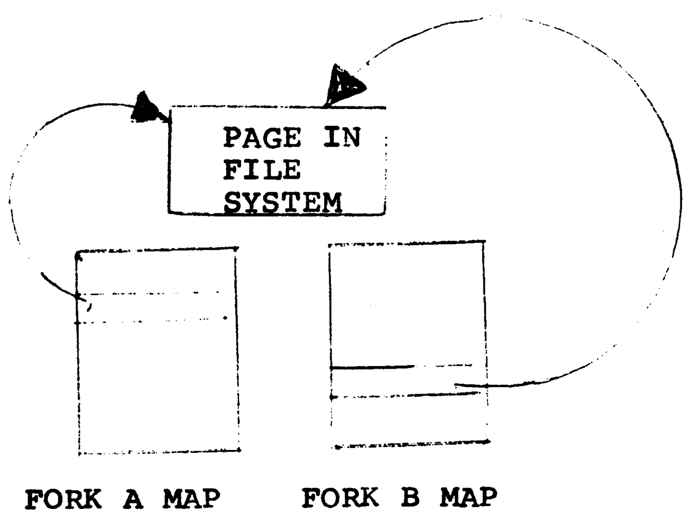
Then any of the words in the page are available to the APR for instruction or operand fetches.

The contents of the virtual memory at any time are specified by the virtual memory map which the user may manipulate. As well as setting words in the map, the user may read the map and move pages around. At any time, the user may ask for the "name" of the page in position N of his map (N in range Ø-511), and the monitor will return a word containing a Job

File Number and page number.

Processes and Forks

Precisely defined, a virtual memory is associated with a process (also called a fork). A process is a basic entity in TENEX. It is a logical entity capable of performing computation. Its state is contained in its virtual memory map and the state of the APR, PC and all flags. A program may be thought of as a named entity capable of performing a set of user related functions, such as LISP or DDT. By this definition, a running program must be associated with at least one and possibly more processes. See memo TENEX-7 for a complete discussion of forks.



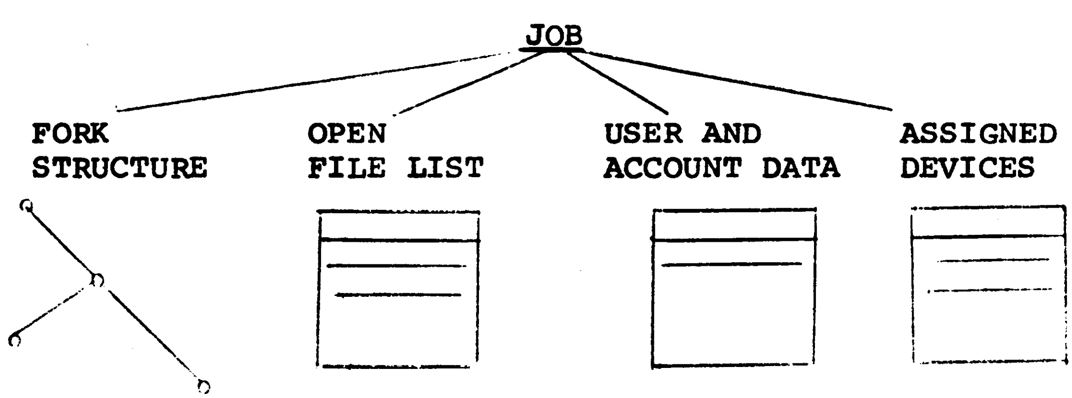
Direct (share) Pointer
Fig. - File Page Mapped Into Fork

Jobs

A job within TENEX is a set of one or more hierarchically related processes which can communicate with each other in defined ways. A job may contain several running or suspended programs. Each active process within TENEX is a part of some job. A job has the following attributes:

1. Name of user who initiated the job
2. Account number to which is charged all costs associated with use of system resources by this job.
3. Some open files
4. A hierarchy of running and/or suspended processes

A job may also have one or more terminal or other devices assigned and attached. Much of the information about the job resides in the Job Storage Block (JSB), a page which is referenced by the monitor and Exec, but does not appear in the virtual memory of any user process.



Private Memory

Every job has at least one open file, a file used as "private memory" by the job (analagous the 94Ø's PMT). This Private Memory File (PMF) is created and opened by the monitor when the job is initiated. Pages will be assigned when named private memory is acquired by one fork, and deassigned in various explicit and implicit ways. Memory for temporary storage is usually acquired by executing an instruction which attempts to reference an address in a page for which the map is empty, i.e. has no memory assigned. When this occurs, the monitor will assign a page (contents initialized to all Ø's) which is not part of any file and is therefore unnamed. If a name is later required (because of program request), the page will be assigned to and placed in the private memory file and will then have a regular JFN-PN name. These files of private memory for the running jobs exists in the PMFDIRn file directory, and the identification of the file for each job contains the system job number (a unique number) for that job. There will be as many PMFDIRn (e.g. PMFDIRØ, PMFDIR1, ...) directories as needed to permit private memory files for all the jobs on the system.

Some Implemention Details

Storage Blocks

Each job has a Job Storage Block (JSB), a page which is used to hold information which is global to the job. Each process (fork) within the job has a Process Storage Block (PSB) which holds information local to that process, and a User Page Table (UPT) which maps the user address space for that fork.

The JSB contains:

1. Job fork structure
2. Open file data and pointers
3. Certain pseudo-interrupt system data
4. Data on user, account, attached file directory, etc.
5. String storage for open file names and fork protection information.

In very large jobs, some of this information can grow to fill more space than is available in one page. When this happens, additional pages will be assigned (up to 8 total) to store the information. This expansion is not implemented in the initial TENEX system.

The top 96 words of the JSB contains pointers to the pages of the common portion of the monitor map. Monitor maps of all processes in the job have indirect pointers to these words of the JSB in the lower 96 positions. The remaining 32 slots of the monitor map are private to each process. The common pages are such things as:

1. Window pages for open sequential files
2. Index blocks for open files

The private pages are such things as:

1. PSB
2. Page table
3. Currently referenced file directory
4. JSB (share pointer)

The PSB contains:

1. Monitor call temp cells and PDL (TENEX-8)
2. Pseudo-interrupt statuses and states
3. Process PC and AC's when process is runnable but not running
4. Table of forks known to this process (TENEX-7)
5. Special capability table (TENEX-11)
6. AC block from user and monitor contents (TENEX-8)

The top 128 words of each PSB are reserved for the monitor map. The lower 96 of these are indirect pointers to job-common map in the JSB as above.

Mapping - File Pages

When a disk file is opened, an SPT (Special Pages Table in monitor) entry will be made for the index block. When a process requests one of the pages in that file for its map, the following will happen:

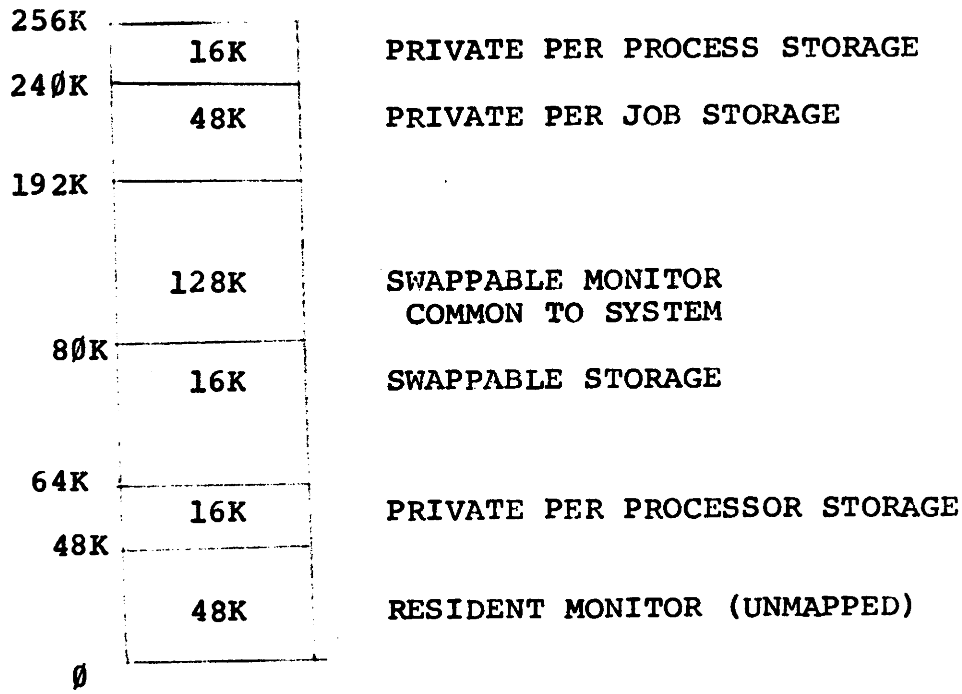
> An SPT entry will be made for this page (if not already in SPT), and the process page table will receive the appropriate share pointer.

Mapping - Private Pages

When a process first requests a new private page (usually by storing into an empty page of its map), a page will be assigned which is logically part of the PMF. The page is not actually put in the file at that time however, and the page table entry can be kept as a private pointer. However, when that process or any other process reads the map (i.e. requests the name of that page), the page must be made part of the file and the page table entry changed to a shared or indirect pointer as above. Then this identifier can subsequently be used to refer to the same page.

Monitor Map

In addition to the user virtual memory described earlier, each process also has a monitor virtual memory. Unlike the user space which is homogeneous, the monitor space is divided into a number of areas.



The top two areas were described above. The swappable monitor space is used to contain a large class of routines which are part of the monitor which can be swapped into core when needed. These include library-type functions (e.g. floating input and output) as well as system related functions. The swappable storage core contains the disk bit tables plus various I/O buffers which are dynamically assigned and locked into core when actually transferring data to or from an I/O device (such as dectape). The per-processor region is used to hold storage which must be different for each of the APR's used on the system. The lower 48K is normally unmapped (or mapped to identical core addresses) i.e. it refers directly to a contiguous, fixed area of core memory. It contains the central routines of the monitor which cannot be swapped out, including the scheduler, core manager, and most I/O drivers.

0. INTRODUCTION

The file system of TENEX provides a means of storing programs and data on various peripheral devices and providing access to such stored data. A file is more or less an ordered set of data which has a name or for some devices simply an unnamed stream of data. All files are handled uniformly with some operations unavailable for the more restricted devices.

File names are kept in directories with each entry in the directory relating the name to the location of information in the file. Directories are also named, and the names of directories are kept in a directory index. Each entry in the directory index relates the name of the directory to a number which can be used to determine the location of the directory. Separate directories are kept for each device in the system. So-called disc files may be kept on the swapping drum but still appear in the same directory as real disc files. Files may be shared in a very general way with explicit access protection. Furthermore, as many system tables and data bases as possible (consistent with efficient operation) are kept as files so that ordinary programs (with special status where necessary to protect the system integrity) may examine and process these tables for extracting information about the state of the system or performing routine activities like providing file backup.

1. FILE DIRECTORY STRUCTURE

All directories for the main file system of TENEX are contained within one large file. This large file is subdivided into regions of 4K (K = 1024) words each, one region for each directory in the system. The position of each directory in the large file is computable with simple arithmetic from a directory number which is associated with a directory name as described below in section 2. The file directory file is permanently open and referenced by a fixed file number. After a user logs in, the eight pages which constitute the portion of the directory under which LOGIN is done are mapped into a 4K region of the monitor address space of every process in the job.

1.1 Individual Directory Format

The portion of the file directory file devoted to an individual directory is divided into three areas. The area at the low end of the directory has a fixed allocation and contains all the bookkeeping information for the directory. The second region contains many different kinds of information and is described in section 1.3. The last region is an ordered symbol table which associates name strings with "value" information in the free storage region.

1.2 The Fixed Allocation Region

The contents of the fixed allocation region are listed and described below:

1.2.1 Directory Lock and Use Indicators

These two words are used to arbitrate various kinds of access to an individual directory and prevent simultaneous references from losing. The directory lock must be set before the directory is modified in any way which could affect other processes. The directory lock must also be set when a process which is examining the directory could be affected by another process modifying the directory. The use indicator serves to prevent unnecessarily locking the directory for long periods of time. Whenever the directory is in use, the use indicator is incremented. Before the directory can be locked for an extended period of time, the use indicator must show no uses are being made of the directory. Manipulating the use indicator requires the lock to be set.

1.2.2 Directory Number

The directory number is used to identify the directory currently being mapped for a particular process. This is done to eliminate unnecessary map switching. Because the information is redundant, it also aids in detecting system failures.

1.2.3 Symbol Table Bounds

Two words define the region of the directory which currently contain the symbol table.

1.2.4 Beginning of Free Storage List

The left half of this word points to the beginning of the free storage chain described below in section 1.3.

1.2.5 Default File Protection Word

The contents of this cell are used to initialize the protection word of a file descriptor block in the absence of an explicitly specified protection given by the user or program.

1.2.6 Directory Protection

This word is used to determine who may reference this directory, and how. The following kinds of protection are afforded:

a. list the directory
b. open files from the directory
c. add new files to the directory
d. attach to the directory
e. ownership rights (delete, rename, change account number)

1.2.7 Default Automatic Backup Protection
This word points to a block of bytes which specifies how many backup versions of various groups of files are to be retained.

1.3 Free Storage Region

The free storage region is used to contain all of the variable length data which is needed for the directory. Examples of items kept in the free storage region are name strings, file descriptor blocks, and protection strings. Space in this region is dynamically allocated and deallocated and incrementally garbage collected. This is done to reduce the need for total garbage collection. (Total garbage collection may be occasionally necessary due to fragmentation of the space.) Use counts are kept where necessary and at the time an item becomes unnecessary, its space is returned to the free storage pool.

All blocks in the free storage region which are in use, have similar formats. The first word contains in the left half a (negative) block type and in the right half the length of the block. The rest of the words in the block contain the data for the block. Free blocks contain in the left half a forward chain pointer to the next free block. The right half of a free block again contains the length of the block. The end of the free chain is marked with a zero in the left half of the first word of the last free block.

Space is allocated in this region by scanning the free storage chain for a block which is:

1. Exactly the right length.
2. Greater than the length needed by the size of the most common block.
3. Greater than the length needed by the smallest amount.

The above criteria are applied in the order given, and if a free block is found satisfying any of them, the space needed is extracted from the block and made into a new used block. If a block cannot be found which satisfies one of the above criteria, an attempt is made to expand the free storage area by moving the symbol table to the end of the next page and adding the 512 words thus gained to the end of the free storage list. If the symbol table cannot be moved then there is no room for this entry. Note that when this happens, it may still be possible to allocate blocks of smaller length.

Deallocation is done by scanning the free list for the free block just preceding the block being deallocated. The block is either linked into the chain, or if it is adjacent to either or both of the preceding or following blocks, it is merged with that block.

1.3.1 Block Formats

The formats of the various block types which may exist in the free storage region are given below.

1.3.1.1 Free block

word	contents
0	XWD pointer to next free block,length
rest	ignored

1.3.1.2 String block

word	contents
0	XWD 4000xx*,length in words(n)
1...	ASCIZ /the string/

(* xx are used to identify what the string is for)

1.3.1.3 File descriptor block

word		contents
0	XWD	400100,25 (block type and length)
1	XWD	control bits,name pointer
2	XWD	other extensions,SIXBIT extension
3		file address (including disc vs. drum) and class
4		protection word
5		first creation date and time
6	XWD	last writer,retention count
7	XWD	other versions,version number (job number is used for version number in the case of temporary files. See section 4.1.)
10		account number (positive) or pointer to account string (negative)
11	XWD	last byte size,number of versions to retain
12		length in bytes
13		this version creation date and time
14		last write date and time

```
15              last reference date and time
16       XWD    write count, reference count
17              last incremental backup date and time
20              last medium term backup date and time
21              last archival backup date and time
22              backup bits
23              backup location (tape number)
24              user settable word
```

1.3.1.4 File indirect pointer block

```
word     contents

0        XWD    400101,6
1        XWD    control bits,name pointer
2        XWD    other extensions,SIXBIT extension
3        XWD    directory number pointed to,
                pointer to file name string of
                file pointed to (includes name,
                extension, and version)
4               protection
5               creation date and time
```

1.3.1.5 Backed-up file block

```
word     contents

0        XWD    400102,3
1        XWD    control bits,name pointer
2        XWD    other extensions,SIXBIT extension
```

1.3.1.6 Account string block

```
word     contents

0        XWD    400200,length of block
1               retention count
2...            ASCIZ /string/
```

1.3.1.6 Protection string block

```
word     contents

0        XWD    400201,length of block
1               retention count
2...     BYTE   (indefinite number of bytes specifying
                protection)
```

1.4 Symbol Table Region

The symbol table region is used for associating a string or symbol with a block of information. Each entry in the table consists of one word containing in the left half a pointer to a string block and in the right half a pointer to a block of other information. Three bits in the top of the right half are used to indicate the type of information. Three types of information blocks are pointed to from the symbol table. There are: file names, group names, and protection names. Entries are ordered in the table alphabetically; i.e. the comparison used to order the entries is "SUB JCRYO" on successive words of the strings. The entry type is also used for ordering, and has the highest weight.

Entry type 0 is used for file names. The left half of the word addresses a string block in the free storage region which contains the file name string. The right half word addresses one of three types of blocks. The block addressed is either a file descriptor block, a file indirect pointer block, or a backed up file block.

Entry type 1 is used for group names. The left half of the word addresses a string block containing the group name string. The right 15 bits addresses a group descriptor block.

Entry type 2 is used for protection names. The format of the symbol table entry is parallel to the above two entry types.

2. DIRECTORY INDEX STRUCTURE

The directory index is also a file as is the file directory. The directory index is also divided into subindices of 4K words each in order to avoid map switching while searching. The correct subindex can be found on the basis of the first character of the directory name by dispatching into a table at the beginning of the directory index. Each subindex is divided into 3 regions which serve the same functions as the three regions of an individual file directory. The first 4K region of the directory index is not a subindex, but contains directory index global information including a directory number table for performing the inverse translation from number to string. This 4K region will be mapped as part of the swappable monitor. A 4K subindex will be mapped in the same 4K region of the process monitor map that is used for directories. To distinguish whether a directory subindex or a directory is currently mapped, each subindex will have the negative of its subindex number in the same position as the file directory number in a file directory.

The fixed allocation region for each subindex of the directory index contains the following items of information.

1. Directory subindex lock used to lock the directory to prevent its use while in a transient state.

2. Directory subindex use indicator serves to indicate the subindex is in use and cannot be locked.

3. Negative subindex number identifies which subindex is currently mapped in this process.

4. Symbol table bounds delimit the current limits of the symbol table.

5. A pointer to the beginning of the free storage chain is used to allocate and deallocate space in the free storage region.

The free storage region is managed in the same way as the free storage region of the file directory. The blocks in the directory index consist of mainly string blocks and directory descriptor blocks. The directory descriptor block describes all the attributes of a directory (and the associated user) which are necessary to identify it to the system. The items contained in a directory descriptor block are the following.

The password string pointer points to a string block which contains the password which must be typed by a user on a terminal in order to login under the particular directory name associated with this descriptor block. If this pointer indicates that no password exists, then login may not be done under this directory name. Files may be referenced by explicitly naming the directory, or attaching to the directory.

The directory name string pointer addresses the string pointed to by the left half of the symbol table entry whose right half addresses this directory descriptor block.

The maximum permanent disc allocation specification determines how much file storage the files in the directory associated with this block are allowed to occupy on a long term basis. Files may be stored on the disc in excess of this number, but at the time the user logs out, he is advised of the fact that he has exceeded his storage allotment and given the opportunity to reduce his storage requirements by deleting files or requesting certain files to be backed up. If he fails to do this, files may be moved to backup storage at the discretion of the system in order to reduce the storage used below this level.

The absolute maximum disc allocation specification determines a stronger upper limit on the amount of file storage a directory may occupy. If this allocation is exceeded, a user attached to this directory may not open any files for writing until he has deleted or requested a backup and delete operation for enough files to reduce his total usage below the absolute maximum.

The date and time of last LOGIN are saved to determine which (if any) login messages the user should see. This prevents the user from seeing a lot of junk that he has seen before.

A word of privilege bits is allocated for specifying special privileges available to a user logging in under this directory name.

A word of mode flags is allocated to control any options which the user has enabled for this directory.

The directory number gives the internal system handle for this directory. It is used to directly find the location of the directory corresponding to this name.

User identification information provides information about the user responsible for this directory which might be necessary to identify him to people. Address and telephone information would be kept here.

Information about special system resource guarantees are kept in the directory index so that such requirements can be stated to the system when a user logs in under this directory name.

The symbol table region of the directory index is similar to that of the file directory. The left half of each entry points to a string block containing the directory name. The right half contains the directory number.

The directory number table is used to translate from a directory number to the location of a directory descriptor block. Since the directory number space is not dense, the directory number table is a hash table. Each entry in the table contains in the left half the location of the directory descriptor block and in the right half the directory number.

3. FILE NAMES

File names in TENEX are composed of five identifiers. These are device, directory name, file name, extension, and version. These five items uniquely identify any file accessible to a user on the system. The device name identifies on what device in the system the file is contained. The directory name gives the directory under which the file appears. The file name and extension and version identify a particular file in the directory given by the device and directory name.

The character set from which these identifiers may be constructed is composed of the upper case letters, digits, and punctuation marks found in the range 40-137 (octal) in the ASCII set with the exception of the characters period : ; < > space and comma. These punctuation marks may be appear in an identifier if they are preceded by a control-V. The control-V is not part of the identifier. Lower case letters may be used, but they are equivalent to the corresponding upper case letters. This is done so that all file names may be typed on upper case only terminals such as model 33 Teletype terminals. The punctuation marks listed above as exceptions are used as delimiters for various fields of the file name and while their use within a particular field may not be ambiguous from a syntactic view, a blanket restriction on their use is made for simplicity.

The general form for a file name is:

device:< directory name> name.extension;version number ;T;Pprotection specification;Aaccount string

The underlined characters are real, the rest is representative.

A file name may either come from the memory of a process, or it may come from a file (including a terminal) or it may come from both memory and then a file. If any of the fields of the name are omitted, the field is supplied by the program, or from a standard set of default values. The standard default values are:

device	DSK
directory	currently attached directory
name	no system default, must be specified by program
extension	null string
version	no system default, program must specify usually most recent for reading, and next higher for output. Special ways of representing these cases are provided for the program.
T	file is assumed not temporary
P	as specified in directory (usually read all and write self only)
A	account number of login.

Recognition is done on file names in a uniform manner regardless of the source of input, or the intended use of the file. The program can control certain aspects however. Whenever an alt-mode is input from memory or file, the portion of the field input prior to the alt-mode is looked up according to which field is currently being input. A match is indicated if the input string either exactly matches an entry in the appropriate table, or is an initial substring of exactly one entry. In the latter case, the portion of the matching entry not appearing in the input string is output on the output file. The field terminator is output also, and recognition is done on successive fields with a null string as input, or if not possible, the fields are defaulted. If the file name cannot be uniquely determined, as much as possible is recognized and a bell is output signifying that more input is required. If the string input cannot possibly match any existing file name by appending more characters, an error return results.

Control-F behaves like alt-mode except recognition is not carried out past the current field. This allows the name to be recognized for example, but not the extension.

If an alt-mode is not used, then each field is delimited as indicated above, and the name so specified must exactly match some existing file name unless the program specifies that new file names are allowed (i.e. an output file). The complete file name is specified whenever all fields have been recognized, or a comma, space, or carriage return is input.

Confirmation may be required if the program has so specified in the call. In this case, one of three messages is output following termination of the name. The messages are: "New File" if no versions of the indicated file exist, "New Version" if other versions exist but the indicated one does not, and "Old Version" if the indicated file name and version already exists. Following the message, one character is input and if it is one of the characters alt-mode, space, carriage return, or comma a normal return occurs. Otherwise, an error return results and no JFN (job file number) is attached to the file name. The confirmation character may be read by the program by using the "back up one character" operation.

Editting characters are recognized while typing file names as follows:

↑ A deletes one character from the current. If no characters remain in the current field, a bell is output.

↑ W deletes the current field. If the current field is null, a bell is output.

↑ X causes the file name gathering operation to be aborted, and an error return given to the program.

Rubout deletes the entire input, and starts over.

↑ R retypes the entire name as specified so far and awaits further input.

4. FILES

There are several types of files in TENEX. These are all handled in a uniform way as far as possible. The exceptions are for operations which have no meaning for certain devices. Such operations are treated as illegal.

4.1 Ordinary Files

So-called ordinary files are maintained on disk/drum/core and provide the heart of the TENEX file system. When the unqualified term "file" is mentioned, it usually refers to an ordinary file. Each file has at least one index block which is essentially a page table in which each entry gives the location of one page of information in the file. There are variants of ordinary files which differ in the way the system handles them, but not in their basic structure.

Files longer than 256K have more than one index block and are called "long files". The location of index blocks for a long file is given in a higher level index block. Long files provide a means for storing up to 128KK or over 128,000,000 words. The management of long files is invisible to the user.

Frequently programs use scratch files to store intermediate results. The names for such files are usually built into the program and therefore if that program is run under under more than one job, under the same directory name, a conflict in the use of the name occurs. It is necessary that these programs reference different files. This is done by making the default version number for these so-called "temporary files" be the job number. Temporary files also have the feature that they nominally disappear when logout occurs and are therefore useful for storing data of a temporary nature.

It is occasionally desirable that the name of a file and its protection and accounting information be permanent even though its contents be deleted. Such a file is a user's message file which must be accessible to other users for appending messages, but must be deletable by the owner. For this purpose, a file may be declared "permanent". Note that a file may be permanent and temporary independently.

The existence of version numbers for files has been alluded to in the above discussion. Version numbers are nothing more than a further extension of the file name which is used for two purposes. First version numbers allow environments to be saved which refer to particular versions of files and subsequently resumed even if incompatible newer versions have been created in the meantime. Second, it provides an automatic backup version of a file which is rewritten because the usual default version number for files opened for writing is one greater than the most recent version number.

4.2 Subroutine files

A subroutine file is a mechanism for making a program look like a file. That is, file operations instead of transferring data to or from a device, make calls to a program. This allows special processing to be interposed between the data the main program sees and the data appearing on an ordinary file, or the generation of data by the subroutine file from internal strings or whatever.

Subroutine files will be implemented in a more general way than was done on the SDS-940 system. First, subroutine files will be processes and have all the capabilities of processes. Subroutine files may be called from any other processes in the job if access is permitted (as for other files). Subroutine files may be named and appear in file directories. Subroutine files will have multiple entry points so that the operations that can be performed on ordinary files can have their counterparts for subroutine files. Subroutine files can generate any of the special signals which can occur when using an ordinary file such as end-of-file, data error, etc. The only restriction placed on a subroutine file is that it may not be called recursively.

Opening a named subroutine file is equivalent to setting up a procedure type file as a process, and then declaring it to be a subroutine file. Opening an unnamed subroutine file is simply declaring a process to be a subroutine file. When a process is opened as a subroutine file, it is started once at its "open" entry. Subsequent data transfers cause the process to be restarted at its "input" entry for input operations, "output" entry for output operations etc. A byte size is associated with a subroutine file and the system packs or unpacks data to match byte sizes between the subroutine file and the program calling it. Operations for which no entry point is provided are treated as illegal.

4.3 Terminals

Under most circumstances, terminals are used like files. That is, they provide a stream of input characters, or accept a stream of output characters. The TENEX Exec will open the terminal for input and output so that programs will not have to do so. Terminals are discussed further in TENEX-5.

4.4 Mailbox Files

Another special kind of file is a so-called "mailbox file". Opening a mailbox file establishes a depository in the monitor for exchanging information between two or more processes anywhere within the system. Mailbox files are discussed more fully in TENEX-10.

5. FILE OPERATIONS

Using a file in TENEX is basically a four step process. First a correspondence is established between a JFN and a file name, next the file is opened establishing the mode and access permission and setting up monitor tables to permit the data of the file to be accessed, third data is transferred to or from the file, and finally the file closed fixing up the directory information and releasing the space occupied in system tables and disassociating the JFN and file name.

5.1 Getting a JFN for a File Name

The system call for getting a JFN for a file name has the following (possibly null) parameters.

1. String pointer to string to be processed as input
2. String pointer to default device name
3. String pointer to default directory name
4. String pointer to default name
5. String pointer to default protection
6. String pointer to default account
7. Default version number
8. Default extension
9. Temporary bit
10. Input JFN

11. Output JFN
12. Allow new file bit
13. Allow old file bit
14. Confirmation required bit
15. Print whole name if only partially specified (used mainly for partial strings from core)

This system call provides for all the ways of establishing an association between a JFN and a file name. The name may be either specified by a string in core, or by a string from a file, or both. It may also be used to change the Protection or account number of a file, but separate monitor calls will be provided for these purposes also.

Possible errors which might occur when using the above system call are that one of the device, directory, name, extension, version, or protection fields cannot be found, or the protection cannot be changed, or the directory is protected against changes necessary to accomplish the intent of the call (e.g. cannot create a new file in another directory).

5.2 Opening a File

Opening a file for explicit access is done by one system call whose parameters are the JFN, the access desired, the data mode, and the byte size (if applicable). Each of the accesses needed are specified independently and the file is successfully opened only if each of the accesses is possible. The possible accesses are:

1. Read
2. Write
3. Append
4. Execute
5. As specified by page table
6. Protected entry only
7. Thawed (not thawed means the contents can be changed while the file is open)
8. Wait

Read access allows the contents of the file to be read via byte input, string input, random input, or activation with only the read bit set in the page table. Write access allows the contents of the file to be written via byte output, block output, random output, or activation with the write permit bit set in the page table. Append access allows the file to be written only via byte output or block output. Furthermore, the byte pointer may not be changed. Execute access allows the contents of the file to be called as a procedure, or pages to be activated with only the execute permit bit set in the page table. As specified access allows the file to be referenced as for read, write, and execute accesses except that each page can be referenced only if the page table permits such access. Protected entry access allows the file to used to initialize a process and started only at specific entry points.

Thawed access allows the file to be referenced even when its contents are liable to be changed. I.e. the contents are not frozen. If the file is opened without thawed access, then there cannot be any writers of the file, and the system will not allow any writers to open the file as long as it is open without thawed access. Thawed access applies only to a specific version of a file. The wait bit controls whether the program will be blocked if access cannot be granted due to the thawed access bit. If the wait bit is one, the process will be blocked until access is permitted. If it is zero, an error return will be given.

Several errors may occur when attempting to open a file. All of the "not found" errors that may occur when getting a JFN for a file name may occur if the item in question is deleted in the interim. In addition, errors may occur if the access requested cannot be granted or if the JFN given does not have a file name associated with it.

5.3 Data Transfer

There are six I-O transfer operations: byte input (BIN), byte output (BOUT), string input (SIN), string output (SOUT), random input (RIN), random output (ROUT). There is also a monitor call to activate a page into the process address space (PIN). The parameters of the calls are given in the following table.

BIN,BOUT	JFN and a byte
SIN,SOUT	JFN and a byte pointer
RIN,ROUT	JFN and byte number within the file and a byte
PIN	Page identification (JFN.Page number), address

The index block for a file is simply a page table. Like all page tables, it contains only addresses of lower accessibility than itself. When a page table is on the disc, it contains only disc addresses.

The contents of a file are always accessed by activating pages of the file into an address space. This is usually done by making the pages of the file into shared pages and putting share pointers into the page tables. Sequential and random accesses are simply monitor calls which reference pages of the file which have been activated into regions of the monitor map of the process. These pages are called window pages into the file.

For each open file, there is an associated byte pointer which addresses the last byte read or written from the file. It is normally initialized to point before the beginning of the file so the first operation will reference the first byte of the file. In append mode, the pointer is initialized to the last byte of the file so that the first write will append to the file. This byte pointer may be arbitrarily repositioned within the file (except for append only files). Random I-O operations do this such that RIN followed immediately by BIN for the same file will read two successive bytes. Separate operations are available to arbitrarily reposition the pointer without transferring any data and also to read the current value of the pointer. If the pointer is positioned beyond the current end of the file, an end-of-file indication will be given if a read operation follows. If a write operation follows, the space skipped over is effectively filled in with zeroes. The pointer may not be set before the beginning of the file. Setting the pointer to zero causes the next byte referenced to be the first byte of the file.

It is also possible to change the byte size. When the byte size is changed, the next byte number is computed as:

< current byte number> *(36/< newsize>)/(36/< oldsize>)

Where A/B means greatest integer less than or equal to the quotient of A and B. This means that unless the byte sizes are commensurate, the next byte may overlap the previous byte. For example, if the previous byte size was 7, and the new byte size is 9, and the current byte number is 2, then the previous byte was number 1 and came from bits 7-13 of word 0 of the file. The new current byte number will be 1 and thus the next byte will come from bits 9-17 of word 0 of the file. Thus the five high order bits of the next byte will be the same as the 5 low order bits of the previous byte. It is not possible to change the byte size for certain devices or to change the byte size to greater than 36 or less than 1.

Several unusual conditions may occur while data is being transferred to or from a file. These conditions generate a pseudo-interrupt and other action as indicated below. These conditions may also be tested with a monitor call.

End of file	(BIN,SIN,RIN) Attempt to read beyond last byte of file. A zero byte is returned.
Device error	(All operations) An irrecoverable device error has occurred. The input data is the best obtainable. For input, each byte that is potentially bad will be accompanied by a device error indication. For output, the error will affect some data preceding the operation in which the error is signalled.
No room	No space is available on the storage medium. The operation in which this error is signalled is not completed.
Not open	The file has not been opened for this JFN.
Illegal access	Access for the attempted operation has not been granted.

5.4 Closing a File

The process of closing a file does several things. It updates the directory entry to reflect a new file length and to make the file known if it was a new file being written. The window pages are removed from all maps of this job, and share pointers converted back into private pointers for the file if the file is not open any place else. The pages of the file are declared to have a preferred residence on the disc (if applicable).

5.5 Other File Operations

There are other operations which may be performed on a file. These operations are not performed on an open file but rather on a file which has been associated with a JFN and not opened.

A file may be deleted. The contents of a deleted file do not disappear instantaneously. Instead, the file is marked as deleted, and the actual process of returning the storage used by the file is deferred to the operation of a file maintenance program which is run at periodic intervals. By deferring the actual deletion, a user has an opportunity to save himself if he accidently deletes the wrong file by undeleting the file.

A file may be renamed. Renaming a file simply changes the directory, name, extension, and version of the file. All other information about the file remains unchanged.

A file's protection may be changed either implicitly by specifying a new protection when gathering a file name to be given a JFN, or explicitly by means of a separate monitor call.

The account information for a file may also be changed implicitly by specifying a new account number when gathering a file name to be given a JFN, or by means of a separate monitor call.

Finally, various kinds of backup requests may be made for file. These requests are processed by the file maintenance program and allow the user to cause his file to be backed up in a special manner.

6. FILE SHARING

There are two ways in which reasons for the existence for a file may arise. First a file may be open in some number of jobs. Second, a file may have been open by a job at the time that the environment was saved. In both cases, it is not desirable that the file disappear even if it is deleted. The problem is to keep track of the reasons for retention of a file. Unfortunately, there is no crash proof way of doing this. It has been said that retention counts are always off by one and pointer structures always form loops after a crash.

When a system crashes, the most vulnerable information is in monitor tables. In this case, the record of open files is most vulnerable. By separating the record of open files from that of saved environment uses, retention counts can be made nearly crash proof. Thus one counter is kept in the file directory for counting the number of saved environments that reference a particular file. A second counter is kept in a table parallel to the SPT for counting the number of times the file is currently open. In this way, if the system crashes, it is possible to assume the file is not open, and reset the second counter without affecting the first.

7. FILE ACCESS PROTECTION

Because TENEX must service a diverse user community, it is essential that access to files be protected in a fairly general way. Generally, access to a file depends on two things: the kind of access desired, and the relation of the Program making the access to the owner of the file. Initially we will implement a simple protection scheme in which the only possible relationships a program may bear to the file's owner are:

1. The directory attached to the job under which the program is running is the same as the owning directory.
2. The directory attached to the job under which the program is running is in the same group as the owning directory.
3. Neither 1 or 2.

Six kinds of access are distinguished for a file. they are:

1. read
2. write
3. execute
4. append
5. directory listing
6. protection modification

The above six protection types and three relationships can be related by 18 bits (a 3 by 6 binary matrix) in which a one indicates that a particular access is permitted for a particular relationship.

Eventually a more general system will be used in which more general relationships exist such as everybody except groups A, B, and C. This scheme will probably use some sort of finite state machine processing a byte string to determine the validity of an access request.

8. SPECIAL FILES

Several special files exist in TENEX. Some of these such as the file directory and directory index have already been mentioned. Another special file is the disc allocation bit table.

The device "NIL:" is an infinite sink for dumping unwanted output and a zero length file (gives immediate end of file) for input.

All mountable directory devices in the system will have the directory for the device stored in a file. If storage requirements permit, the directory will permanently reside on the disc and will be named in a way which indicates the kind of device and its identification such as MTA12451. When something is mounted on a particular unit, a file name corresponding to the unit number is made into an indirect pointer to the directory file (e.g. MTAUNIT2 points to MTA12451 or some such naming convention) and the contents of the directory file compared with the directory stored on the reel (or whatever). Discrepancies are noted in an attempt to minimize operator error. When a discrepancy is noted, the operator is informed and allowed to either specify the correct reel number (if he previously specified it incorrectly) or to force the system to accept the directory on the reel.

TENEX Terminal Service

The TENEX terminal service routines will handle a variety of terminals including all models of teletypes as well as the IBM 'Selectric', the GE terminal, etc. The initial implementation of TENEX handles only models 33, 35, and 37 teletypes ASR or KSR, but is implemented in such a way that terminal dependent functions, such as code translation operations, can be added easily.

The overall design of the EXEC (Memo TENEX-6) and the general purpose CUSP's are based on the use of the model 33 and 35 full-duplex teletypes. This means that:

1. Features not available on these units (e.g. lower case) are not used in the design of control languages. However, optional use of such features is possible for special purpose programs (e.g. RUNOFF).

2. System design has not been compromised to accomodate those terminals which lack some of the features of the 33-35's (e.g. control characters). The system capabilities utilizing these features are:
   a. Not available on such terminals, or
   b. Simulated by some special sequence of input or output, or
   c. Available via alternate conventions determined by change of mode.

   Which of the above alternatives is adopted for each incompatibility is documented with the description of the various terminal and control features in this and other TENEX memos, or has yet to be determined. Operation of the system may be somewhat less convenient from such other terminals.

TENEX will provide a number of terminal-dependent functions related to half-duplex teletypes such as are mentioned in connection with the pseudo-interrupt capability. However, procedures which are required only for half-duplex terminals are in general <u>not</u> yet implemented. That is, half-duplex terminals are <u>not</u> usable on the current version of TENEX.

Terminal I/O

Input and output operations with terminals are done via the regular file system mechanisms and monitor call instructions, including:

1. Open File - The name TTY (as a device) designates the terminal whether typed in by a user or supplied as a string by a program. The direction of transfer (in or out) is a parameter of the call.

2. Transfer Data - The instructions BIN and BOUT transfer data between the user's AC and the terminal if given a JFN obtained by opening the TTY as above. For convenience, JFN's 100 and 101 will refer to the primary (usually terminal) input and output files respectively of a job.

3. Set Status - A number of JSYS instructions are available for specifying statuses which may be pertinent. Some are device dependent and will be ignored if inappropriate to the actual device.

Most necessary steps have been taken to ensure that the terminal and other serial character files may be used interchangeably. This means that either a regular file or the terminal may be specified any time a program requests a file name, and that the user can cause a file to be used in place of the terminal when the terminal is assumed.

To realize this goal and to relieve the various user and subsystem programs from terminal dependent concerns, the system provides an interface to the terminal(s) which:

1. Makes the terminal look like any other file to a program, and
2. Performs those functions which users expect to have on a terminal.

The following are specific points toward those ends:

1. Certain status bits and/or parameters apply to some types of files (devices) and not others. Monitor calls are designed such that parameters for all kinds of files may be communicated, and those which do not apply are ignored. Standard default cases are defined for unsupplied parameters.

   For example, a program may specify a wakeup control for a file regardless of whether or not that file is a terminal. If it is not, the specification will be ignored but remembered and given back to the program if requested. In this way a program can (and is expected to) give specifications for any situation which might exist where the default specification is not satisfactory. (See JSYS manual section 4B discussion of JFN mode word for details.)

2. The internal character set includes an End-of-Line (EOL) character which is the combined functions of carriage return and line feed. In normal modes, the terminal service routines will echo CR-LF (or just LF for HDX) when a CR is typed, and an EOL will be supplied to the program(1). Similarly, on output EOL will be translated to CR-LF. On input of LF, or output of CR or LF, no special action or translation will occur.

   Other format affector characters are output directly or simulated as may be required by the particular device to obtain the appropriate action. These include Form Feed (↑L) and Tab (↑I).

3. Line editing capabilities on input will be provided within the terminal service routines, including:
   a. Delete last character - ↑A
   b. Delete line - ↑X
   c. retype current line - ↑R
   d. Delete last word or field - ↑W

   Automatic line editing capabilities are not available in current TENEX but are a planned TENEX extension.

   These features can be available only if the program specifies an input mode of line-at-a-time. Clearly the line editor can only delete those characters which have not been delivered to the program, and will deliver buffered characters to the program on receipt of an input terminator. Therefore, characters can be deleted or retyped only back to the last input terminator. If this terminator is CR (and EOT perhaps), then one can reasonably and logically do line editing which seems natural to the user.

- - - - - - - - - - - -

(1) A subsystem may provide a quote operator to cause input of an actual CR, e.g. ↑V-CR typed would appear to the program as ↑V-EOL and could be interpreted as ↑V-CR. The LF would still be echoed however. Alternately, the program could set the mode to no-echo or use binary (8-bit) mode.

This capability is sufficient for most programs which require teletype input, and sufficient for the case where the user uses the teletype for input in place of a file. Programs which have a very interactive input syntax (e.g. DDT) or require special processing (e.g. LISP) will probably choose not to use the built-in edit features. Nonetheless, any such programs should conform to the editing character conventions used by the system, and great pressure will be brought upon all programmers to do so.

Paper Tape Readers on Terminals

Many terminals provide paper tape readers. TENEX will provide two alternative methods for reading paper tapes via these readers. The current version of TENEX does not have these features.

Method 1 (for FDX ASR terminals only):

On a full duplex terminal, it is simply necessary to output XON or to depress start reader to read in a paper tape. The system will detect the condition input buffer nearly full and characters being input at highest rate for that terminal. If this condition is true, an XOFF is transmitted and a flag (XOVFLG) is set to remember this event. When the input buffer subsequently becomes nearly empty and this flag (XOVFLG) is set, an XON is transmitted. All characters input get handled as though they were typed. Note this means CR gets input as EOL and echoed as CRLF. However, most paper tapes will contain CRLF which means EXTRANEOUS LF's WILL BE INPUT by this method.

Method 2 (for any ASR terminal, FDX or HDX)

A separate device name will mean terminal paper tape reader (e.g. TTYPTR). If the user specifies input from this file, an XON is transmitted and the XON, XOFF sequences mentioned above are transmitted to FDX terminals. For HDX terminals only specially prepared tapes (see TTYPTP) can be used. In all cases the sequence CR-LF is echoed as CR-LF, but the character EOL is input to the program. This mode filters out the extraneous line feeds.

If the user specifies the file name TTYPTP for output, the tape is begun with leader, and the sequence CR-XOFF-LF-RUBOUT is punched for each EOL. When the file is closed, trailer is punched. On input, when an XOFF is received, it is echoed immediately and the XOVFLG is set as above. On an FDX terminal, input buffer nearly empty and XOVFLG set causes XON to be transmitted, and on an HDX terminal, input buffer empty and XOVFLG set causes XON to be transmitted.

Input Buffer Overflow

For FDX terminals, input buffer full and any character typed in causes an immediate typeout of control-G (bell).

X-Y Paper Position

The system calculates the X-Y position of the paper in the terminal from the known motion caused by typing characters and format affectors. This position, represented by a line count and a character count is available to the user program.

Horizontal Tab

The system has a flag for each terminal which states whether the terminal has a mechanism for horizontal tabs. It is always assumed that these tabs are preset to a horizontal stop every eighth character. The system software tab settings are preset to these same stops. If the program outputs a tab and the software tab settings are consistent with the preset stops, and the terminal has a horizontal tab mechanism, the tab is output directly. In all other cases, the tab is simulated by multiple spaces. If the user outputs more characters than will fit on a line, the system will insert an EOL and ** to indicate that the line has been broken.

Three 36 bit words per file specify the tab settings. A bit represents each space on the line (maximum terminal paper width is 36*3=1Ø8 characters). A 1 means there is a tab stop at that position, a Ø means there is no tab stop. A monitor call is available to change the tab settings from the preset values (above) to any special setting the user might wish.

Form Feed

Form Feed is handled much like horizontal tab. The system will normally print through the paper folds unless programs explicitly transmit control-L. A system flag indicates whether the terminal has a FF mechanism, or whether software simulation by multiple LF is necessary. Vertical tab is not part of the set of format effectors in TENEX.

Terminal Dependent Specification

The system will remember the characteristics of the "normal" terminal used on each of the scanner lines. When a terminal is attached to a job, these characteristics will be used to initialize the words actively controlling the behavior of the service routines. If the user finds (or knows) that his is a different type of terminal (e.g. half-duplex), he may change the active status specifications by EXEC command or JSYS. Only the operators or other system personnel may change the permanent record.

These characteristics include:

a. Has lower case
b. Has horizontal tab
c. Has form feed
d. Is half-duplex
e. Page length
f. Page width

Note: (Parity (even) is always generated on control characters.)

Terminal I/O Control

There are a number of statuses which control the behavior of certain sections of the service routines. These statuses are recorded for each file but have significance only if the device used on that file is a terminal. They are set and read by regular file JSYS instructions. Normal or default states are defined for each condition and will be used unless the user specifies otherwise.

Echo Control

There are four echo mode states, defined by two bits. The first state eliminates all echos. This is useful for password input or to allow a program to use special echo characters by doing its own echoing. The three remaining states differ in the timing of the echo characters. The states are combinations of the following two basic types of echos:

1. Immediate Echo - An echo sent to the printer immediately on receipt of a character from the keyboard.

2. Deferred Echo - An echo deferred until the input character is delivered to the program

These two types are combined in the following four ways:

00 - No Echos

01 - Immediate echoing only. Causes terminal to behave somewhat like half-duplex (i.e. local copy) terminal except that typing during output does not produce garbled characters.

10 - Immediate or Deferred - This is the normal mode and should be most convenient for most applications. Echos are immediate until a wakeup character is struck. From then until the next time the program is blocked for input, the echos are deferred. This mode allows "typing ahead" but keeps the printed copy in correct sequence when there is rapid interaction between user and program.

11 - Immediate and Deferred - Like mode 10 except that immediate echos are generated in addition to deferred echos when the program is not blocked for input. That is, if the program computing and a key is struck, an immediate echo will be seen. The same echo will be printed again when the program gets around to accepting the character from the service routine. This allows users to see what they are typing when they type ahead and still see sequentially correct copy containing user and program typescript.

Wakeup

There are six bits reserved to specify classes of characters on which a program should be restarted after being blocked for input. Four are currently defined:

1. Format control (control characters having format effect, plus RUBOUT, EOL, and ESC)
2. Non-format controls (remaining control characters)
3. Punctuation
4. Letters and numbers

Logical Data Modes

There are four bits reserved to specify the data mode for all sequential files. Two of the 16 possible combinations are currently defined:

0 - Binary. For terminals means that all 8 bits received by scanner are passed to program unchanged and unechoed. Called 8-bit mode on some systems.

1 - ASCII. For terminals means that translation, echoing, etc. are performed as described and as specified by other modes. Data size is 7 bits.

Lower Case Output Control

A one-bit status tested if program outputs LC character and device does not have lower case printer.
1 - Indicate lower case. Print 'a' as '%A'.
Ø - Do not indicate. Print 'a' as 'A'.

Lower Case Input Control

A one-bit status tested if terminal supplies lower case character on input.
Ø - Echo and pass character to program unchanged.
1 - Convert character to UC for echo and program.

Control Character Output Control

A status tested when program outputs control character. Two bits for each of the 32 control characters.
ØØ - Ignore character
Ø1 - Indicate. Control-A prints as ↑A
1Ø - Send code directly and account for format effect.
11 - Simulate and account format effect (used when device does not have mechanical format device).

Normal setting is ignore null, send direct or simulate format affectors according to capability of device, indicate all others.

Terminal Interrupts

Thirty-six of the terminal codes are defined as potential interrupt characters. The user program may assign any of these characters to the designated PSI channels (c.f. PSI Structure, TENEX-7). Internally, each of the characters is assigned to one bit of a word as follows:

Bit	Code	Key	Bit	Code	Key
0	00	@ or break	18	22	↑R
1	01	↑A	19	23	↑S
2	02	↑B	20	24	↑T
3	03	↑C	21	25	↑U
4	04	↑D	22	26	↑V
5	05	↑E	23	27	↑W
6	06	↑F	24	30	↑X
7	07	↑G	25	31	↑Y
8	10	↑H	26	32	↑Z
9	11	↑I (tab)	27	33	ESC (ALTMODE)
10	12	Line Feed	28	177	RUBOUT
11	13	↑K (Vert Tab)	29	40	Space
12	14	↑L (Form feed)	30		dataset carrier off
13	15	CARRIAGE RETURN	31)		
14	16	↑N	32)----		To be assigned
15	17	↑O	33)		
16	20	↑P	34)		
17	21	↑Q	35)		

As is implied by the above, the check for interrupt character occurs before any translation or echoing is done. Interrupt characters are neither echoed nor placed in the input buffer.

With half-duplex terminals a special method is provided to allow interrupt characters to be typed when output is in progress. Output is suspended if an "echo-check" indicates the echo input character was not the same as the output character. Output will be suspended for 5 seconds during which time the user may type the interrupt character. At the end of this 5 second period, output will be re-enabled.

The character control-C is handled as one of the possible interrupt characters. It can be enabled, however, only by forks having a special capability, and will be so enabled by the EXEC at any fork level that it is run. The usual effect of this convention is that ↑C will return control immediately to the "current" EXEC (the EXEC most recently in control of the teletype), usually the top level. Potentially, privileged programs other than the EXEC could use control-C also.

Terminal Links

The link strategy will be similar to the 940 link structure but somewhat more restricted to permit links among a large number (perhaps 1000 or more) of terminals without the resident core requirments of the 940 structure (goes up as N squared). Instead it will be possible for only a few terminals to have complicated link structures at any one time, and each terminal may be linked to only a subset of the total existing terminals. The implementation is effectively a short list of terminals to which output is to be sent, and a short list of files to which input is to be given. The link structure is not implemented in the current version of TENEX.

Big Buffer Implementation

In order to reduce the amount of time spent processing terminal interrupts, as character interrupts are received (input or output) minimal operations are performed at interrupt level. All input characters are simply DATAI'd (this includes terminal line number) into a global input buffer with no echoing or testing performed. Input is moved from the big buffer to the separate line input buffers and any immediate echos are generated by a program running under control of the scheduler. The scheduler periodically (e.g. every 1/50 or 1/60 second) runs this routine. Deferred echos will be generated by those routines (in monitor mode) called directly by the user program to deliver characters.

Character Set

The internal character set consists of 128 characters of 7 bits each. The codes are generally those defined by 1967 ASCII as modified by DEC. They are consistent with BBN's Anelex line printer. The two old alt-mode codes 135 and 136 will be translated to ESC (33) on input unless the terminal is known to have lower case.

The control (non-printing) group Ø-37, is shown below

```
     Key  Mech Fn      TENEX Logical Function
     ---  -------      ---------------------
          null (or)
Ø    ↑@   break key    Fill - ignored on output
1    ↑A                Reserved for line edit - delete char
2    ↑B
3    ↑C                Reserved for EXEC interrupt
4    ↑D    (EOT)
5    ↑E    (WRU)
6    ↑F
7    ↑G   Bell

1Ø   ↑H   Backspace
11   ↑I   H. Tab       Tab - used directly or simulated
12   LF( J)            Line feed     Line feed
13   ↑K   Vert tab
14   ↑L   Form feed    Form feed - simulated if necessary
15   CR( M)            Car. Ret.     On input becomes EOL (37)
16   ↑N
17   ↑O

2Ø   ↑P
21   ↑Q    (XON)
22   ↑R                Reserved for line edit - Retype
23   ↑S    (XOFF)
24   ↑T
25   ↑U
26   ↑V
27   ↑W                Reserved for line edit - delete word

3Ø   ↑X                Reserved for line edit - delete line
31   ↑Y
32   ↑Z
33   ESC(ALT MD)
34    ↑\
35    ↑]
36    ↑↑
37    ↑←  EOL          Code used for end-of-line function
```

( ) Indicates mechanical function of concern only on some half duplex terminals.

The printing groups (4Ø - 177) are shown below.

<table>
<tr><th>Low 5
Bits</th><th>
4Ø-77</th><th>Group (high 3 bits)
1ØØ-137</th><th>
14Ø-177</th></tr>
<tr><td>ØØ</td><td>space</td><td>@</td><td>\</td></tr>
<tr><td>Ø1</td><td>!</td><td>A</td><td>a</td></tr>
<tr><td>Ø2</td><td>"</td><td>B</td><td>b</td></tr>
<tr><td>Ø3</td><td>#</td><td>C</td><td>c</td></tr>
<tr><td>Ø4</td><td>$</td><td>D</td><td>d</td></tr>
<tr><td>Ø5</td><td>%</td><td>E</td><td>e</td></tr>
<tr><td>Ø6</td><td>&</td><td>F</td><td>f</td></tr>
<tr><td>Ø7</td><td>'</td><td>G</td><td>g</td></tr>
<tr><td>1Ø</td><td>(</td><td>H</td><td>h</td></tr>
<tr><td>11</td><td>)</td><td>I</td><td>i</td></tr>
<tr><td>12</td><td>*</td><td>J</td><td>j</td></tr>
<tr><td>13</td><td>+</td><td>K</td><td>k</td></tr>
<tr><td>14</td><td>,</td><td>L</td><td>l</td></tr>
<tr><td>15</td><td>-</td><td>M</td><td>m</td></tr>
<tr><td>16</td><td>.</td><td>N</td><td>n</td></tr>
<tr><td>17</td><td>/</td><td>O</td><td>o</td></tr>
<tr><td>2Ø</td><td>Ø</td><td>P</td><td>p</td></tr>
<tr><td>21</td><td>1</td><td>Q</td><td>q</td></tr>
<tr><td>22</td><td>2</td><td>R</td><td>r</td></tr>
<tr><td>23</td><td>3</td><td>S</td><td>s</td></tr>
<tr><td>24</td><td>4</td><td>T</td><td>t</td></tr>
<tr><td>25</td><td>5</td><td>U</td><td>u</td></tr>
<tr><td>26</td><td>6</td><td>V</td><td>v</td></tr>
<tr><td>27</td><td>7</td><td>W</td><td>w</td></tr>
<tr><td>3Ø</td><td>8</td><td>X</td><td>x</td></tr>
<tr><td>31</td><td>9</td><td>Y</td><td>y</td></tr>
<tr><td>32</td><td>:</td><td>Z</td><td>z</td></tr>
<tr><td>33</td><td>;</td><td>[</td><td>{</td></tr>
<tr><td>34</td><td><</td><td>\</td><td>|</td></tr>
<tr><td>35</td><td>=</td><td>]</td><td>}</td></tr>
<tr><td>36</td><td>></td><td>↑</td><td>-</td></tr>
<tr><td>37</td><td>?</td><td>←</td><td>RUBOUT</td></tr>
</table>

TENEX EXECUTIVE TECHNICAL DESCRIPTION

The Executive is the user's handle on the TENEX time sharing system. The Exec is a user fork which decodes and executes requests for:

- access to the system
- utility operations regarding files and file directories
- initiating private programs and subsystems
- limited debugging aids
- initiation of batch (detached) operations
- printout of information including system statistics
- system maintenance

The set of acceptable commands is referred to as the Executive Command Language.

This memo is an outline of the current state of the TENEX Executive Language. It is intended for the systems programmers. A TENEX Executive Language User manual is also available.

Notation: The following notation is used in the commands and error messages given in this document:

- UPPER CASE is literal
- lower case describes argument fields in commands
- ()'s are literal - they enclose noise words
- []'s enclose optional fields in commands
- / means "or"
- <>'s are used with /'s for grouping where necessary

General Form of Commands

Each command begins with a keyword. Depending on the command, the initial keyword may be followed by arguments such as file names, numbers, and additional keywords, and/or "noise words" to make the command more readable. The noise words will be enclosed in parentheses to distinguish them from the arguments. The initial keyword usually identifies the command function. Some commands include optional arguments or argument lists of indefninite length. A few commands, such as that for file directory listing, take optional "sub-commands", each with arguments, to specify options.

Any initial word not recognized as a command keyword is taken as the name of a subsystem to be started.

COMMAND INPUT

The exec types "@" when ready for command input, or "@@" when ready for a "sub-command", except that "!" and "!!" are used for privileged users who have ENABLEd their special capabilities.

Three general styles of input may be used. The styles are distinguished by syntactic analysis and by input terminators; hence they do not require different input modes and thus may be intermixed freely within a session or even within a statement.

1. Complete Input. A complete command may be typed in, with all keywords and noise words given in their entirety, and without any non-printing characters being used. This style is good for novices who are copying a typescript, command files, IBM terminals without special characters; also it may simplify documentation.

2. Abbreviations. The user may shorten a command in two ways: he can omit noise words completely, and he can shorten keywords. Any keyword may be abbreviated with any initial substring (terminated with space) long enough to distinguish it from the other keywords acceptable in that context. Keywords will be made unique in three characters or less insofar as possible without producing very non-english-like words.

3. Recognition. The user types the same characters as in abbreviated input, except he terminates each field (keyword or argument) with the altmode key. This produces a print-out of the complete command -- each alt mode causes the rest of the field (if an abbreviated keyword or file name) and any following noise words (with enclosing parentheses) to be printed.

After recognition and checking of the arguments, most commands will wait for the user to type a confirming carriage return before execution, except that if the user terminates the last field with carriage return (as opposed to space or alt mode) confirmation is skipped. As indicated in the descriptions, some commands may be confirmed with an altmode, which is echoed with a carriage return. Some commands, notably those that write files, require confirmation even when the last field is terminated with a carriage return. These always type something to prompt the user before awaiting the second, confirming carriage return: either [OLD/etc FILE] if the last argument was an output file name, or [CONFIRM:] if not.

The above may be clarified by a description of the input terminators:

Space: may be used to terminate any field (full key word, abbreviated key word, parenthesized noise word, or argument) when recognition is not desired.

Alt Mode: same but causes recognition: types rest of abbreviated key word or file name, then any following noise words. If an ambiguous abbreviation has been given, rings bell and accepts further input. Also acceptable as confirmation character for certain commands.

Carriage Return: confirmation character. Also optional as terminator for last field of command, in which case recognition is not used and confirmation is omitted (except for commands which write files). Also used to terminate list of sub-commands.

Comma: special meanings in certain commands; indicates that additional arguments are to be given in an indefinite list (as in SAVE command), or that sub-command input is desired (as in DIRECTORY).

Semicolon: acceptable in place of confirming carriage return, or terminating carriage return when last field is not a file name. Causes characters to be ignored until carriage return, to allow comments (particularly useful in command files).

Optional fields, defaulting, nulling: Some commands end with one or more optional fields. If an alt mode only is entered in one of these fields, the field is defaulted, and its default value is printed (if it has no particular default value, "-" is printed). Entering a carriage return only defaults the field and all following fields without further printout; terminating a field with a carriage return defaults all following fields. To explicitly null a field and continue to the next one, without the noise word printout invoked by the alt mode (not to mention the difficulty of editing command files containing alt modes), you may enter "-" followed by space or any other terminator legal in that context. It is not possible to null a field by typing space alone, because leading spaces are ignored. See the description of the DETACH command for examples of fields you might wish to null.

Extra spaces and tabs may be used freely in commands, to permit formatting of typescripts or command files. For example, a command can be terminated with one or more tabs, a comment (text preceded by ";"), and a carriage return.

Continuation: A command may be continued on the next line by typing "&" in any context where a space is legal; a carriage return will be echoed.

Form feed (↑L) is treated as a carriage return in most contexts, except that it is not legal as a file name terminator.

Lower case letters in Exec commands are treated the same as upper case letters.

EXAMPLES: These show the use of various input terminators. User typing is underlined, with alt mode represented as "$" and carriage return as "%".

```
  DAYTIME (a command not requiring confirmation)

  @DAYTIME%
   MONDAY, NOVEMBER 16, 197Ø 13:36:56
  @DAY
   MONDAY, NOVEMBER 16, 197Ø 13:37:06
  @DAY%
   MONDAY, NOVEMBER 16, 197Ø 13:37:12
  @DAY$TIME
   MONDAY, NOVEMBER 16 197Ø 13:37:22

  COPY name (TO) name (confirmation mandatory)

  @COPY FOO.MAC;1 (TO) FILE.MAC;1 [NEW FILE]%
  @COP FOO.MAC FIE.MAC;1 [OLD VERSION]%
  @COP FOO.MAC FIE.MAC;1%[OLD VERSION]%   (first carriage
       return not echoed,  does not suppress confirmation)
  @COP$Y FO$O.MAC;1 (TO) FIE.MAC$;2 [NEW VERSION]%

  SAVE (CORE FROM) n (TO) n, (FROM) n (TO) n... (ON) name

  @SAVE (CORE FROM) Ø (TO) 2ØØØØ (ON) FOO.SAV [NEW FILE]%
  @SAVE Ø 2ØØØØ FOO.SAV%[NEW VERSION]%
  @SAV$E (CORE FROM) Ø$ (TO) 2ØØØØ$ (ON) FOO.MAC$;5%
@SAV Ø$ (TO) 2ØØØØ, 4ØØØØ$ (TO) 5ØØØØ$ (ON) FOO.SAV%[NEW VERSION]%

  DIRECTORY (Types file directory.  If comma typed before
  confirming carriage return or alt mode, takes "sub-commands".)

  @DIRECT%

  @DIR$ECTORY$

  @DIR$ECTORY%

  @DIR$ECTORY,%
  @@CH$ONOLOGICAL (BY) $WRITE% (field defaulted)
  @@O$UTPUT (TO) LPT%
  @@V$ERBOSE%
  @@% (CR terminates sub-commands, starts listing)

  @DIR,%
  @@CH%    (optional field omitted)
  @@O LPT%
  @@V%
  @@%
```

Input Editing

The editing characters available in the exec are:

↑A	Delete character, echo "\" and character deleted
↑W	Delete word or field, echo "←"
↑X	Delete command, echo "↑X"
rubout	Delete command, echo "XXX"
↑R	Retype line

The implementation of ↑A and ↑W is only partial because the exec does not read in an entire statement before processing it, but rather processes each statement field-by-field as it is entered in order to do recognition and to give immediate error messages. It is only possible to back up within the current field with ↑A or ↑W; once a field has been terminated with space, alt mode, or comma, it will be possible to change that field only by deleting the line (↑X) and entering it again in corrected form. Thus the number of consecutive ↑A's that can be processed is limited to the number of characters that have been typed in the current field; the number of ↑W's is never more than one. If an invalid ↑A or ↑W is entered, the exec will ring the bell rather than printing "\" or "←".

Exception: when a file name is being input, "↑X" deletes the entire file name and echoes "←←←" and a carriage return. "↑W" in a file name deletes one field (name, extension, version, etc.).

Interrupt Characters

When the monitor sees a ↑C, it will PSeudo-Interrupt a fork which has a PSI channel enabled for ↑C. Since a special status is required to enable a ↑C PSI, this fork will normally be the exec. Upon receiving a ↑C PSI, the exec will take the following actions: suspend any running subsidiary forks, terminate any command being executed within the exec, delete any partially typed in exec command, clear terminal input buffer, and echo "↑C"- carriage return. If a second "↑C" is typed promptly, the terminal output buffer is also cleared. Thus two ↑C's provide quick termination, while a single ↑C will lose no output from a running program.

Error Messages

For each command, a list of error messages is given. In addition to these, there are many general input syntax errors, most of which type "?" and return to command input. Some errors type " ? " and allow the user to try the same field again; these are labeled "type 2" errors. This happens, for example, if a garbage character is input when a confirming carriage return is expected, or when an input file is not found. Exception: if the erronious argument was terminated with a carriage return, the command is always aborted.

In addition to the messages listed with each command, all commands which require the user to be logged in type LOGIN PLEASE if he is not.

File name errors: Some or all of the following group of errors apply to every command containing a file name argument:

```
" ? " if file not found and existing file required (type 2)
NO JFN'S AVAILABLE: YOU MUST CLOSE SOME FILES FIRST
JSB FULL: TRY CLOSING SOME FILES THEN REPEATING COMMAND
DIRECTORY FULL: CAN'T CREATE NEW FILES UNTIL YOU
     DELETE SOME FILES AND "EXPUNGE (DELETED FILES)"
NEW FILE NAME REQUIRED
DEVICE NOT MOUNTED
DEVICE ASSIGNED TO ANOTHER JOB
NO FILES IN THAT DIRECTORY
```

Error pseudo-interrupts: If an error condition produces an unexpected error pseudo-interrupt while an exec command is being executed, a message such as ILLEG INSTRUCTION TRAP IN EXEC is printed, followed by the PC, AC's 1, 2, and 3, and a system error message (a la ERSTR JSYS). Faults such as illegal instruction in user programs which do not do their own error PSI processing cause the exec to print a message such as ILLEG INST 0 AT 771371 and return to command input. (Any PSI occurring on a channel which the program has activated but not assigned a level to causes termination; where the channel number does not implay a fault, "CHAN n INTERRUPT AT pc" is typed).

JSYS error returns: An unexpected JSYS monitor call error return produces a message of the same form, beginging with JSYS ERROR RETURN IN EXEC. The Exec tests for all but the least likely JSYS error returns and types messages of its own; any easy-to-reproduce method of getting a JSYS ERROR RETURN IN EXEC message constitutes a bug and should be fixed or reported to the person who can fix it.

System Access Commands

LOGIN

Two general forms, and mixtures thereof, are permitted:

LOGIN (USER) user (PASSWORD) no echo (ACCOUNT) account

LOGIN
(USER) user name
(PASSWORD) no echo in full duplex case
(ACCOUNT) account

The second form is inconsistent with the general form of Exec commands; it was added because a string account field might otherwise run off the right margin, and because of half duplex password input considerations. The second form can be invoked by the user by terminating fields with carriage returns. A modification of it is forced by the Exec for password input on a half duplex terminal: the Exec forces a carriage return after ther user name and password fields, and just after typing "(PASSWORD)" it types a carriage return and a mask of giberish characters onto which the user types his password.

The input acceptable for the "account" argument depends on the user: some users require string account numbers, and any string of 0 to 39 alphanumeric characters is accepted; others require a number, and any decimal number from 1 to 999999 is accepted. For users in the latter category, the Exec types "(ACCOUNT #)" instead of just "(ACCOUNT)".

This command types out a message containing the TSS job number, the terminal line number, the date and time of login, and the system login message (see below).

In a later version of the system, this command will, for certain users, autoatically transfer control to a default subsystem, eg, TELCOMP.

Alt mode is acceptable for confirmation. Obviously, the job needn't be logged in. Minimal abbreviation: "LOG", despite "LOGOUT".

System Login Message: The operator may write a message to users to be printed when they LOGIN on the file <SYSTEM>LOGINMESSAGE. If this file exists, LOGIN prints its contents if the write date and time of the file are more recent than the date and time the user last logged in. For certain users, this file is printed at every LOGIN even if it is older than the last login date and time for this user (this action is invoked by a directory bit -- see !CREATE below).

Message Message: If a file MESSAGE.TXT;1 of non-0 length exists in the user's directory, the Exec prints YOU HAVE A MESSAGE after LOGIN or after a ↑C during LOGIN's printout.

Errors: YOU ARE ALREADY LOGGED IN
"?" after user name if no such user
"?" after password if incorrect
"?" after account if must be numeric but isn't

LOGOUT

Closes all open files and logs job off system;
may also be used to kill unloggedin jobs.

Errors: NOT LEGAL IN INFERIOR EXEC

CHANGE (ACCOUNT TO) account

Changes account for subsequent charges.

Account: as for LOGIN.

Errors: "?" if account must be numeric but isn't

ATTACH (USER) user name (PASSWORD) no echo (TSS JOB #) job number

Attaches terminal to existing, detached job. General format and handling of password on half duplex terminals same as LOGIN. "Tss job #" field defaults to number of existing job with given user name if there is exactly one such detached job.

If current job is not logged in, it goes away; if it is logged in, ATTACH detaches it and type "DETACHING JOB n".

The ATTACHed-to job continues running, or starts running if it was hung waiting for a controlling terminal. If the job's primary I/O was redirected, ATTACH does not itself redirect it back to the terminal, but typing ↑C after ATTACHing will do so.

Note: ENABLEd WHEELs and OPERATORs may "steal" a job: for them ATTACH types [ATTACHED TO TTYn] instead of giving a JOB n NOT DETACHED error, then requires a(nother) confirming carriage return.

Alt mode is acceptable for confirmation.

```
Errors:   "?" after user if no such user
          THAT'S A FILES-ONLY DIRECTORY NAME
          "?" after password if incorrect and
               current job is not logged in
          JOB n NOT DETACHED
          JOB n NOT LOGGED IN
          JOB n NOT LOGGED IN UNDER name
          NO DETACHED JOB LOGGED IN UNDER name
          TSS JOB # REQUIRED - name HAS
               MORE THAN ONE DETACHED JOB
          INCORRECT PASSWORD (if logged in)
```

DETACH (INFILE) [existing file name/*] (OUTFILE) [file name/*]
(AND) [START/REENTER/CONTINUE]

Detaches job from terminal, leaving terminal free. If all arguments are omitted (eg by typing "DETACH-carriage return"), the job hangs, because the Exec immediately requests input from the non-existent controlling terminal.

The optional file name arguments will later permit redirecting primary I/O in a manner similar to the REDIRECT command (see index). At present they are NOT INPLEMENTED YET; the fields should be nulled by entering "-" if you want to get past them to give the third argument.

The third argument permits starting the program under the Exec. If it is omitted but an input file is given, then the Exec proceeds to take commands from the file.

Errors: see REDIRECT

Resource Allocation Commands

ASSIGN device name (AS) [logical name]

Assigns device to this job, preventing other jobs from using it. Also mounts device if mountable. Synonyms (logical names) are NOT IMPLEMENTED YET.

Errors: " ? " if no such device (type 2)
device: CANNOT BE ASSIGNED
device: NOT AVAILABLE (not assigned, but in use by another job)
device: ALREADY ASSIGNED TO JOB n
YOU CAN'T ASSIGN YOUR CONTROLLING TERMINAL
SYNONYMS NOT IMPLEMENTED YET
TTYn: IS THE CONTROLLING TERMINAL FOR JOB n

DEASSIGN device name

Cancels a previous ASSIGN for the same device: unmounts it if mounted, and makes it available to other jobs.

Errors: " ? " if no such device (type 2)
device: NOT ASSIGNED
device: NOT ASSIGNED TO YOU

LIMIT (ADDITIONAL) CORE/DISK/CPU/KILOCORESECS (TO) number

Command to allow users to impose limits on their resource usage. NOT IMPLEMENTED YET, and probably won't be fully implemented in Minisys.

Alt mode acceptable for comfirmation.

Errors: limit unreasonable
limit greater than your absolute maximum

File Commands

COPY existing file name (TO) file name

After the output file name is terminated, system responds with [NEW FILE], [NEW VERSION], [OLD VERSION], or just [CONFIRM] for a non-directory device.

Confirmation is mandatory: a carriage return must be typed after [NEW FILE]/etc, even if one was typed just before it.

The output file's name and extension are defaulted to be the same as the input file's.

Subcommands: If a comma is given before the confirming carriage return, subcommands are accepted after confirmation. The subcommands are:

ASCII

Causes copying in ASCII mode (mode 1) for devices for which this is legal (as determined with DVCHR), and in normal mode (mode 0) for other devices, in 7 bit bytes. Useful mainly with paper tape or to cause proper byte-size to be stored in file when copying to disc from another device.

BINARY

Causes copying in BINARY mode (mode 14) for devices for which this is legal, else mode 0, in 36-bit bytes. If either device is a terminal or line printer (for which 36 bit bytes are illegal, causes the warning message

[ILLEGAL MODE SUBCOMMAND BEING IGNORED]

and copying in mode 0, byte size 7. Useful mainly with paper tape.

IMAGE BINARY

Causes copying in IMAGE BINARY mode (13) where legal (paper tape only); otherwise the same as binary.

.IMAGE

Causes copying in IMAGE mode (10) for devices for which this is legal (else mode 0) and 8-bit bytes. If mode 10 is legal for neither source nor destination,

[ILLEGAL MODE SUBCOMMAND BEING IGNORED]

is typed and the mode and byte size of copy defaulted as though no subcommand had been given. Useful only with paper tape.

Typing a carriage return only terminates subcommand input and starts copying.

Method of Copying: The method of copying depends on the devices on which the source and destination files are since for some devices, particularly paper tape, there is more than one useful mode of copying. The mode subcommands described above are provided. In most cases other than disk-to-disk copying (below), byte I/O is used. This method does not transmit any zero bytes which may occur in the file if the byte size is 7, and cannot transmit "holes" (non-existent pages) which can occur in disk files (see warning messages below). If the mode and byte size were not specified with an acceptable subcommand, normal mode (mode 0) is used and the byte size is defaulted as follows:

IF a terminal or line printer is involved, use 7-bit bytes.

ELSE IF both files are on disk, use source file's byte size.

ELSE IF source is PTR: and destination PTP:, use 8 bit bytes. This duplicates any paper tape.

ELSE IF source is disk file and destination is PTP:
IF source file byte size is 7, use 7 bit bytes (ASCII paper tape mode).
IF source byte size is 8, use it (IMAGE paper tape) and type warning message (see below).

ELSE IF either PTR: or PTP: is involved, test extension of other file. Use 36-bit bytes (BINARY paper tape) for .REL or .SAV source, 7-bit bytes (ASCII paper tape) for others, typing a warning in either case.

ELSE use 36-bit bytes. This covers all other copies, such as those between DECtapes or between a DECtape and disk.

Note that use of 7-bit bytes for binary data loses bit 35 of each word, and assumes the wrong paper tape format. Also note that if an ASCII file is copied from disc to DECtape, then back to the disk, without using the ASCII subcommand, it will have a byte size of 36 stored with it. This is not likely to have any ill effects, but one possibility is that it will be subsequently copied to paper tape, in which case the wrong format (BINARY instead of ASCII) will be assumed if no subcommand is given.

Disk-To-Disk Copies: If both files are on disk, and COPY (as opposed to APPEND) is being used, and no subcommand was given which specified a byte size different from that of the source, then copying is done by pages. This method maintains the file's page structure, keeping any "holes" and copying pages after the byte EOF.

Warning messages:

[ILLEGAL MODE SUBCOMMAND BEING IGNORED]

The following two messages can occur when source is disk but copying by pages is not being used (see above).

[HOLES IN FILE WILL BE LOST]
If disk source file consists of non-contiguous pages, nothing is put in the destination for the "holes", so that the data is compacted downward.

[PAGES AFTER EOF WILL NOT BE COPIED]
If there are pages in the disc source file after its "end of file" (the highest address to which byte I/O has been done), they are not copied. Such pages may be put into a file with the PMAP JSYS and occur in SSAVE files.

[YOU DIDN'T SPECIFY PAPER TAPE FORMAT WITH A SUBCOMMAND, SO "mode" IS BEING ASSUMED.]
"mode" can be ASCII, IMAGE, or BINARY.

Errors: file name errors
READ PROTECT VIOLATION FOR FILE name
WRITE PROTECT VIOLATION FOR FILE name
name CAN ONLY BE APPENDED TO
device: IS ASSIGNED TO JOB n
device: IS NOT MOUNTED
device: CAN'T DO INPUT
device: CAN'T DO OUTPUT
device: CAN'T DO NORMAL MODE INPUT
device: CAN'T DO NORMAL MODE OUTPUT
FILE name BUSY

APPEND existing file name (TO) existing file name

Description for COPY applies. Data is always transferred by bytes, in the byte size of the source file.

Errors: see COPY. also:
DESTINATION FILE NOT ON DISK

RENAME (EXISTING FILE) existing file name (TO BE) file name

After output file name is terminated, system responds with [NEW FILE] etc, as for COPY. Confirmation is mandatory. Output file name and extension default to those of input file.

Errors: file name errors

DEFINE (NEW FILE) new file name (AS) file name

Creates an "indirect pointer" for the first file name to the second file name. If the second file name does not exist the command is allowed anyway, since the pointer needn't point to a real file until it is opened for input.

Errors: file name errors
FIRST FILE NOT ON DISK
protection error

PROTECTION (OF FILE) existing file name (IS) protection

"Protection" is an 18-bit octal number in Minisys. NOT IMPLEMENTED YET.

Errors: file name errors
YOU CAN'T PROTECT THE PROTECTION FROM YOURSELF

ACCOUNT (OF FILE) existing file name (IS) account

Errors: file name errors
DISK FILES ONLY
"?" if account must be numeric but isn't

DELETE existing file name

For disk files, this merely flags the file; until it is really deleted with the EXPUNGE command or by the operators, it may be UNDELETED. Version number defaults to <u>lowest</u> existing version.

Errors: file name errors
PROTECTION VIOLATION

UNDELETE deleted file name

Errors: file name errors
NOT DELETED

EXPUNGE (DELETED FILES)

Really deletes any "DELETED" files in the connected disk directory.

CLEAR (DIRECTORY OF DEVICE) device name

Erases all files on specified device; currently legal only for DECtapes. Confirmation mandatory.

Errors; " ? " if no such device (type 2)
DECTAPES ONLY
device: NOT AVAILABLE
device: ASSIGNED TO JOB n
device: NOT MOUNTED

MOUNT device name

"Mounts" a device such as a DECtape, that is, causes system to read directory and set a flag permitting access to files on the device. This is a subset of ASSIGN and should only be used when you wish to let more than one job access the device. If device is already mounted, no error results.

Errors: " ? " if no such device (type 2)
device: NOT A MOUNTABLE DEVICE
device: NOT AVAILABLE
device: ASSIGNED TO JOB n

UNMOUNT device name

"Unmounts" device, making access illegal until another MOUNT or ASSIGN is given. By making access impossible, using this command (or DEASSIGN) protects against potentially catastrophic screwups which can occur if a new tape, etc. is hung and accessed before its directory has been read.

Errors: " ? " if no such device (type 2)
device: NOT A MOUNTABLE DEVICE
device: NOT AVAILABLE
device: ASSIGNED TO JOB n
device: NOT MOUNTED

CONNECT (TO DIRECTORY) directory name [(PASSWORD) password]

Changes default disk directory name. Password input similar to LOGIN. Password may be omitted if directory doesn't have a password or if user is an ENABLEd WHEEL.

Errors: " ? " for no such directory (type 2)
INCORRECT PASSWORD

SHUT (ALL OPEN FILES)

Closes all open files which were opened by forks inferior to the Exec (ie by the program running under the Exec). Useful during program debugging.

Errors: none

JFNCLOSE jfn

Closes a (specified opening of) a specified file and releases the JFN. Allows closing a file left open by a program, eg during debugging. A list of assigned JFNs may be obtained with the FILSTAT command, below. Note: in most normal cases, the Exec closes all files and releases all JFN's used in Exec commands; files used by programs run under the Exec are closed by the commands GET, RUN, START, RESET, and SHUT.

Errors: "?" for no such assigned jfn

CLOSE (FILE) name of an open file

Closes all openings of the specified file in the current job, and releases all JFNs assigned to it. More convenient than JFNCLOSE. NOT IMPLEMENTED YET.

Errors: "?" for no JFN assigned to given file name

DIRECTORY [device:][<directory name>][file name]

If no argument and no subcommands are given, directory prints "name.extenstion;version", and ";T" for temporary files, for each file in the job's connected disk directory, on the primary output file (normally terminal), ordered as in directory.

If more than one version of a file exists, the name and extension are printed only once, with the several version numbers seperated by commas on the same line. If additional fields are being printed, this of course applies only when all printed fields are the same. Whenever the name or the name and extension are the same as those last printed, they are omitted and a few spaces typed in their place.

Variations may be specified with an argument and/or with subcommands. The argument can specify a device other than disk (the other legal device is DECtape, see below), a particular disk directory, and/or a specific file. It will later be extended to allow listing of only those files which have a given name, extension, account, etc.

Subcommand input is initiated by typing a comma immediately before the confirmation character (the comma may terminate the preceding field, or an altmode or space may intervene). The following subcommands apply when listing disk directories. Whenever a field is optional, the default value is given first.

These subcommands cause additional fields to be printed:

```
ACCOUNT
PROTECTION
SIZE (IN PAGES)
LENGTH (IN BYTES)
DATES (OF) [WRITE/READ/CREATION]
     give one DATES command for each date to print
TIMES (AND DATES OF) [WRITE/READ/CREATION]
VERBOSE        Prints account, protection, size in
               pages, dates of last write and read.
               This fits on one Teletype line.
EVERYTHING     Prints all of the above fields.
```

A heading is printed if any field beyond protection is requested. Protection and preceding fields are printed in standard TENEX file name string form.

The following determine the order of printout and are NOT IMPLEMENTED YET.

ALPHABETIC
CHRONOLOGICAL (BY) [WRITE/READ/CREATION]

Causes _inverse_ chronological printout unless "REVERSE" is given.

REVERSE Reverses order of printout in all cases.

The following end subcommand input and start listing:

START
carriage return

Miscellaneous subcommands:

```
DELETED (FILES ONLY)
OUTPUT (TO FILE) file name
             Default file name is "DIR";
             Default extension is ".DIR".
LPT          Short for OUTPUT (TO FILE) LPT:
NO (HEADING)
DOUBLESPACE
SEPERATE (LINES FOR EACH VERSION)
CRAM         Suppresses most spaces in printout.
             Faster, but does not produce columns.
```

Listing DECtape directories: A dectape directory can be listed by giving "DTA1:" etc as argument. No additional argument or subcommands are permitted (Though in the future a file name argument and some subcommands will be permitted). The format is identical to that of the 10/50 system.

Errors: " ? " if no such device (type 2)
device: IS NOT A DIRECTORY DEVICE
device: NOT MOUNTED
device: ASSIGNED TO JOB n
ILLEG DEVICE
"?" if name or extension given with DECtape device name
"?" if no such directory name
file name errors
SUBCOMMANDS LEGAL ONLY FOR DISK

The following can occur during listing if invalid information is found in a disk directory.

0 WORD FOUND IN DIRECTORY SYMBOL TABLE
0 FDB ADDRESS FOUND IN DIRECTORY SYMBOL TABLE
BAD BLOCK TYPE FOUND IN DIRECTORY
NO NAME POINTER IN FDB
NO EXTENSION POINTER IN FDB
FANCY PROTECTION
;ANONE if account information missing

QFD file name

Quick File Direcory. Primarily used to obtain a quick printout of all of the directory information about a single file. However, it will take all of the argument and subcommand options of directory -- it is actually equivalent to DIRECTORY plus the subcommands CRAM, EVERYTHING, and NO (HEADING). See DIRECTORY description above.

LIST (FILE) file name

Lists symbolic files on line printer, a la 940 UTILITY.

May be confirmed with alt mode. Takes sub-commands, in the same general form as directory, if argument is terminated with comma or the command is confirmed with a comma.

Unless a heading is specified or suppressed with a subcommand, each page of the listing is headed with the file name, date & time, and page number. The date & time is that of the last write for disk files; for other sources, it is the date & time the listing is begun.

The sub-commands are:

NOT IMPLEMENTED YET and yet to be specified.

Errcrs: file name errors
LPT: NOT MOUNTED (off line)
device: NOT MOUNTED
device: IS ASSIGNED TO JOB n
READ PRETECT VIOLATION FOR FILE name
name CAN ONLY BE APPENDED TO
FILE name BUSY

TYPE (FILE) file name

Same as LIST except default output file is the job's primary output file, normally the terminal, instead of the line printer. When using TYPE, it is convenient to terminate the file name with space or alt mode (as opposed to carriage return), then confirm the command with a form feed (↑L) so that the listing starts at the top of a new page.

File Group Descriptors -- a Future Extension

A future version of the Exec will allow a "File Group Descriptor" in the file name arguments of the Utility Commands. A File Group Descriptor is like a file name, except that the name, extension, version, and perhaps other fields may be replaced with *'s. In most contexts the *'s mean "repeat the command for every existing value of this field", but in certain output arguments they mean "copy this field from the corresponding input file name field". Also, in commands taking only one file argument, a list of such descriptors seperated by commas may be used.

Examples:

```
DELETE *.MAC                (deletes all versions)
LISTS *.MAC                 (lists latest versions only)
LIST *.MAC;*                (forces all versions)
LIST M.MAC,*.TEL,FOO.*
DIRECTORY DTA4:*.MAC,DTA5:,<STROLLO>;A11451
DIRECTORY <*>      (might list all directories, for operator)
COPY *.MAC (TO) DTA4:*.*
                   (puts files on dectape with current names)
COPY <MURPHY>*.MAC;A11451 (TO) *
                            (steals all of his 11451 files)
ACCOUNT (OF FILE)*;A11206 (is) 1
RENAME *.MAC (TO BE) *.BAK
RENAME (EXISTING FILE) VERYLONGFILENAME.EXT (TO BE) *.FOO
```

For user protection, commands which output to files, delete files, etc will require the user to confirm each name of a group specified with "*" before operating on that file. The command will type the file name, and [OLD/etc FILE] if pertinent, and await a carriage return to mean "yes" or an N to specify skipping to the next file. EG:

```
@COPY *.MAC (TO) *.BAK;*
 M.BAK;5 [NEW FILE]
 C.BAK;17 [OLD VERSION] N
 FOO.MAC;33 [NEW VERSION]
@
```

The following commands will permit *'s in their arguments to specify repetition for a group of files, and will take a list of descriptors seperated by commas: DIRECTORY, LIST, TYPE, DELETE, UNDELETE, ACCOUNT, PROTECTION, CLOSE.

COPY, APPEND, and RENAME will permit *'s (but not commas) in their first arguments to specify iteration, and in their second arguments, to specify field-copying. DEFINE is the same except that the arguments are reversed.

DEASSIGN will take "*" to mean "all assigned devices".

Implementation of these features requires implementation of JSYS's to input File Group Descriptors and to iterate over a group of files.

Subsystems and Programs

subsystem name

A subsystem name may be input without a preceding keyword -- whenever the first word of a command is not found in the Exec's tables, disk directory <SUBSYS> is searched for a file with the given name. If the file is found and the command comfirmed, the subsystem is run, as though "RUN <SUBSYS>name" had been input.

Care must be taken to prevent subsystem names from duplicating exec commands. There is provision in the Exec for making special table entries to prevent recognition of a command on seeing a substring of it which is also present in the subsystem directory, but it is undesirable to make such entries for strings longer than two characters. Also a subsystem name must begin with a letter or a digit and cannot begin with a string of digits followed by "/" or "\".

A subsystem is a save file in the directory <SUBSYS>; the default extension is .SAV.

Errors: "?" if file not found
file name errors

DUMP (ON) file name

Dumps environment on named file. Types [OLD VERSION]/[NEW FILE]/etc and requires confirmation after that even if a carriage return terminated the file name. NOT IMPLEMENTED YET.

Errors: file name errors
attempt to dump execute-only or proprietary program (these may come under "protection errors")
protection error or other error while opening file
device hung
data error
etc

SAVE (CORE FROM) n (TO) n, (FROM) n (TO) n,... (ON) file name

Saves part or all of the assigned memory of the fork inferior to the Exec in a non-shareable TENEX core save file (very similar to a 10/50 core save file). This type of file must be completely copied when it is retrieved. The program's entry vector is also stored in the file.

The lower and upper limits default to 0 and 777777 respectively. If an alt mode is typed directly after a comma, the Exec types out the noise word "(FROM)" rather than defaulting the next field as it does in most cases. This is because there is no other way to elicit the noise word, since an alt mode in place of the comma proceeds to "(ON) file name".

The default file extension is ".SAV".

Errors: NO PROGRAM
"?" for upper limit less than lower
file name errors

SSAVE (PAGES FROM) n (TO) n, (FROM) n (TO) n,... (ON) file name

Similar to SAVE, but saves indicated pages of assigned memory in a TENEX sharable save file. When activated with GET, RUN, etc, each page will be shared with the file and with other jobs using the file until written into, at which time a private copy will be made. SSAVE files thus speed up GETs and reduce system overhead due to page-swapping, yet can be used (non-optimally) even if the program isn't fully reentrant.

Errors: see SAVE

GET file name

Does a RESET (see below), then creates an inferior fork and reads the specified file into it. The file may have been created with SAVE, SSAVE, or DUMP. The default extension is ".SAV". The entry vector is also read from the file; the fork's special capabilities possible are transmitted from the Exec, but not normally enabled (see ENABLE, below).

Errors: file name errors

MERGE file name

Combines an additional file with a program previously GETed. Same as GET except does not reset first and does not update entry vectory unless current one is 0.

Errors: file name errors

RUN file name

Does a GET on given file name, then STARTs it.

Errors: file name errors
NO START ADDRESS

RESET

Eliminates program under Exec (ie kills all inferior forks) and closes all files opened by inferior forks.

The RESET function is the first step in the execution of the following commands: GET, RUN, subsystem name.

Errors: none

Program Control and Debugging Commands

START

Closes all files opened by forks inferior to the Exec, then starts program.

Errors: NO PROGRAM
NO START ADDRESS

REENTER

Errors: NO PROGRAM
NO REENTER ADDRESS

CONTINUE

Resumes execution of job after ↑C.

Minimum abbreviation is "CON", despite "CONNECT" and "CONVERT".

Errors: NO PROGRAM
PROGRAM HASN'T BEEN RUN
NOT INTERRUPTED

GOTO octal number

Starts program at specified location.

Errors: NO PROGRAM
NO SUCH PAGE
CAN'T EXECUTE THAT PAGE

octal number/

Examine location. No confirmation. User types location and slash (with no space or other characters between them!). Exec responds with tab, contents in N,,N form, and carriage return.

Errors: NO PROGRAM
NO SUCH PAGE
CAN'T READ THAT PAGE

octal number \ octal number
octal number \ octal number,,octal number

Deposit into location. The address precedes the backslash, the contents follows. The contents may be a single 36-bit unsigned octal number, or two 18-bit numbers seperated by space, alt mode, a comma, or two commas. Like most Exec commands, this must be terminated by or confirmed with carriage return.

Errors: NO PROGRAM
NO SUCH PAGE
CAN'T WRITE THAT PAGE

ENTRY (VECTOR LOCATION) octal number [(LENGTH) octal number]

Declares the location of the program's entry vector, which is the block of memory containing the program's entry point(s). Location 0 of the entry vector is the program's starting address; location 1 (if length greater than 1) is its reenter address. If omitted, length defaults to 1. To specify 10/50 compatible entries, give length 254000. The current entry vector is typed out by MEMSTAT, described below.

Errors: NO PROGRAM

TEN50 DDT

Transfers control to a 10/50 DDT loaded with the users program. Uses contents of JOBDDT (location 74) as start address of DDT.

Minimum abbreviation: "T D". Alt mode acceptable for confirmation.

Errors: NO PROGRAM
NO PAGE 0
CAN'T READ PAGE 0
NO 10/50 DDT LOADED WITH PROGRAM (JOBDDT 0)

DDT

Future command to transfer control to an invisble DDT. NOT IMPLEMENTED YET.

FORK octal fork handle

Changes the fork accessed by the following commands:

START
REENTER
GOTO
/
\
ENTRY
TEN50 DDT
SAVE
SSAVE
RUNSTAT
MEMSTAT
MERGE

The specified fork is used until another FORK command, or until a GET, RUN, RESET, or subsystem name. The JOBSTAT command may be used to obtain handles for all inferior forks, and print a listing of them. The 400000 bit may be omitted in the fork handle.

In the normal case, there will be only one subsidiary fork, or the additional forks will be managed entirely by the top subsidiary fork, and use of this command will not be necessary.

ENABLEd WHEELs may give "FORK 0" to operate directly on the running Exec, eg to SAVE a newly patched version.

This command should not be included in user documentation at this time.

Errors: FORK HANDLE MUST BE BETWEEN 1 AND 34
NO SUCH FORK

Also see SHUT, CLOSE, and JFNCLOSE above.

Primary Input/Output Redirection Commands

REDIRECT (INFILE) [existing file name/*] (OUTFILE) [file name/*]
(AND) [START/REENTER/CONTINUE]

Redirects primary input and/or output and optionally starts execution. NOT IMPLEMENTED YET

If an input file name is given, primary input is redirected to it, starting at the beginning of the file. If "*" is given, input is resumed from the last previously used input file, using the pointer position at which interruption occured. If the field is nulled (with carriage return, alt mode, or dash), primary input is not redirected.

Likewise for the output file argument.

The third argument permits starting the job without requiring an Exec command in the file. In conjunction with *'s for the file names, this is useful for resuming after typing ↑C.

Default file extensions: ".INP" and ".OUT".

Confirmation is mandatory if an output file name is given.

Errors: error in file name (" ? ")
"*" given when there is no previous input or output file (" ? ")
error in opening file
errors in START/REENTER/CONTINUE: see same.

COMMANDS (FROM FILE) existing file name

Redirects primary file input. Primary output file (normally terminal) is not changed; all input is echoed on it (whether this pseudo-echoing is done by the Exec or the monitor must be decided).

This subset of REDIRECT is NOT IMPLEMENTED YET.

Errors: bad file name (" ? ")
error in opening file

Information Printing Commands

The commands in this group do not require confirmation, and have no errors except as indicated.

AVAILABLE [LINES/DEVICES]

Types a list of free lines or of other free devices, depending on the second keyword, which defaults to LINES if omitted. In the DEVICES case a list of devices assigned to the current job is also printed. Login not required.

DAYTIME

Types current date and time. Login not required.

WHERE (IS USER) user name

Types out the terminal line number or "DETACHED" for each job logged in under the given name. Login not required.

Error: "?" if no such user

JOBSTAT

Types out the current job's TSS job number, the user's name, and the terminal line number. Will later type a table giving the job's fork structure, including the name of the program in each fork, but this is NOT IMPLEMENTED YET.

RUNSTAT

Types a very brief description of the state of the program under the Exec. Possible responses include:

```
NO PROGRAM
NEVER STARTED
↑C FROM RUNNING AT pc
↑C FROM IO WAIT AT pc
HALT AT pc
HALT: ILLEG INST AT pc (or other error condition)
↑C FROM FORK WAIT AT pc
↑C FROM SLEEP AT pc
```

USESTAT

Types: USED cpu time IN console time
and should later type any other chargeable resources used in this session.

FILSTAT

File statuses: types connected directory name if other than user name, and a table of assigned JFNs, names, what access open for (NOT OPENED, READ, WRITE, EXECUTE, APPEND, PROCEDURE, and/or PER PAGE TABLE), and condition (DATA ERROR, EOF). Also types a list of devices assigned to this job.

DSKSTAT

Types number of disk blocks (pages) in use in the currently connected directory,

SYSTAT

System status. Types system uptime, number of logged-in jobs, and a table showing TSS job number, terminal line number (DET if detached), user name (or NOT LOGGED IN), and subsystem name ("(PRIV)" for a private program) for each job. Job needn't be logged in.

MEMSTAT

If there is no fork inferior to the Exec, MEMSTAT types NO PROGRAM. Otherwise, it types the number of assigned pages, the entry vector location and length unless entry vector word is zero, and a memory map. The memory map shows, for each assigned page or group of adjacent pages, the page number(s), owning file name or fork number or "PRIVATE", page number(s) in owning file or fork, and access allowed: R=read, W=write, E=execute. Indirect pointers are indicated with an "@".

STATUS

Types:

THE STATUS COMMANDS AVAILABLE ARE:
JOBSTAT, RUNSTAT, USESTAT, MEMSTAT, FILSTAT, DSKSTAT,
AND SYSTAT.

Do not document this in future user documentation.

VERSION

Prints system name and version number, and Exec version number. This information should be included in all software trouble reports.

See also ERRSTAT and STATISTICS in the privileged commands section.

Miscellaneous Commands

LINK (TO TERMINAL) number

This won't be in Minisys.

Errors: no such terminal number (type " ? ")
that terminal has refused links
that terminal not logged in (?)

BREAK (LINKS)

REFUSE (LINKS)

RECEIVE (LINKS)

Terminal Characteristics Commands

HALFDUPLEX
FULLDUPLEX
TABS
FORMFEED
LOWERCASE
NO <TABS/FORMFEED/LOWERCASE>

The default terminal characteristics are full duplex with no tabs, formfeed, or lower case.

HALFDUPLEX will not be implemented in Minisys.

Login not required, alt mode acceptable for confirmation.

STOPS n,n,n...

Sets software tab stops. The default tab stops are at every 8th column.

[NO] RAISE

Controls conversion of lower case characters to upper case on input. RAISE permits you to type in lower case but have the characters be echoed and recived by the program in upper case. In contrast, [NO] LOWERCASE determines whether lower case characters output by your program are converted to upper case before being transmitted to your terminal.

QUIT

Halts the Exec, returning control to program it is being run under. If the Exec is not being run under another program, QUIT is illegal except for ENABLED WHEELS and OPERATORS.

Errors: NOT LEGAL IN TOP-LEVEL EXEC

Privileged Commands

The commands described here may only be used by those whose user names have one or more of the proper special capability bit(s) set. There are three special capability bits of interest here, WHEEL, which allows use of all privileged commands, OPERator, which allows use of a subset relevant to system operations, and CONFidential information access, which allows use of certain commands which print information but do not otherwise effect the system. With a few exceptions, these commands are prefixed by a non-printing control character which will be represented in this memo by "!". The ENABLE command must be given before most other privileged commands will be accepted (indeed, before the non-printing control character will be accepted). If a user who does not have the appropriate special capability (or who has not ENABLEd where required) attempts to give a privileged command, the Exec will type "?" as it would for any other unrecognized command, before even printing the rest of the word if an abbreviation of it was terminated with alt mode.

ENABLE

Enables right half special capabilities (WHEEL, OPERator, CONFidential information access) in the Exec and in its inferior fork (if any; effected by FORK command) and enables recognition of the non-printing control character which prefixes most of the privileged commands. Requires WHEEL, OPER or CONF special capability possible.

DISABLE

Opposite of enable; requires same capability.

ERRSTAT

Types information on recent system errors. Currently this includes only disk and drum errors. Confirmation not required. Requires WHEEL, OPER, or CONF special capability possible (but not necessarily ENABLEd).

STATISTICS

Prints out information on system loading, performance, etc. for example:

```
IDLE TIME 39% WAITING 28% CORE MGMT 2% PAGER TRAPS 2%
DRM READS 266272 WRITES 203734
 DSK READS 27903 WRITES 21432
81 PAGES OF USER CORE
15954 TERM WAKEUPS 820 TERM INTERRUPTS
40934367 TIME INTEGRAL OF  JOBS IN BALANCE SET
```

All numbers in the above are cumulative since the system was started. Also, a table of CPU time and page faults for each subsystem will be typed. Requires WHEEL, OPER, or CONF possible.

!SET (DATE AND TIME)

Causes Exec to request date and time, then set same in the running monitor. Requires OPER or WHEEL special capability ENABLEd.

!UNHANG device name

Performs a device-dependent function on the service routine for specified tape drive, etc, to make it available again after an error condition from which the software doesn't recover by itself. Requires OPER or WHEEL special capability ENABLEd. NOT IMPLEMENTED YET.

!LOGOUT (TSS JOB #) number (AFTER DUMP ON) [file name]

For eliminating unwanted detached jobs, crashed jobs, unauthorized jobs, etc. The default directory for the optional file name is that to which the specified job is connected; if a file name is given, the job's environment is dumped on that file before the job goes away. Requires OPER or WHEEL special capability ENABLEd. NOT IMPLEMENTED YET.

!ASSIGN device name

Assigns the indicated device to this job, with option to take it even if somebody else is using it, or to wait until it is free and take it. Probably this should be done with options on the regular ASSIGN command which come into effect for ENABLEd WHEELs and OPERs only. Requires WHEEL or OPER ENABLEd. NOT IMPLEMENTED YET.

!BROADCAST
message

Sends text beginning on next line and ending with alt mode to all terminals. Requires OPER or WHEEL ENABLEd. NOT IMPLEMENTED YET.

!NOACCOUNT

Turns off system accounting, for use during system checkout. Requires WHEEL special capability ENABLEd. NOT IMPLEMENTED YET.

!ACCOUNT

Turns acccounting back on, after !NOACCOUNT. Requires OPER or WHEEL special capability ENABLEd. NOT IMPLEMENTED YET.

!EDDT

Transfers control to a DDT looking at Exec, with symbols. Gets DDT if necessary from file <SUBSYS>UDDT.SAV, and stores symbol table pointer into it. Requires WHEEL ENABLEd.

!PRINT (NAME) name [VERBOSE]

Prints out the various parameters associated with a TENEX user name or files-only directory. Recognition is applied to the name. Requires WHEEL or OPERator special capability ENABLEd; also requests a superpassword before printing. Default-value parameters are suppressed as specified below unless "verbose" is entered after the name. Sample printout:

```
PASSWORD ABCD
DISK LIMIT 9766
WHEEL
DIRECTORY NUMBER 5
LAST LOGIN 22-SEP-70 12:37
USER GROUPS 1,2,3
DIRECTORY GROUPS 2
```

PRINT's complete "vocabulary" is:

```
PASSWORD password
DISK LIMIT decimal number     (suppressed if 488)
WHEEL                         (if privilege word B18 on)
OPERATOR                      (privilege word B19)
CONFIDENTIAL INFORMATION ACCESS (B20)
OTHER PRIVILEGE BITS octal number
        (other set bits, suppressed if none)
FILES ONLY                    (mode word B0)
ALPHANUMERIC ACCOUNTS         (mode word B1)
REPEAT LOGIN MESSAGES         (mode word B2)
OTHER MODE BITS octal number  (suppressed if none)
SPECIAL RESOURCE INFORMATION octal number   (if non-0)
DIRECTORY NUMBER octal number
DEFAULT FILE PROTECTION octal (if not 500000777752)
DIRECTORY PROTECTION octal    (if not 500000777740)
DEFAULT # FILE VERSIONS TO KEEP decimal number
                              (if not 2)
OTHER FILE RETENTION SPECIFICATIONS octal number
                              (if not 500000000000)
LAST LOGIN date time          (if any)
USER GROUPS n,n,n ...         (if any)
DIRECTORY GROUPS n,n,n ...    (if any)
```

See the description of CRDIR in section 10 of the BBN TENEX JSYS Manual for further description of the various parameters.

Errors: "?" for no such directory
"?" for bad superpassword

!CREATE (NAME) name [(PASSWORD) password]

Creates a new TENEX user name or files-only directory, or modifies parameters of an old one. Requires WHEEL or OPERator special capability ENABLEd and requests a superpassword.

After the name is entered, the Exec responds with [OLD] or [NEW]. After a carriage return, the command creates a new directory with all default parameters, or does nothing to an old one except update the password if one is given. A comma after the name or after the password causes subcommand input to be initiated after confirmation. Subcommands are used to specify non-default parameters for new directories or to change parameters of old directories.

The default characteristics for a new directory are all zero except:
    disk limit: 488
    default file protection: 500000777752
    directory protection: 500000777740
    default number of file versions to keep: 2

Several of the parameter words consist of independent bits, only some of which have assigned functions at this time. For these specific subcommands (WHEEL, FILES (ONLY), etc.) are provided for the already assigned bits, as well as general subcommands for changing any bits in the word. These subcommands are marked with an * and operate in a strange way described below.

The subcommands are:

NAME name
    for changing directory name; NOT IMPLEMENTED YET.

PASSWORD password for changing password; redundant except
    that it allows making it null.

DISK (STORAGE LIMIT) decimal number of pages

Privileges:
    [NOT] WHEEL (Privilege B18)
    [NOT] OPERATOR (Privilege B19)
    [NOT] CONFIDENTIAL (INFORMATION ACCESS) (Privilege B20)
    *[NOT] PRIVILEGES octal number

Mode:
    [NOT] FILES (ONLY) (Mode B0)
    [NOT] ALPHANUMERIC (ACCOUNTS) (Mode B1)
    [NOT] REPEAT (LOGIN MESSAGES) (Mode B2)
    *[NOT] MODE octal number

*[NOT] SPECIAL (RESOURCES INFORMATION) octal number

NUMBER octal directory number

PROTECTION (OF DIRECTORY) octal number

DEFAULT (FILE) PROTECTION octal number

DEFAULT (FILE) NUMBER (OF VERSIONS TO KEEP) decimal
sets B32-B35 of retention specifications word.

*[NOT] RETENTION (SPECIFICATIONS) octal

[NOT] USER (GROUP) decimal number

[NOT] DIRECTORY (GROUP) decimal number

The above two subcommands turn on or off one group bit; multiple subcommands must be used to change more than one bit.

KILL (THIS DIRECTORY)
NOT IMPLEMENTED YET

ABORT
Aborts this CREATE. ↑C does the same.

LIST [VERBOSE]
Prints out what !PRINT would print if this CREATE were completed. Highly recommended to check changes before ending the CREATE. Default-valued fields are suppressed unless "VERBOSE" is given.

carriage return
Ends subcommand input. After extra confirmation, the specified changes are put into effect.

The subcommands marked with "*" modify the existing value of the parameter. The bits set in the argument are set in the parameter (ie. the argument is OR'd into the parameter), or, if the subcommand is prefixed by NOT, the corresponding parameter bits are cleared. The bits not set in the argument are unaffected. The "existing value" means the current value for old directories, and the default value for new directories, both, of course, as modified by preceding subcommands.

Whenever an octal number is specified in the subcommands, it may optionally be typed in as two half-words separated by space, alt mode, comma, or comma-comma.

ERRORS: PASSWORD REQUIRED FOR NEW NAME UNLESS FILES-ONLY
YOU CAN'T CHANGE THE NUMBER OF AN OLD DIRECTORY
NUMBER ALREADY IN USE

Planned Exec Extensions

CONCISE COMMAND LANGUAGE

Eventual implementation of CCL commands that work with the 10/50 CUSPS and also with new subsystems is expected. The CCL scheme will use a command compiler built into the exec, rather than a special subsystem; it will put the compiled commands into exec memory rather than a disk file.

Two schemes of transmitting commands to the subsystems appear plausible at this time. First, an exec subroutine file could be created with a suitable name (such as PIP017.TMP) and directory entry. A 10/50 CUSP, or a new program coded following the conventions of DEC's CCL, would be started at its CCL entry and would read the file as it does on the 10/50 system. This scheme would eliminate some of the IO overhead of the scheme used in the 10/50 system.

The second scheme would be to create a subroutine file and redirect primary IO to it. This saves additional overhead as it would be unnecessary to make a directory entry or a directory search. When used with this scheme, old programs would be started at their non-CCL entries; new programs, not necessarily coded with CCL in mind could also be used. In this scheme, more work would be done in the exec and less in the subsystem. For instance, compiled commands requiring transfer of control to a different subsystem would be detected and executed in the subroutine file code rather than in each subsystem.

COMMAND FILES AND BATCH PROCESSING

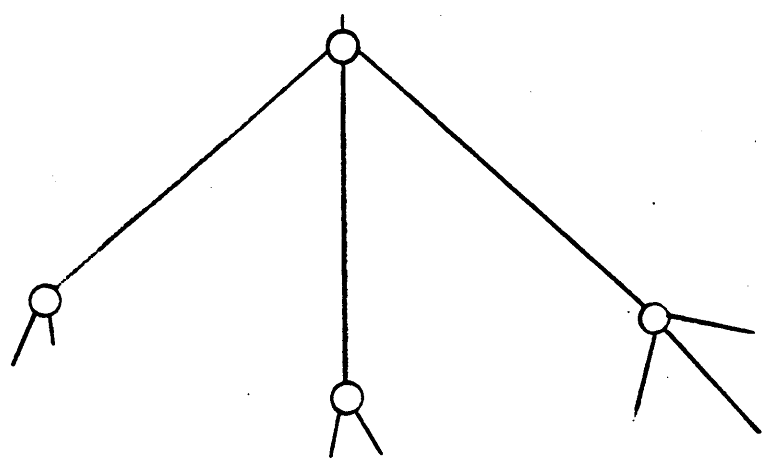
The TENEX exec will be capable of taking commands from a file. This file will contain ASCII text. Any of the command formats acceptable from the Terminal will be acceptable in the file, though it is expected that people will generally not omit noise words and will not use alt mode characters, because they will want the file to be easy to read and easy to edit. While a command file is in use, input characters will be output to the terminal, so a complete typescript can be produced. Use of a command file is initiated with a COMMANDS (FROM) command, or with a DETACH command. The latter takes input and output file names as arguments and causes the job to continue execution without a terminal.

The facility eventually envisioned will permit batch processing. The command file will be able to contain input for subsystems and private programs as well as for the exec, and it will be possible to direct all terminal output to a file for later listing. There will be provisions for suspending execution of jobs using non-conversational input files when an error is detected. Methods whereby errors will be detected by the exec will include: monitoring terminal output (which is going through a subroutine file) from 10/50 programs for lines beginning with "?" and the inclusion of special monitor calls in error routines of new subsystems.

Index

Fork Structure and Communication

TENEX permits each job to have multiple simultaneously runnable processes or forks. The fork structure is quite similar to the SDS-940 structure in that both parallel and subsidiary forks are allowed. The structure looks like an inverted tree. A fork always has one superior fork (except for the top-level EXEC fork), and may have one or more inferior forks. Two forks are said to be parallel if they have the same superior fork.



It is possible for a fork to create inferior (subsidiary) forks but not parallel or superior forks in the structure. A fork can communicate with other members of the structure by

(a) sharing memory
(b) termination, initiation, or suspension of any parallel or subsidiary fork.
(c) pseudo (software simulated) interrupts

Fork Accumulators

The accumulator values for a fork are in the hardware AC's when the fork is running and saved in the PSB when the fork is dismissed. Forks may access one anothers AC's through the RFACS, SFACS JSYS's.

Fork Structure Specification

The current fork structure of a job is recorded by a pointer structure in the monitor address space. This structure is started in the job storage block (JSB) and if unusually large, will grow into a separate full page. The JSB and any additional pages used for fork structure will reside in a contiguous 4K area of the monitor map of all processes. Every fork of a job will have a 12-bit basic identifier which when added to a base value, will address parallel tables in this 4K space. This block contains the relative pointers and all other pertinent data for the fork.

FIRST TABLE: FKPTRS

SUPERIOR POINTER	PARALLEL POINTER	INFERIOR POINTER

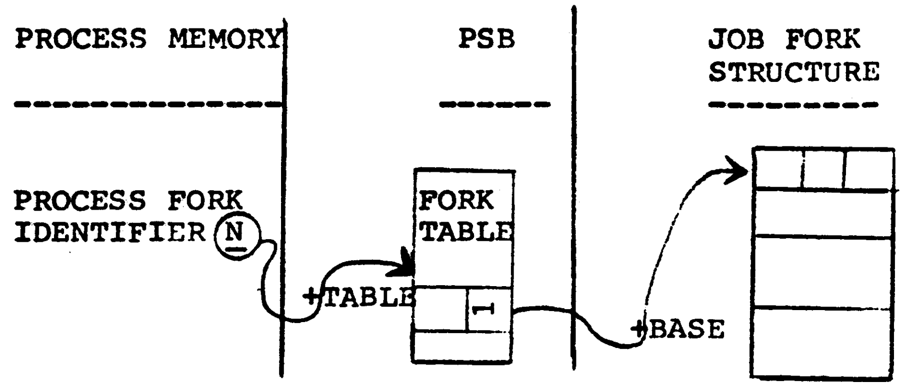
SECOND TABLE SYSFK

SYSTEM FORK INDEX OR -1

The 12-bit identifiers are used by the various monitor routines which operate on fork structures, but they are not given directly to user programs. Instead, when a fork is created, the creating process is given a small integer (plus 400000)* which is an index into a table in its PSB. The associated word in this table contains the job fork identifier. Entries in this table are 18 bits, 2 per word.

* The 400000 bit is on to distinguish fork handles from JFN's which can both be used as arguments to some JSYS's.

This is the mapping process:



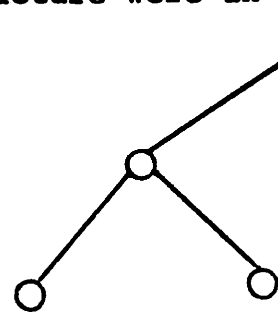
A process thus has a handle on any fork it has created, and this handle is used to reference that fork until deleted. Handles for the fork itself and the superior fork are also defined as numbers (fork table entries) 400000 and -1 respectively.

Fork handles may be passed between forks, by which means a fork may reference other than its own immediate inferiors or superior. Any particular handle is valid only within one process however (normally the process which created it), so if a handle H is passed between forks, it must be translated to a new handle H' which is valid in the receiving fork. The operation of translating a handle first identifies the fork being translated using the old handle H and the handle of the fork in which H is defined, and then adds an entry to the fork table of the receiving fork into which the new handle H' is an index. For example, an inferior fork N creates an inferior fork and gets the handle M. One can reference that new fork directly by first obtaining that handle (via shared memory, say) and then creating a new handle via a monitor call which says "get a handle on the fork in position M of the fork table of fork N".

A program may also cause an image of its parallel and inferior fork structure to be created in its own memory. Local fork handles will be created for all forks in that structure. The structure will consist of 2-word entries for each fork of the form

PARALLEL POINTER	INFERIOR POINTER
SUPERIOR POINTER	FORK HANDLE

The pointers will be 18 bit absolute addresses. For example, if this fork structure were in existence,



then the following would be created for the top fork.

```
BLOCK/   Ø        BLOCK+2   ;SELF. NO PARALLELS, INFERIOR PTR
         Ø        Ø         ;SUPERIOR NOT INCLUDED, SELF #Ø
BLOCK+2/ Ø        BLOCK+4   ;NO PARALLELS
         BLOCK    2         ;SUPERIOR PTR, NUMBER IS 2
BLOCK+4/ BLOCK+6  Ø         ;PARALLEL PTR, NO INFERIORS
         BLOCK+2  3         ;
BLOCK+6/ Ø        Ø         ;END OF PARALLEL LIST
         BLOCK+2  4         ;
```

Fork Creation and Control

Monitor calls are available which explicitly create and delete a fork. A fork is said to exist when there is a process storage block (PSB) assigned to it. Executing the create fork monitor operation will assign a PSB and add the fork to the fork hierarchy. The virtual memory map for a fork may be specified at the time the fork is created or at a later time. Other monitor calls initiate running of the fork or cause running to be suspended. A fork continues to exist and be potentially runnable until either it is explicitly deleted or its superior fork is deleted.

There will be other monitor calls which provide convenient and often used sets of fork operations. For example, a single monitor call may initiate running of a fork and wait for it to terminate. Another monitor call will effectively initiate the running of a named file (e.g. subsystem) under the current fork.

Fork Suspension

A process may be suspended (temporarily stopped) by one of several conditions

(1) The process's execution of an instruction which causes a hardware alarm (memory trap, instruction trap, etc.)
(2) The request for suspension by a superior process (e.g. the EXEC on receipt of control-C).
(3) Suspension of the superior process.

Pseudo-Interrupts

Certain conditions will cause a process to receive an interrupt, meaning that control will be transferred to specified locations called the pseudo-interrupt routines. Simultaneously, the process PC will be saved so that the pseudo-interrupt routine may resume the interrupted sequence upon completion of its tasks. These conditions are:

(1) Terminal Pseudo Interrupts--generated when selected terminal keys are typed.
(2) Illegal Instruction Traps (such as attempts to execute I/O instructions in ordinary user mode) or attempts to execute privileged monitor calls.
(3) Memory Traps including read, write and execute, directed traps, and un-assigned memory.
(4) Arithmetic Processor Traps
(5) Unusual File Conditions (EOF, errors)
(6) Specific Time of Day reached
(7) Generated Pseudo-Interrupts
(8) Subsidiary Fork Termination
(9) System Resource Allocation traps

There are 36 pseudo-interrupt channels and some number NPLEVS (currently 3), pseudo-interrupt priority levels. Priority levels are numbered from 1 to NPLEV. Ø is not a legal priority level. Most of the possible causes of PSI's are each permanently assigned to one of the 36 channels. The remainder are user-assignable to one of several of the 36 channels. Each channel can be activated or deactivated. If activated, the channel can cause an interrupt on the user-specified priority level. This two step interrupting procedure eliminates the need for user decoding of interrupt cause.

An interrupt for any channel will be initiated only if there are no interrupts in progress on the same or higher priority channels. Otherwise, it will be remembered and initiated when the last equal or higher priority channel de-breaks. Since a higher level (lower priority number) interrupt can interrupt a lower level PSI routine, there can be up to NPLEVS interrupts in progress simultaneously. The user's PSI routine exits with a monitor call which resets the interrupt-in-progress status of that PSI priority level.

The user can turn the pseudo-interrupt system on or off. When the system is off, interrupt requests are remembered until the system is turned back on except for certain "panic" channels (e.g. instruction trap, described below) where an interrupt request will take, as though the PSI system was always on. The user can also clear the entire interrupt system, thereby forgetting all stacked requests.

Channels vs. Priority Levels

Note carefully the distinction between channels and priority levels. A channel corresponds to one particular cause of interrupts. There is a one-to-one correspondence of channels and interrupt causes as shown in Table 1. Some interrupt causes are assignable to different channels, but at any given time, an interrupt cause is associated with at most one channel, and each channel is associated with at most one interrupt cause. The priority levels are provided to allow some interrupt conditions to be able to interrupt the service routine for other interrupt conditions.

For example, a program could assign a "break key" type of user request (e.g. LISP's control-H) to a low priority level, some APR overflow conditions to a medium priority level, and an "abort key" user request (e.g. LISP's control-E) to a higher priority channel. Thus, an unexpected or infrequently occurring APR condition (e.g. PDL overflow) could be handled during the program's main sequence or during a user initiated break sequence, but a user initiated abort request would override any of the above sequences.

As stated earlier, each activated PSI channel is user assigned to one of NPLEVS priority levels. The user makes his assignment by considering when one interrupt condition can arrive during the servicing of another, and when servicing of the later interrupt cannot be deferred until completion of the first.

Interrupt Service Conventions

Before using the pseudo-interrupt system, the user must execute a monitor call to specify the location of his channel table (CHNTAB) and level table (LEVTAB) in two half-words, i.e.

LEVTAB	CHNTAB

which the monitor will keep in the PSB. Then, for each channel activated, the user must set up the contents of location CHNTAB plus the channel number to contain:

Left half: number of priority level to which this channel is assigned.
Right half: address of interrupt service routine for this channel.

For any priority level specified by one or more of the above channel words, the user must set up the contents of location LEVTAB plus the priority number minus 1 to contain:

Left half: (Presently unused.)

Right half: location of word (in writable page) in which to store interrupt PC and flags.

When an interrupt is requested, the channel word (at location CHNTAB plus the interrupt channel number) is fetched. The left half specifies the priority level to be used. If this left half is Ø, or if the pseudo interrupt system for this fork is off or if an SIR has never been done for this fork, the system considers this fork is not prepared to handle a pseudo-interrupt on this channel and the pseudo-interrupt is changed to a fork terminating condition. If neither that level nor any higher priority level interrupt is in progress, the process PC will be set to the right half of the channel word. The old process PC will be stored as specified by the right half of the priority level word (at location LEVTAB plus the priority number minus 1), and the process will be run. When the interrupt routine is completed, it is dismissed with a monitor call which restores the process PC as specified by the right half of the priority level word, and the process is resumed.

There are some special conditions governing interrupts from monitor calls; these are discussed in Memo TENEX-8. However, if the service routine does not change the interrupt PC, all interrupts are guaranteed to be completely transparent, i.e. the fork will be resumed on de-break and will continue to do whatever it was doing. Note that, in general, the service routine must save any AC's or other temp storage possibly in use by the interrupted routine. The monitor protects temp storage in use by interrupted monitor routines (as described in TENEX-8), but the user is responsible for protecting all temp storage in user memory. If the service routine does change the interrupt PC, (in any way, even the flag bits) the de-break will cause the fork to be <u>restarted</u> at the location specified by the interrupt PC.

In one particular situation, interrupted monitor calls cannot be resumed but must be restarted. When the environment is saved and later resumed, any interrupted monitor calls will be restarted.

Panic-Channels

Certain channels including APR PDL overflow, file data error, illegal instruction, illegal memory read, illegal memory write, illegal memory execute, machine size exceeded, etc. are special "panic" channels in that they cannot be completely turned off. While they will respond normally to the channel on/off and read channel mask JSYS's, a pseudo interrupt request received on such a channel which has been turned off will be considered a fork terminating condition.

Implementation

The fork structure area contains one word for each fork to indicate pseudo-interrupt channel arming. Each bit corresponds to one of the 36 channels and if set means the channel is armed. Each PSB contains one word with a bit for each channel to remember a deferred request for a pseudo-interrupt (because of higher priority request or PSI system off). There is also a bit for each priority level to specify a pseudo-interrupt in progress on that level.

Pseudo Interrupt Fork Specification

When a particular pseudo-interrupt condition arises, one fork will be pseudo-interrupted. It is often not obvious which fork should be interrupted. For example, when a terminal pseudo-interrupt character is typed, it is quite possible that several forks may be armed for that pseudo-interrupt condition. The following rules specify which fork gets the various pseudo-interrupts.

1. Terminal Pseudo-Interrupts

   Up to 36 terminal keys may be used to specify pseudo-interrupts. Each of these may be armed in multiple forks, but when a fork arms a particular key, the assignment of that key passes to that fork alone. When that fork terminates or disarms the key, the assignment will be passed back to the fork from which it was taken. See the further discussion of terminal interrupts below for implementation details.

2. Directed Pseudo-Interrupts

   The generated pseudo-interrupts are directed to specific fork(s) which completely specifies the fork to interrupt.

3. Terminating Conditions

Some interrupt causes result from conditions indicating program malfunction and may be received only by the fork in which they occur. These include resource allocation exceeded, illegal instruction, file error conditions, and memory violation conditions. If one of these conditions arises and the corresponding channel in that fork is armed, then an interrupt will be initiated for that process. If the channel is not armed, the process will be terminated and the cause of termination reflected in the job status which is available to the superior fork.

4. Program Conditions

Other conditions arising from program execution including non-error file conditions (such as EOF), inferior fork termination, and APR traps (overflow, floating overflow, floating underflow, and no-divide) are handled as for number 3 above except that the process continues in sequence if the channel is not armed. The monitor will set the actual APR bits in a manner appropriate to each process each time the monitor begins to run the process.

5. Fork Termination

When a fork terminates, only the immediate superior will be checked for fork termination interrupt enabled.

Terminal Interrupts

There are a maximum of 36 codes which can cause interrupts. 18 of the 36 interrupt channels are capable of being used for terminal interrupts. Each of the 36 codes may be assigned to any one of these 18 channels. A channel may have at most, one terminal code assigned. The channels useable for terminal interrupts are Ø thru 5 and 24 thru 35. Each PSB contains a 36-byte table (PSICHA) to record the assignment of channel to code.

N		
↓	CODE Ø	CHANNEL #
	CODE 1	CHANNEL #
	CODE 35	CHANNEL #

Channel to use for code N

A second table is used to record the number of the fork having the code enabled before this one, i.e. the fork from which the assignment was taken when given to this fork.

CHAN Ø	CHAN 1		CHAN 35

Number of fork from which was taken the terminal code now assigned to channel N.

When a process is suspended, its code assignments will be passed back to the fork(s) from which they were taken. When it is restarted, the PSICHA table will be scanned and the codes re-assigned.

The JSB has a table (PSIFKA) of 36 bytes indexed by character code to record the number of the fork currently assigned for each code.

36 BYTES

Ø	1		34	35

Number of fork currently interruptable on receipt of code N.

The terminal service routine need maintain only a single word for each terminal, in which the 36 bits specify whether or not any fork of the attached job has the corresponding code enabled.

36 BITS	Interrupt of code N enabled anywhere.

This places a minimum time and space demand on the terminal service routine.

Table 1
INTERRUPT CHANNEL ASSIGNMENTS

CHANNEL	INTERRUPT
Ø	Terminal key or general
1	"
2	"
3	"
4	"
5	"
6	APR Overflow (includes NODIV)
7	APR Floating overflow (includes FXU)
8	Unassigned
9 (1)	APR PDL Overflow
1Ø	File Condition 1, EOF
11 (1)	File Condition 2, data error
12	File Condition 3, (un-assigned)
13	File Condition 4, (un-assigned)
14 (2)	Time of Day
15 (1)	Illegal Instruction (I>>)
16 (1)	Memory, Illegal read (MR>>)
17 (1)	Memory, Illegal write (MW>>)
18 (1)	Memory, Illegal execute (MX>>)
19	Subsidiary Fork Terminated
2Ø (1,2)	Machine size exceeded
21	Presently unassigned
22	Presently unassigned
23	Presently unassigned
24-35	Terminal key or general

Notes: (1) channel is a "panic-channel"
(2) NOT IMPLEMENTED YET

Perhaps to be added also are:
- I/O: Device full and device inoperative
- Memory: user-directed trap (MD>>)

This table was constructed in order of expected decreasing use of interrupt. The assumptions are:

1. One or two terminal interrupts (e.g. RUBOUT) will be used by most all programs.

2. It is decreasingly likely that programs will use:
   a. APR conditions
   b. File conditions
   c. Timer
   d. Instruction or memory trap conditions
   e. Fork termination
   f. Machine size or other allocation traps
   g. More than 6 terminal interrupts

Note that there are some channels indicated for general interrupts (the <u>initiate interrupt</u> monitor call). However, an interrupt may be explicitly initiated on any channel, whether or not assigned to some particular cause. User programs may occasionally wish to use this feature; normally explicit interrupts will be initiated on channels logically assigned for some independent purpose.

The particular monitor calls related to fork control and the PSI System are discussed in the JSYS manual, sections 5 and 6.

Monitor Calls and Pseudo-Interrupts

There are two types of monitor calls, UUO's and JSYS's. There are two classes of each of these, "fast" and "slow". "Slow" means that because of some additional overhead, the routine may be pseudo-interrupted and subsequently resumed. "Fast" means that the call will take sufficiently little time that pseudo-interrupt requests may be deferred until completion of the call, and that the additional overhead is undesirable and unnecessary.

The two classes of UUO's are distinguished by a bit in the left half of the UUO dispatch word. That is, if a UUO with opcode <u>n</u> is "slow", then

```
UUOT+n/   XWD 0,ADR
```

and if it is "fast", then

```
UUOT+n/   XWD 400000,ADR
```

If the bit is off, the UUO dispatcher will go to the UUO code via the "slow-call" setup routine (shown later), whereas control will be transferred directly to the UUO code for a "fast" UUO.

The two classes of JSYS are distinguished by virtue of the fact that the "slow" JSYS code contains an explicit call to the "slow-call" setup routine, whereas the code for a fast JSYS does not.

<u>Entry Procedure</u>

The "slow-call" setup routine, called MENTR (<u>M</u>ONITOR <u>ENTER</u>), is invoked from monitor code by execution of the instruction JSYS MENTR. Note that a user-mode program cannot execute this instruction with the same result because the effective address is greater than 1000(octal).

The following convention is observed:

> All user-executable monitor call instructions (JSYS and UUO) store their return PC in the same cell. It is called FPC and is located in the PS block.

This is necessary to insure correct action on pseudo-interrupt requests occurring during the execution of monitor code as will be demonstrated.

The "slow-call" setup routine maintains a stack containing returns and temp storage for slow monitor routines. When executing a user-to-monitor call, this setup routine places the push pointer in AC17, and saves the return. When executing a monitor-to-monitor call, the push pointer is assumed to already exist in AC17, so the MENTR routine need only add the return to the stack (ala PUSHJ).

Returns

The return from a fast JSYS or UUO would appear to be simply JRSTF @FPC, and it would be, except for the requirments of the pseudo-interrupt logic. The return of a slow monitor call is effected by the return routine MRETN (MONITOR RETURN) which performs the inverse function of MENTR.

Pseudo-Interrupts

Pseudo-interrupt requests can occur at any time. A PSI (pseudo-interrupt) request may be processed immediately if it occurs while the process is in user mode. When the interrupt request occurs during a monitor call however, it may be serviced immediately only if it can be guaranteed that:

1. Temp storage, including PC and AC's, in use by the interrupted call is protected from change by the user directly or by other monitor calls executed in the user's interrupt service routine. Otherwise, the routine may malfunction on being resumed, and, since it is running in monitor mode, could possibly destroy the monitor.

2. The routine can be aborted (by explicit user request) without leaving anything in inconsistent or transitory states.

Sometimes these conditions cannot be met, so a PSI request must be saved and serviced at a later time.

Interruptibility of "Slow" Monitor Calls

In order to meet condition 1 above, it is at least necessary that the temp storage in use at the time of the interrupt be identifiable. The most convenient way to do this is to establish a stack (push list) in the PS block to be used for all temp storage for all interruptable monitor routines.

This stack, then, along with the AC's and the process PC would represent the complete state of the process. When a PSI is requested, the AC's and PC are added to the stack and the stack pointer increased accordingly. This procedure effectively protects temp storage as required by condition 1. Additional monitor calls can be entered from the user's interrupt service routine, and additional interrupts can be initiated on higher priority channels to a depth limited only by the size of the stack and the number of priority levels.

The routines MENTR and MRETN handle the maintenance of the stack on entering and leaving "slow" monitor routines as mentioned above.

Fast-Slow Distinction

As can be seen, there is an overhead involved in the stack maintenance procedure, a cost greater than that of a simple JSYS-JRSTF call and return sequence. However, monitor routines which are so short that this overhead time is a significant fraction of their execution time are likewise so short that there is no problem in deferring interrupt service to their completion. This is precisely the distinction between "fast" and "slow" monitor calls.

Interrupting "Fast" Monitor Calls

Since fast monitor routines are by definition not in a state to be interrupted, it must be possible to save an interrupt request and service it at a later time, preferably at the termination of the fast routine. We propose to do this by making the return for fast monitor calls be done by executing the instruction XCT MJRSTF. The contents of MJRSTF will normally be JRSTF @FPC if there has been no pseudo-interrupt request. Since all returns are saved in FPC, this instruction is always the appropriate one. If there was a PSI request, the monitor's PSI control routine will have changed the contents of MJRSTF to JRST PSISVO which will again consider initiating an interrupt, assuming now that the process PC is specified by the contents of FPC.

PSI Strategy

The process by which the PSI routine decides whether to interrupt "immediate" or "deferred" is somewhat complex. The first decision factor is the state of the user mode flip-flop available in the process PC word. If the process to be interrupted is in user mode, the interrupt can be done immediately. If the process is in monitor mode, the PSI routine must distinguish "fast" vs. "slow" code. The flag SLOWF (in the PS block) makes this distinction. It is initialized to -1; entering slow monitor code (via MENTR) makes it positive, leaving returns it to its previous state. Therefore if SLOWF is negative (and process is in monitor mode), "fast" code is implied and interrupt request is deferred as described above.

One other flag, INTDF (INTERRUPT DEFER FLAG) is also included to enable "slow" routines to temporarily defer interrupts when aborting the routine would leave something in an inconsistent state (e.g. during a change to the PAC slot list structure). It is also necessary for some of the monitor-to-monitor calling sequences shown later. This flag is normally -1 (off, i.e. interrupts not deferred). It is turned on with AOS, and turned off with XCT INTDFF. INTDFF normally contains SOS INTDF if no interrupt is waiting, otherwise JSYS PSISV1.

Note that the routines at PSISV0 and PSISV1 do not necessarily initiate the interrupt whenever they are entered, rather they reconsider the state of the process as specified by the various flags and accordingly either initiate the interrupt or set up another defer trap and resume the sequence.

Monitor Routine Programming Considerations

System programmers writing monitor routines called via these sequences must be aware of the following points.

1. In some cases, an argument given to a monitor call is an address (in the user memory) which is to be referenced. To facilitate such references, the UMOV group of instructions (UMOVE, UMOVEI, UMOVEM, and UMOVES) has been installed on the APR and is described in the system reference manual. A more general way to reference the user map from monitor code is the XCT instruction with AC≠0. (See System Reference Manual section 10.5.2 for details).

   The effect of both types of instructions is to cause the user map rather than the monitor map to be used for certain memory references, whether direct or calculated.

   Example:

   ```
   UMOVE A,100    ;CONTENTS OF USER'S 100 => A

   UMOVEM B,0(A) ;B => ADDRESS GIVEN IN A
   ```

   Normal user mode addresses in the range 0-17 go to the fast AC's as do monitor mode addresses in that range. To facilitate general monitor references to user addresses, however, monitor mode references to user AC's (via UMOV or UXCT) are mapped into a block of 20 (octal) words determined by the AC base retister in the pager. The monitor maintains this register pointing to one of several blocks in the PSB and updates it each time a slow routine is entered or exited. For example, UMOVE 1,5 will move the contents of word 5 of the current AC block 0 to real AC 1. This means that a monitor routine which is given a user address (second example above) may reference that address without checking to see if it is an AC. Further, to receive or return a parameter in an AC, a monitor routine should use the UMOV or UXCT instruction thereby avoiding conflicts with AC's in use as temps.

   Example:

   ```
   UMOVE A,1      ;TO GET PARAMETER FROM AC1

   UMOVEM A,2     ;TO RETURN VALUE IN AC2
   ```

   Note however that the user AC's must be explicitly moved between the real AC's and the appropriate AC block when entering or leaving a monitor routine. We have decided to include in MENTR the saving, and in MRETN the restoring of the user AC's. The reasons for this are:

A. If the user AC's are to be saved at all, it is most efficient to do so in conjunction with setting up the PDL which MENTR already does. Further, this eliminated the need for a separate call or in-line code to do the save.

B. A routine already using MENTR can tolerate the additional overhead (30-40 usec) of saving the AC's <u>and</u> will probably need to do read references at <u>least</u>.

C. The need for PUSH's and POP's to save temp AC's is eliminated.

2. Monitor routines should use the slow-call procedure unless a good case can be made against it. In general, a fast routine must:

   a. Be less than 100 usec maximum execution time.

   b. Use <u>no</u> other monitor calls (private subroutines called by PUSHJ are OK)

   c. Not use a push list.

   d. Save and restore any AC's used, and avoid use of UMOV and UXCT instructions which could reference user AC's.

3. Monitor routines which are changing tables critical to the process should use the interrupt defer flag to prevent interrupts during a transition period. That is, execute AOS INTDF to become non-interruptable, and XCT INTDFF to restore interrupts. This prevents interrupts but does not affect scheduling, so the noschedule-schedule sequences must be used if the tables being changed contain job or system global data.

4. Note that a monitor routine may reference an address given it as a parameter (#1 above) with an indexed or indirect instruction with no special checks. The "call from monitor" flag in the APR records the state of the user mode flag in the <u>previous</u> context, and causes UMOV and UXCT instructions to reference monitor memory (except that addresses 0-17 always refer to the current AC block) when the calling program was in monitor mode. This flag is saved in the PC word and restored on an MRETN or fast return.

5. Argument and value conventions for UUO's (after effective address) and JSYS's:

AC1	First argument	First value
AC2	Second argument	Second value
..	..	..
..	..	..

Sample Routines

Following are the typical routines used for entering and leaving slow JSYS's and for decoding UUO's.

```
;UUO DISPATCH.ROUTINE

41/     JSYS UUOH               ;JSYS RATHER THAN JSR TO BE REENTRANT

UUOH:   XWD FPC,.+1             ;RETURN GOES TO FPC AS FOR JSYS'S

        MOVEM 1,XMENTR          ;AC1 => TEMP
        HLRZ 1,40               ;GET OP CODE
        LSH 1,-↑D9
        CAIL 1,100              ;CHECK FOR OUT OF BOUNDS
        JRST ITRAP              ; ILLEGAL INSTRUCTION
        SKIPL 1,UUOT(1)         ;GET DISPATCH WORD AND CHECK FAST OR SLOW
        JRST UUOH2              ; SLOW...
        EXCH 1,XMENTR           ;FAST, RESTORE AC1, SETUP DISPATCH ADR
        JRST @XMENTR            ;DIRECTLY TO ROUTINE
```

Comments:

At UUOT is a 100(octal) word dispatch table for UUO's with indicator bit in each left half as mentioned.

Slow UUO's exit with JRST MRETN, fast UUO's exit with XCT MJRSTF.

```
;SLOW-CALL SETUP ROUTINE

MENTR: XWD XMENTR,UUOH1      ;SLOW JSYS' BEGIN WITH JSYS MENTR

UUOH2:  EXCH 1,XMENTR        ;SLOW UUO'S ENTER HERE
UUOH1:  SETOM SLOWF          ;INIT SLOW STATE
        EXCH 0,FPC           ;GET RETURN PC
        TLNE 0,UMODE         ;USER OR MONITOR MODE?
        JRST MENT1           ;USER
        PUSH P,INTDF         ;SAVE CURRENT DEFER DEPTH
        PUSH P,MPP           ;SAVE PREVIOUS ERRORSET
        PUSH P,0             ;SAVE RETURN
        MOVEM P,MPP          ;SAVE CURRENT STACK POINTER
        AOS P,ACBAS          ;GET NEXT AC BASE ADR
        SETACB P             ;GIVE IT TO PAGER
MENT2:  MOVE 0,XMENTR        ;LOCAL RETURN => FPC
        EXCH 0,FPC           ;AC0 => 0
        SETZ P,
        XCTMU  [BLT P,P-1]    :MOVE FROM REAL AC'S TO USER BLOCK
        MOVE P,MPP           ;RESTORE P
        SETZM SLOWF          ;NOW IN SLOW CODE
        XCT MJRSTF           ;JRSTF @FPC OR INTERRUPT

MENT1:  MOVEM P,XMENT1       ;SAVE USER'S AC-P
        MOVE P,UPP           ;GET STACK POINTER
        PUSH P,Ø             ;TWO RETURNS
        PUSH P,Ø             ;SO ONE CAN BE DIDDLED
        MOVEM P,MPP
        SETOM INTDF          ;INIT INTDF
        MOVE P,ACBAS1        ;FIRST AC BASE TO USE
        MOVEM P,ACBAS        ;INIT AC BASE
        SETACB P             ;SET PAGER
        MOVE P,XMENT1        ;RESTORE USER'S AC-P
        UMOVEM P,P           ;PUT USER'S AC-P WHERE IT BELONGS
        JRST MENT2
```

Comments:

P is 17

Return is always last entry on stack, so skip return can be done by AOS Ø(P), etc.

When coming from user mode, additional procedure is to setup stack pointer, save original return (in case regular return is modified).

Interrupt requests occurring during this code will be deferred (to XCT MJRSTF).

```
;SLOW-CALL RETURN ROUTINE

MRETN:   SETOM SLOWF          ;RESET FLAG
         MOVE P,MPP           ;GET STACK POINTER AT LAST ENTRY
         POP P,Ø              ;POP RETURN
         MOVEM Ø,FPC          ;SETUP RETURN
         TLNN Ø,UMODF         ;USER MODE?
         JRST MRETN1          ;NO
         SETZ P,
         XCTUM [BLT P,P]      ;RESTORE USER AC'S
         XCT MJRSTF           ;RETURN OR INTERRUPT

MRETN1:  MOVEM P,MPP          ;SAVE P
         SETZ P,
         XCTUM [BLT P,P-1]    ;RESTORE AC'S
         SOS P,ACBAS          ;RESET AC BASE TO LAST ONE
         SETACB P
         MOVE P,MPP
         POP P,MPP            ;RESTORE PREVIOUS STACK POINTER
         POP P,INTDF          ;RESTORE INTERRUPT DEFERRED STATE
         SETZM SLOWF
         XCT MJRSTF           ;RETURN OR INTERRUPT
```

Comments:

This routine is like a POPJ with flag restoring. Interrupt requests occurring during this code will be deferred or immediate as for MENTR.

<u>Initiation of Interrupt</u>

When an interrupt is to be initiated from a monitor call, these PS block cells must be added to the stack:

UPP - Initial stack pointer; changed only for interrupt service (monitor to user transfer), set to current stack position at start of interrupt

4Ø, 6Ø - General UUO temps

SLOWF - Slow code level (flag)

FPC - Temp possibly in use by MENTR or MRETN
XMENTR - " "

Also, the following must be added to the stack:

AC's Ø-NSAC - presumed to be in use by mon code

process PC - pointing into monitor routine

Then:

1. Current stack pointer => UPP
2. Get user AC's (from UPGØ) and PC
3. Go to user's interrupt routine

When the user debreaks, the monitor routine will be resumed if the user did not change the interrupt PC, otherwise the stack will be cleared back one level (using the saved UPP), and the process will be started in user mode at the specified location.

Nested Monitor Calls

Monitor calls may be executed within other monitor calls, but extra instructions are required in some cases. There are four possibilities:

Slow to Slow

Same as user; save 40 if nested UUO

Slow to Fast

Become non-interruptable first, i.e.

```
AOS INTDF               ;DEFER INTERRUPTS
one or more fast calls
XCT INTDFF              ;SOS OR JRST
```

Fast to Fast

Save return on special stack, i.e.

```
MOVE AC,FPC
AOS FPTR                ;SPECIAL STACK POINTER
MOVEM AC,@FPTR
one or more fast calls
MOVE AC,@FPTR
SOS FPTR
MOVEM AC,FPC
```

Fast to Slow (Arising where fast routine wants to be conditionally slow)

Execute slow-routine entry procedure and observe slow routine conventions.

```
AOS INTDF               ;DEFER INTERRUPTS
JSYS MENTR              ;INITIALIZE STACK, ETC.
PUSH P,..               ;SAVE LOCAL TEMPS
..                      ..
XCT INTDFF              ;ENABLE INTERRUPTS.
```

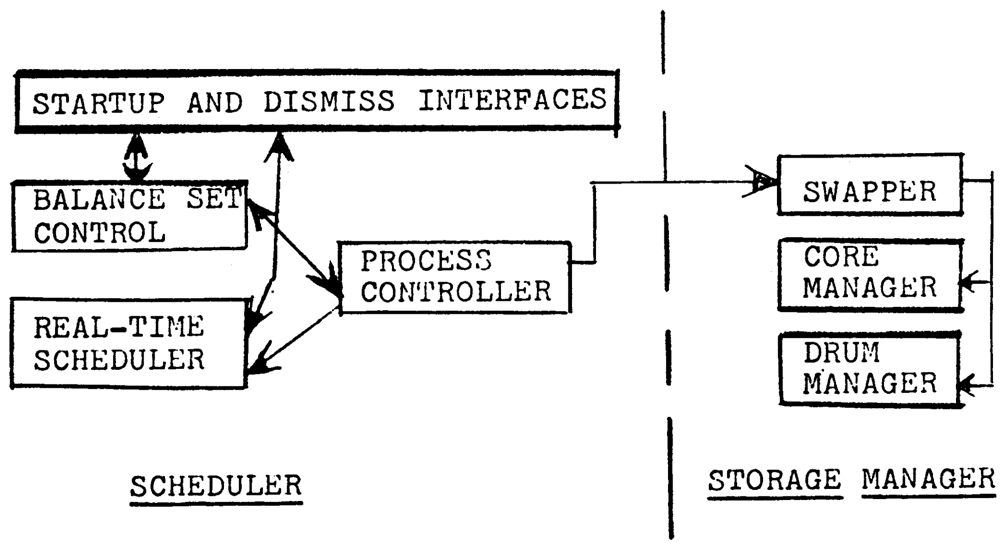
The routine is now effectively "slow", and should return with JRST MRETN. It can become "fast" again with the following kludge: (this is not done in current TENEX).

```
AOS INTDF               ;DEFER INTERRUPTS
POP P,..                ;RESTORE LOCAL TEMPS
..                      ..
POP P,TAC               ;ORIGINAL RETURN
PUSHJ P,MRETN           ;UNDO SLOW SETUP AND RETURN HERE
MOVEM TAC,FPC           ;REPLACE ORIGINAL RETURN
XCT INTDFF              ;ENABLE INTERRUPTS
```

This last case should be done only rarely and with extreme caution to be sure that there is not a higher level fast routine (in which this one is nested) which does not expect to be interrupted and which may have vulnerable temps.

Scheduling and Storage Management

The gross functions of scheduling and storage managing will be handled by a number of inter-related modules of TENEX monitor software, each with a specific, separable set of operations to perform.



The modules to the left of the dashed line comprise the scheduler, those to the right are the storage manager.

Scheduler

The process controller performs those functions usually associated with a time sharing scheduler. It contains tables of all processes existing in the system and their state of execution (runnable, blocked for I/O, etc.). It contains routines which change the state of processes on request from other system modules or as a result of process activity. A central routine of the process controller performs the basic scheduling function, i.e. it considers the state of the processes in existence and the available system resources, and selects a process to be given some CPU service. It keeps an accounting of the recent activity of each process, particularly CPU usage, and allocates each system resources among the processes competing for it according to some defined criteria.

The capabilities and requirements of TENEX impose the need for two other separate modules to handle specific parts of the total scheduling function. The <u>real-time scheduler</u> is concerned only with those processes which are currently making real-time demands on the system by use of the hybrid or display processors. Its scheduler portion is invoked whenever an external signal or clock indicates that re-scheduling may be required. Because the set of processes in its tables is small, it can very quickly determine which real-time process is to be run. If there are no real-time processes requiring service, then the selection of a process to run falls to one of the other two modules.

The real-time scheduler also communicates with user programs, accepting requests for real-time service, keeping track of the current demands, and informing the program whether service can or cannot be guaranteed.

The <u>balance set control</u> is concerned with making effecient use of the core and drum channel resources of the system.

It constantly monitors the state of core utilization and working set requirements of the processes in core, and decides when another process can be admitted or one must be thrown out. The "balance set" is defined as a set of runnable processes whose working sets can co-exist in core. It is thus a subset of the set of all runnable processes, and normally consists of those runnable processes which are most due for CPU service as determined by the process controller.

The information gathering and decision making procedures involved in determining working sets and core utilization are quite complex, and incorrect handling of these functions in a multi-process paged system can result in poor effeciency and bad service. The first step in avoiding this pitfall is to define a portion of the monitor which is directly responsible for these functions rather than having them diffused through many parts of the system. This we have done in the Balance Set Control module.

The function of the startup and dismiss routines is fairly common and straight forward. Included in this section are routines to save and restore environments as they go out of and into execution. No important scheduling or other decisions are made by this module.

Storage Manager

The swapper handles the communication between the secondary storage devices (drum and disk) and core memory. It receives requests from the scheduler to move processes into and out of core, constructs I/O requests and performs queuing.

The core manager selects core pages to be used for swap reads from the drum or disk, performs some "aging" operations, and handles the selection of core pages to be swapped to the drum. It has principal use and control of the Core Status Table (CST) which reflects at all times the current state of each page of core memory. The CST is also modified by the paging hardware, recording information about the activity of the running process.

The drum manager is responsible for assigning storage on the swapping drum and for selecting pages to be moved to the drum to the disk in the event the drum becomes full.

Design of Process Control Module

General Scheduling Algorithms

The CPU* is used at least some of the time by all user processes during the course of their existence, as well as by the monitor routines which control the basic behavior of the system. Therefore, we shall examine first the algorithm for allocating CPU usage among the various processes. Real-time considerations aside, there are two main goals which a scheduler attempts to attain:

1. Provide both rapid response to interactive users and "fair share" service to compute-bound users of the system. This usually means equal service for similar processes, but may be affected by externally defined priorities or privileges.

2. Make effecient use of the resources of the machine, principally CPU and core.

The actions of the scheduler affect the utilization of all of the resources in the system since all activity on behalf of a user is the result of the execution of instructions by the CPU. Hence the scheduling algorithm must include procedures to affect the scheduling as a result of I/O or

---

* or APR in DEC terminology

other non-CPU activity.

As time sharing systems have become more complex, the importance of goal 2 above has increased greatly. With all-core systems, any algorithm could easily be efficient, but with the addition of swapping and much larger processes, simple time-multiplex schedulers become extremely ineffecient because they do not take into account the limitations of core and swapping channels.

The TENEX paging concept is designed to allow the existence of much larger processes than would be possible or feasible without paging and to permit increased efficiency in handling all sizes of processes. This imposes an even greater demand on the scheduling procedures to inter-relate the use of core with the allocation of the CPU. Thus the development of the scheduling algorithms presented here will frequently include considerations of core usage.

Basic Scheduling Concept

The goal of quick interactive response suggests that processes which have a short amount of computing to do be given some preference over those with considerably more. Ideally, a time sharing system running N users would always be able to give interaction and compute times no worse than N times as long as if the system were running 1 user, and generally the distribution of types of activity of the N processes means that service can be better than this. To reach this ideal means that the system would have to have some idea of how much service was wanted when a request for service was made. For example, consider a system running five users. If a process completes a user interaction, will compute for 0.1 second and interact again, that process must be scheduled and run within 0.5 second elapsed real time. If the process is going to compute for an hour, however, it can easily go for minutes or tens of minutes without being run, just so it accumulates an hour of run time within the elapsed time of five hours.

This implies that the scheduler should know how much time a process is going to use when it makes a request for CPU service. To always do this correctly is obviously impossible. The scheduler can only guess at the future behavior of a process based on its past behavior, and in so doing it must assume that any guess can be completely wrong. It should guard against cases where a wrong guess or a strangely behaved process can produce gross inefficiency or unfair allocation.

The most significant piece of data from the recent history of a process is the amount of time it has used since its last request for service. In what way can this information be used? We know that within any short period of time, the more time a process has used, the closer it must be to completion. However, we know that the number of processes completing during any fixed period of time decreases as the total run time increases*, so the longer a process has run the less are the chances that it will complete "soon". Thus we have two conflicting ideas about how to predict time to completion.

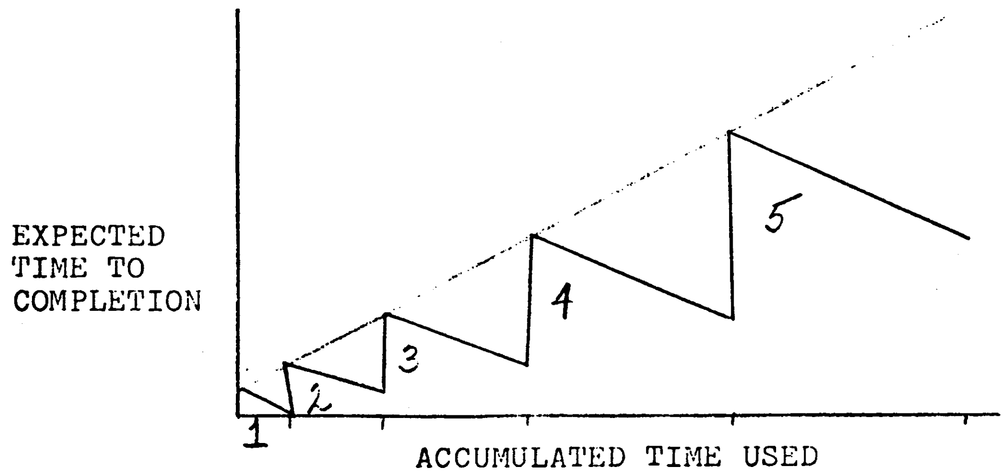
The first premise suggests a scheduler which always selects for running the process which has already run the longest. This means, however, the any new request for service would have to wait until all existing requests were completed (which could take hours), so the response characteristics of such a scheduler would be unsatisfactory. The second premise suggests a scheduler which always selects for running the process which has run the least. This procedure would provide quick response characteristics for short requests, but would cause constant rescheduling of the longer running processes as each process, when run, immediately surpassed the others in total time used.

We can find a middle ground by combining these two notions of process behavior. We say that over the long run, the

- - - - - - - - - - - -

* Statistically, for compute bound jobs, the time to complete is (roughly) exponentially distributed.

more time a process has used, the more time it may be expected to use, but we then break up the run into separate regions in which we say that the longer a process has run, the closer it must be to completion.



Note that each region (which we shall begin calling a <u>queue</u>) includes a longer time than the previous region by some factor. If our scheduling algorithm selects for running the process with the least expected time to run as determined by the above graph, then the following characteristics will be observed:

1. If two processes are widely separated in accumulated run time (are in different queues), the one with the lesser time will be preferred.

2. If two processes are closely spaced (are in the same queue), the one with the greater time will be preferred.

We do not extend the graph as shown indefinitely however for two reasons.

1. A process that had run a very long time (e.g. 1 hour) would get no service if another process began a long compute run until that second process had run nearly as long as the first. A long running process could also be shut out of service by a set of short running processes which used 100% of the CPU.

2. Although the frequency of rescheduling (and consequently the amount of rescheduling overhead) goes down as the queue time becomes large, a point is reached at which the overhead is an insignificant fraction of the total time and no gain is achieved by reducing it further.

Instead, we define a "last queue" (#5 in the above graph), and as a process reaches the end of the last queue, it goes back to the beginning of this queue. This means that we do not distinguish among processes that have run longer than a certain amount (typically 10-15 seconds) but schedule them in a "round-robin" manner.

Use of this graph results in a procedure which has three parameters.

1. The factor by which the time on each queue is greater than the last.

2. The amount of time of the first queue.
3. The number of queues.

The actual values are selected on the basis that fewer and longer queues result in less system overhead but produce a poorer approximation to ideal scheduling. Therefore, we select the largest values of 1 and 2 which give the desired response characteristics, and then a value for 3 as specified above. Our first set of values are:

1. 4 (i.e. T(I+1)=4*T(I) for queues I and I+1)
2. 64 msec for queue 1.
3. 5 queues (T(5)=T(1)*4↑ (5-1)=64*256=16.384 SEC.)

They may easily be changed, however, and we expect to experiment with different combinations. In fact, the Mini-System is implemented with a table which gives the time of each queue, so #1 above need not be a constant but can vary from queue to queue.

To see further how this algorithm works, consider a process which has just made its first request for service. It is placed on queue 1 with a 64 msec quantum. This process will be serviced before any processes on higher queues, but since this is the largest quantum on queue 1, any other processes on queue 1 will receive service first. However, there can be at most N-1 other processes on that queue, so this last process will receive service within 64*(N-1) msec. If the process uses all of its 64 msec, it will be placed on queue 2 and given a new quantum of 256 msec. Now it must wait for other processes on queue 2, and when running may be pre-empted by processes appearing on queue 1. So long as it continues to demand CPU service, it will fall to lower priority (higher numbered) queues until it reaches queue 5. Then, each time it uses the 16 sec quantum, it is placed back at the beginning of queue 5 and given another 16 sec. Thus, the scheduler will "round-robin" any set of long running processes, giving each 16 seconds of CPU service before passing on to the next.

Waits

We must now add to the algorithm the procedures for handling periods of no-CPU demand by processes. If a process is not demanding CPU service, it is explicitly or implicitly waiting for some external condition or event, e.g. an I/O device to complete or a user to type a character. For CPU scheduling purposes, we can say that it does not matter what causes the waiting, we can still provide at least 1/N of the CPU to all users by the following procedure:

> During periods of I/O wait, give the waiting process "credit" for CPU time not used by reducing the time-used value at the rate of 1/N. Reducing the time-used quantity will tend to move the process to

higher priority queues so it will be preferred over processes which continue to run.

Some of the effects of this are:

1. A process which waits long enough will have its run-time reduced to 0 and so will receive highest priority when it again requests service. This will tend to happen to processes which are doing much teletype interaction and only short compute bursts. However,

2. A process cannot grab more than its fair share of the CPU by doing lots of interactions. This happens in many systems because I/O waits erase all previous history of the process, and generally, long or short waits are treated as equivalent.

3. Use of high-rate I/O devices will not "swamp" the system or lock out ordinary processes.

This procedure does not include waits occasioned by disc or drum transfer because processes cannot directly request such transfers; they arise only indirectly or as a result of scheduling decisions. When a process cannot be run because a needed page is not in core, this is <u>not</u> considered a wait because in fact the process is still demanding CPU service. The service cannot be given because core, rather than CPU in this case, is not available. Handling this part of the scheduling function is the process of the balance set control module described in the next section.

Implementation

The foregoing scheduling algorithm will be implemented in the TENEX process controller. This part of the scheduler is responsible for continuously monitoring the activity of all processes on the system and maintaining an equitable distribution of CPU service, generally to attain the 1/N compute and response rate described earlier. By application of the algorithms discussed, the process controller can at any time establish the preferential order of all processes to receive CPU service. Whenever an event occurs which could change that ordering, the process controller again checks the state of the processes to see if the currently running process is now less preferred than some other process. Such events include:

1. Process in I/O wait becomes runnable.
2. Currently running process blocks for I/O wait.
3. Currently running process exhausts time allocation for its current queue.

Note that the algorithm as presented is not limited to scheduling only one process to run at a time. The second process, third process, and so on can be selected from the preferential ordering for simultaneous running so long as processing capability exists. This is important for two reasons:

1. The complete TENEX system will contain two CPU's and must be capable of running processes on both simultaneously.

2. Because of the paging mechanisms, portions of several processes will exist in core simultaneously, and these processes may be switching rapidly between the states of runnable and page fault wait. It is useful to consider these processes as running simultaneously from the point of view of the process controller. The final decision of which of these processes to actually run can then be made by the balance set control using additional information and procedures which are tuned for high core and CPU efficiency.

Therefore we say that in general there are a set of N processes running and these are the top N processes of the preferential ordering. The rescheduling procedure, triggered by one of the events listed above, causes to be removed from the running set any process which is now of lower preference than any process outside of the running set. In the next section, we shall call this running set the <u>balance set</u>, indicating that it is a set of processes whose core and CPU demand balances the available core and CPU resources of the system.

Balance Set Control

As stated in the introduction, the balance set control is responsible for making efficient use of core memory. The existence of paging makes this a critical task and one which should be centered in a specific module. The logical storage organization of TENEX includes the drum and disk as well as core memory, i.e. core, drum, and disk are part of the mechanism which implements the virtual memory and file capability of TENEX. This means that these devices and their associated channels are servicing the demands of many users either simultaneously or over short intervals, and making efficient use of core is closely related to making efficient use of the data channels to the drum and disk. This is why we have placed the emphasis on core utilization and have specified that drum and disk waits are not handled like other I/O waits.

The basic functions of the balance set control are:

1. Maintain the list of processes in the balance set such that the working set of all of these processes can exist in core.

2. When the running process must be stopped for a page fault, select one of the other processes in the balance set for running.

3. On occurrance of a resheduling event (quantum overflow, I/O block, I/O unblock), remove and/or add processes to the balance set in co-operation with the process controller.

We believe that even simple algorithms for handling these functions, working in conjunction with the process controller described in the last section, will provide reasonable efficiency. The design of the balance set controller has not been firmly decided at this time, and we expect the Mini-System implementation to be fairly simple. We will then experiment with more complex procedures and different algorithms to improve system performance. We believe that the organization of the scheduler into the modules shown on page 1 means that such experimentation will be effected easily and that our first simple implementation will function satisfactorily. The following discussion will attempt to clarify the operation of the balance set control and give some of the algorithms which have been proposed.

Function 1 - Maintain Balance Set

As stated earlier, the paging mechanism allows portions of several processes to be in core simultaneously. Determining how many processes of what size is the central function of the balance set control. Usually, an estimate of the working sets of the running processes must be maintained. The working set model of program behavior is developed and discussed in an article by P. J. Denning in the Communications of the ACM, May, 1968, to which the reader is referred for an extensive treatment of this subject. Basically, the model suggests that there is a relationship between the number of pages of a process in core and the average time that that process will run before page faulting (referencing a page which is not in core).

Control of this time to page fault, T, is critical to the efficient use of core and CPU, because what the balance set control tries to do is make sure that there is always at least one process to run (page swap completed) whenever the running process page faults. It is obvious that T is an increasing function of the number of pages in core. The end points are clearly T=0 for 0 pages in core, and T=infinity for all pages of the process in core. Several suggestions for the shape of this curve in between have been given, and different programs probably have differently shaped curves. The working set is defined as that number of pages which will cause Tav to be large enough so that the process can do some useful computation between page faults.

The balance set control must try to keep a balance set which maximizes the probability that there will always be at least one process to run. That is, whenever one process page faults, there should be another ready to run. This suggests that the processes must run an average time greater than the average interval, W, over which one page transfer will be completed for one of the page-waiting processes, i.e. T> W. For example, if there are exactly two processes in core, then one must run for at least as long as it takes to swap a page for the other. Swapping time is <u>write</u> <u>access</u> plus <u>write</u> (to put the page being replaced back on the drum) plus <u>read</u> <u>access</u> plus <u>read</u>, or, on the average,

W = 2 * (AVERAGE ACCESS + TRANSFER)

for fixed size (one page) transfers. W may be less if the write operation need not be done because the page was not changed while in core.

As the number of processes in core (and waiting for a page) is increased, the average time to completion of a page swap decreases, since there can be several processes completing during one drum revolution. There are two limits to the increase however. First, the number of pages that each process can have will decrease (as the fixed number of pages of real core are divided among more processes) and so Tav will fall, eventually to a value below the CPU process switching time. Secondly, the drum has a maximum data transfer rate which is reached when every sector has a transfer waiting. That is, if the drum has S sectors and a rotation time of R, there can be at most S pages transferred and therefore S processes completing during the next R seconds(1). Thus the minimum approachable average time for processes to complete page waits is between R/S and 2R/S, depending on what fraction of pages are changed while in core.

So we see that there is a range into which W will fall, the maximum

$$W = 2 * (R/2 + R/S) = R(S+2)/S = \text{(approx) } R$$

when there is one process waiting and pages must always be written, and the minimum

$$W = R/S$$

when there are many (> > S) processes waiting and pages never need to be written.

One possible algorithm is to estimate a value of W based on N, the number of processes in core, e.g.

$$W = R/N \quad \text{for } 0< N< S$$

then attempt to adjust the size of the processes in core so that Tav for each one (measured dynamically) is slightly greater than W. This adjustment could cause the allocation

- - - - - - - - - - - - -

(1) This can only be approached by having at least one process waiting for a page on every sector, which means considerably more than S total processes waiting. However, this situation also means that some processes will be experiencing delays of several revolutions in receiving a page transfer, and response time may become unacceptable.

of core to change so that there would be room for one more or one less process, whereupon W must be re-estimated. Note that, for example, if T< W causes core to become full and one process to be thrown out, then the new W will be slightly larger than the old W, thus moving in the desired direction of T> W. This means that the iterative procedure should tend to converge, with suitable precautions against oscillations around T=W.

Function 2 - Reschedule on Page Fault

As described above, the balance set control control attempts to always have at least one process to run whenever the currently running process page faults. If there is more than one process ready, the balance set control must select one for running, and we may ask what algorithm should be used for this. One that has been suggested is that the process with the largest number of pages in core (largest working set) be selected, reasoning that the process tying up the most resources should be pushed to completion so as to free the resources for other use. A second procedure would be to schedule on the basis of the preference value determined by the process controller for similar reasons.

Whether one of these or some other algorithm is used, we may also ask if rescheduling should be done only on page fault or on page wait completion also. That is, should the balance set control possibly reschedule when the swapper signals that a page transfer has been completed or only when the running process page faults?

Function 3 - Change Balance Set upon rescheduling event

The three regular rescheduling events affect the balance set control in similar ways.

1. Quantum overflow - If the currently running process exhausts the quantum for its queue, it may have become of lower preference than some other process not currently in the balance set. When quantum overflow occurs, the balance set control will call the process control to establish the new preferential ordering which will indicate whether the quantum overflow process is to be thrown out and one or more new processes admitted.

2. I/O block means that the process <u>must</u> be removed from the balance set because it cannot run. Space is then available for one or more new processes.

3. I/O unblock - This event is detected by the various I/O service routines which then initiate a request for rescheduling. If the unblocked process is now preferred over one or more processes in the balance set, then the unblocked process must be added and one or more of the other processes removed.

There are some other aspects of adding and removing processes to consider. We believe that the loading of a new process can be handled strictly by demand page faults rather than by any scheme of preloading pages. If the balance set control is successful in keeping several runnable processes in core, then the rapid page faults of one process loading its working set will not cause a degredation of efficiency. Also, the disadvantages of preloading will be avoided, which are:

1. Need for storage and procedures to record actual pages of working set for each process.

2. Extra load on drum channel caused by pages which are preloaded but not needed.

Demand loading insures that any page loaded is actually needed. However, only one or a few processes should be in the demand loading phase at any time to avoid reducing the Tav for all processes too seriously.

One provision which could serve to improve efficiency is to record for every process the <u>size</u> (number of pages) of its working set and its Tav at last running. Then, when a process is to be brought into core, the balance set control would know how many pages of core are needed and consequently how many other processes are to be thrown out. Further, when room in core becomes available (because of I/O waiting for page transfers. This measurement will be maintained by the scheduler itself and will be one of our most closely watched indicators of system performance.