Model 990 Computer
DX10 Operating System Release 3.4
System Design Document

Part No. 939153-9701 *C
1 October 1981

TEXAS INSTRUMENTS

LIST OF EFFECTIVE PAGES

INSERT LATEST CHANGED PAGES AND DISCARD SUPERSEDED PAGES

Note: The changes in the text are indicated by a change number at the bottom of the page and a vertical bar in the other margin of the changed page. A change number at the bottom of the page but no change bar indicates either a deletion or a page layout change.

Model 990 Computer DX10 Operating System Release 3.4 System Design Document (939153-9701)

Original Issue ..... 15 December 1977
Revision ..... 15 December 1979
Revision ..... 15 April 1981
Change 1 ..... 1 October 1981

Total number of pages in this publication is 302 consisting of the following:

PAGE NO.	CHANGE NO.	PAGE NO.	CHANGE NO.	PAGE NO.	CHANGE NO.
Cover	1	6-1 - 6-8	0	9-15 - 9-18	0
Effective Pages	1	6-9 - 6-11	1	9-19	1
iii - v	0	6-12 - 6-17	0	9-20 - 9-28	0
vi	1	6-18	1	10-1 - 10-22	0
vii - viii	0	6-18A - 6-18F	1	10-23/10-24	1
ix - x	1	6-19 - 6-50	0	A-1 - A-18	0
xi - xii	0	6-51 - 6-55	1	B-1 - B-6	0
1-1 - 1-40	0	6-56 - 6-60	0	C-1 - C-12	0
2-1 - 2-4	0	6-61	1	D-1/D-2	0
3-1 - 3-6	0	6-62 - 6-66	0	E-1 - E-2	0
4-1 - 4-7	0	6-67	1	F-1 - F-14	1
4-8	1	6-68	0	User's Response	1
4-9 - 4-27	0	7-1 - 7-2	0	Business Reply	1
4-28 - 4-31	1	8-1 - 8-4	1	Sales and Service	1
4-32 - 4-46	0	9-1 - 9-12	0	Cover	1
5-1 - 5-4	0	9-13 - 9-14	1		

PREFACE

The purpose of this manual is to familiarize the reader with the flow of control between major parts of the DX10 operating system, with the internal data structures used, and with the organization of the system disk.

The manual is organized into the following sections:

1. DX10 Implementation Tutorial -- Describes the general concepts used throughout DX10 and control paths through major parts of DX10.

2. Organization and Structure of DX10 Source Libraries -- Describes the organization of the DX10 source disk, and includes a disk map.

3. System Loaders -- Describes the software necessary to start up a DX10 system, including the boot loader, disk loader, DX10 loader, and system restart task.

4. Disk Organization -- Describes the physical and logical format of a DX10 disk, as well as the internal structure of all file types that are supported.

5. System Files -- Describes some special files used by DX10.

6. Data Structures -- Describes many of the internal data structures built and maintained by DX10.

7. DX10 Data Base Modules -- Describes the information contained in the two data modules within DX10.

8. Common System Routines -- Names and describes the stacking and queueing routines used by many of the system routines.

9. DX10 Source Modules -- Contains a tabularized description of the most important modules in DX10.

10. System Command Interpreter -- Describes the separate functional parts of the system command interpreter (SCI) and the flow of control between those parts.

This manual also includes appendices that describe how to analyze system crashes; how to regenerate DX10, SCI, SDSMAC, and the link editor from the source; the scheduler structure and operation; the effect of device states on LUNO assignment; VDT character input; and operation of the system level debugger.

This manual assumes that the user has a detailed, working knowledge of the information contained in the following Model 990 Computer Manuals:

DX10 Operating System Concepts and Facilities Manual	946250-9701
DX10 Operating System Production Operation Manual	946250-9702
DX10 Operating System Application Programming Guide	946250-9703
DX10 Operating System Developmental Operation Manual	946250-9704
DX10 Operating System Systems Programming Guide	946250-9705
DX10 Operating System Error Reporting and Recovery Manual	946250-9706

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

APPENDIXES

LIST OF FIGURES

LIST OF FIGURES (continued)

LIST OF TABLES

SECTION 1

DX10 IMPLEMENTATION TUTORIAL

1.1 GENERAL CONCEPTS

The DX10 operating system is physically divided into two parts: one is memory resident and the other is disk resident. Memory resident DX10 includes:

- ...System tables and device buffers
- ...System overlay areas
- ...Task scheduler
- ...Task loader
- ...Overlay loader
- ...System overlay loader
- ...XOP processors
- ...Most SVC processors
- ...Interrupt processors
- ...Some system tasks that may have overlays (e.g., disk manager, file manager)

These parts are linked together when the operating system is generated, and loaded into memory when the system is booted, forming the nucleus of DX10.

Disk resident parts of DX10 include system overlays and system tasks. System overlays are loaded into the system overlay areas reserved within the memory resident nucleus. System tasks are loaded into available memory and are mapped in (i.e., share memory) with the nucleus.

Figure 1-1 shows a simplified view of DX10 physical organization.

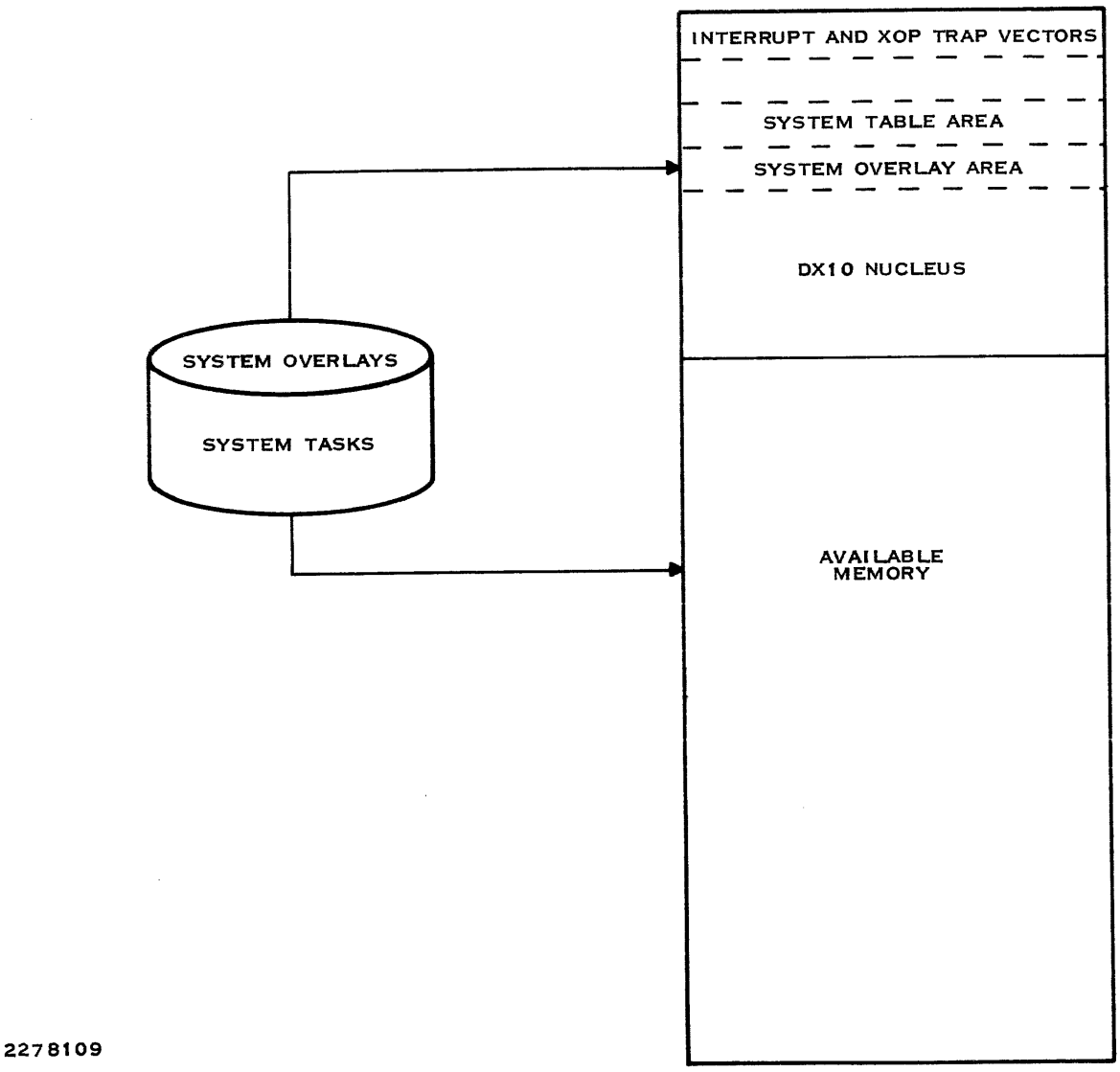


Figure 1-1 DX10 Physical Organization

The following paragraphs describe several specific concepts utilized in DX10.

1.1.1 QUEUE SERVERS AND ACTIVE TASK QUEUES.

An important concept in DX10 is that of information queues and queue servers. A queue is a first-in, first-out list of data to be processed. In DX10, each queue consists of a queue anchor, located in the memory-resident nucleus, and the queued blocks of data. Each block is linked to the next block in the queue (see Figure 1-2). A queue server is a task that is dedicated to the processing of the data blocks in its associated queue. For example, the Bid Task SVC processor is a disk-resident task that is a queue server. The queue entries are buffered supervisor call blocks.

A queue server operates in the following manner: when an entry is placed in the queue, the queue server is activated (bid). The queue server, operating as a task under DX10, dequeues an entry from its queue and processes it. The queue server continues to dequeue and process queue entries until the queue is empty, at which time it suspends itself by issuing a Suspend Awaiting Queue Input SVC (code >24). When a new entry is placed in the queue, the queue server is activated.

Nearly all of the functions of DX10 are performed by queue serving routines. Many SVC processors (e.g., I/O SVCs, Install Task, Kill Task) plus all of the disk-resident SVC processors are queue servers.

NOTE

Disk-resident queue servers are pseudo-memory resident. When such a task terminates awaiting queue input, its memory is not released until it is required to load other tasks. At that time, the memory is released and the task must be reloaded the next time it is bid.

The active task queue keeps track of the active tasks wihin the system. It is organized in priority order (i.e., all tasks of the same priority level are grouped together in a list), starting with the highest priority level at the top of the queue. The top task on each priority list is the oldest task on that list, and the bottom task is correspondingly the youngest. The scheduler always causes the top task on the active queue to be in execution. The sequence of task execution is dependent on the placement of tasks on the appropriate list within the queue. Figure 1-3 shows how the active task queue can appear. A list is maintained on the queue for each priority level. Figure 1-3 shows a queue in which priority levels R1 through R38 are void, level R39 has two tasks, R40 has one task, and level R41 has three tasks. The remaining real time priorities are void with level 1 having five tasks, level 2 having three tasks, and level 3 having 5 tasks.

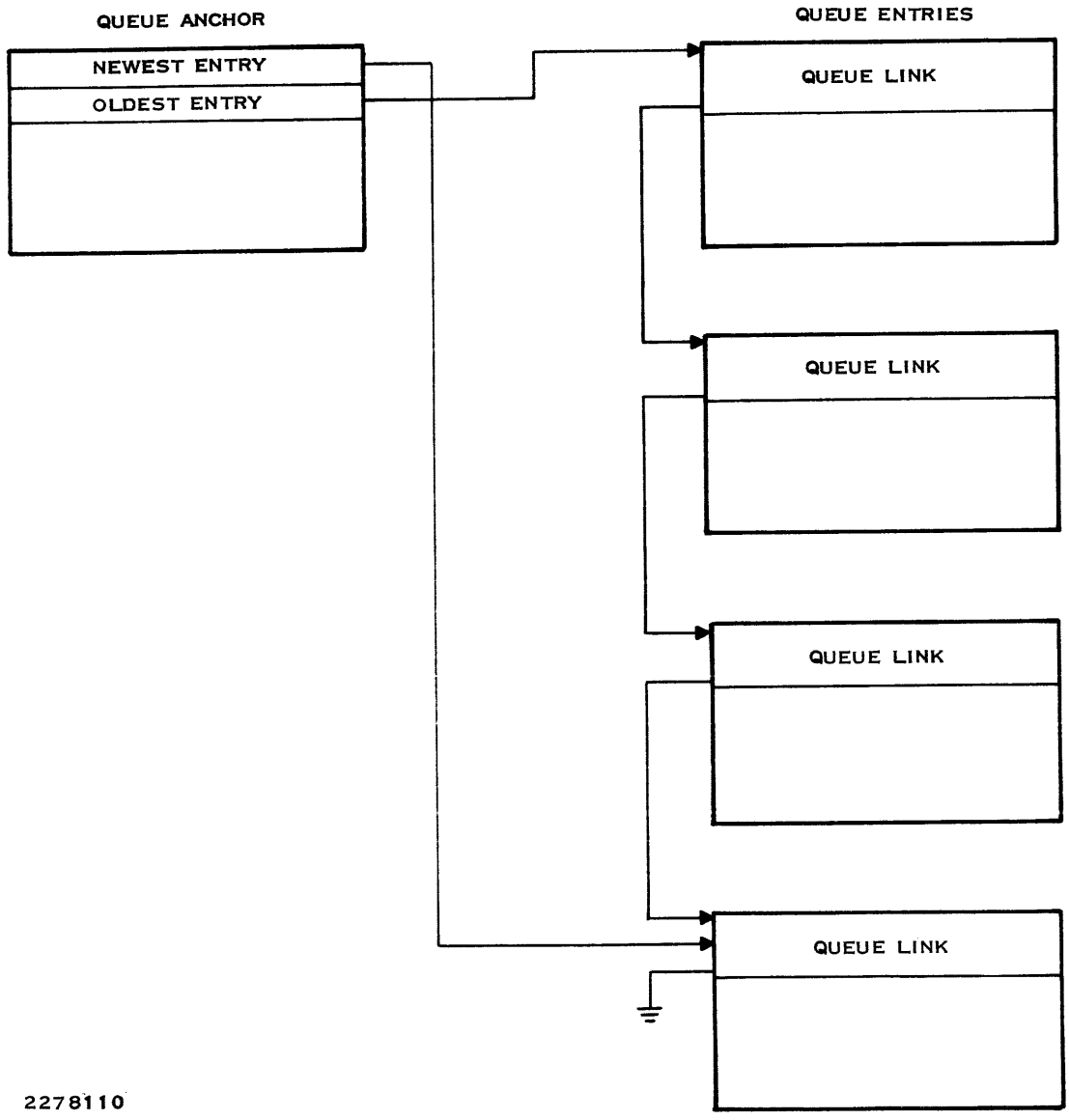


Figure 1-2 DX10 Queue Structure

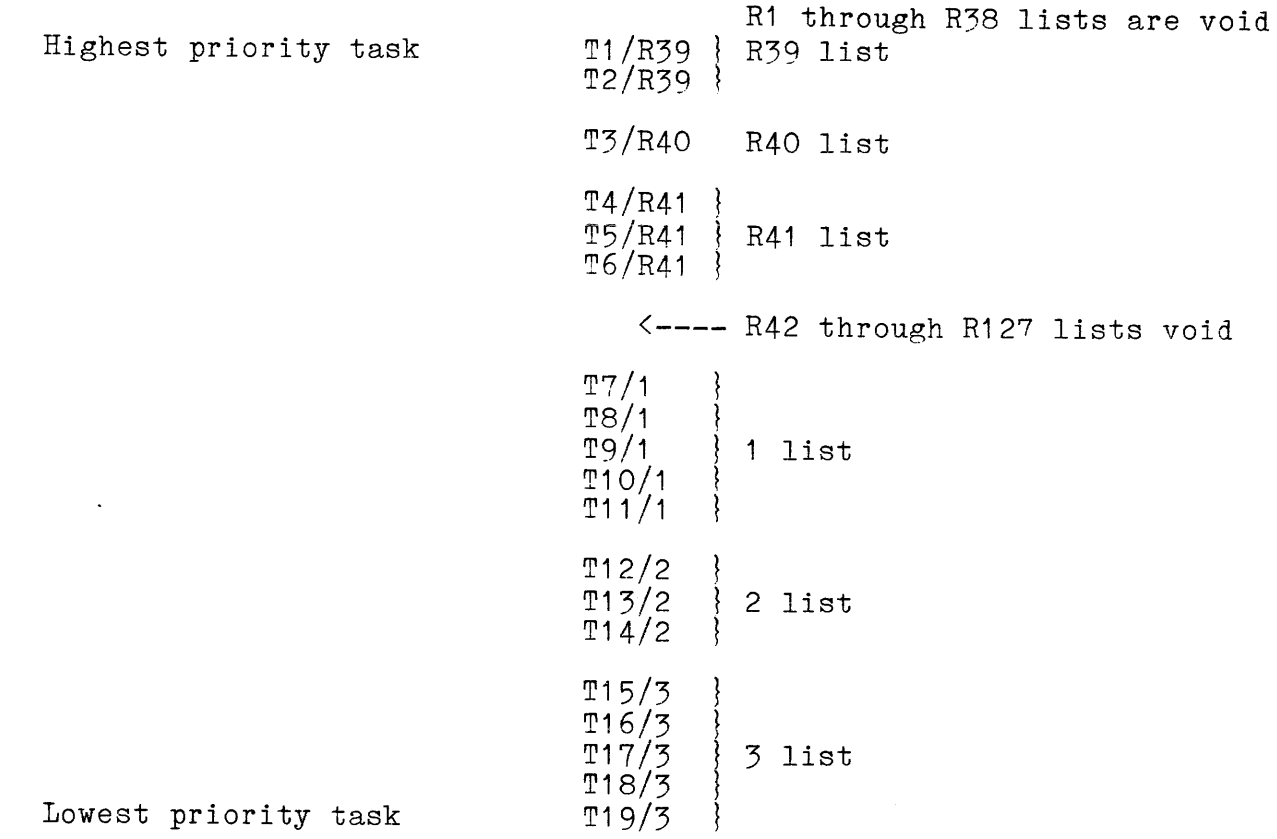


Figure 1-3 Active Task Queue

1.1.2 A 32-BYTE BLOCK OF MEMORY--BEETS.

Under DX10 memory management, a beet is defined to be a 32-byte block of memory. A beet address (boundary) is an absolute address that can be evenly divided by 32. The concept of a beet address is necessary to DX10 memory management in order to fit a 20-bit absolute (unmapped) memory address into a 16-bit word. All memory allocations are integral numbers of beets, and begin on a beet boundary.

1.1.3 CALLING CONVENTIONS.

Within DX10, routines normally call each other using the following sequence:

```
        BL    @SUBR
        DATA  ERROR
NORMAL  EQU   $
```

where SUBR is the entry label of the routine being called, ERROR is an address within the calling routine to which the called subroutine should return in case of an error, and NORMAL (i.e., the next instruction) is the normal (non-error condition) return point.

Since the call is made using a BL instruction, R11 points to the word containing the error return address. The following sequence is generally used by the called subroutine, when returning.

```
        MOV   @ERRCOD,R0      Put any error code in R0.
        JEQ   RTNORM          If no error, do normal return.
ERRET   MOV   *R11,R11        If error, return to address
        RT                    contained in word following
RTNORM  INCT  R11             the call.  Normal return
        RT                    is to the second word
                              after the call.
```

Since the return sequence is often used in DX10, a special routine, POPO, is provided to perform the return (see the section on common routines).

1.1.4 SYSTEM MEMORY MAPPING.

Using the memory mapping option available on the 990/10 computer, the DX10 operating system is divided into several different mapping schemes using map files 0 and 1 (Figure 1-4). All mapping schemes include the DX10 data base, system table area, and common system routines (called the system root) as the first segment (memory mapping allows a program to be divided into three segments which need not be in contiguous memory locations). All schemes which use map file 0 have the I/O common routines (routines commonly used by device service routines) as the second segment. The third segment of all map 0 schemes is one of the following:

...The task scheduler and SVC and XOP code

...A device service routine (DSR).

Figure 1-5 shows how the logical address space of map 0 schemes is arranged.

One of the map file 1 schemes includes not only the system root in the first segment, but also the file management and key indexed file handling code (i.e., file management and key indexed file code are physically located in memory immediately after the system root). This map scheme allows file management to map I/O buffers into its address space using the two remaining map segments.

Other map 1 schemes include either a system task (memory resident or disk resident) or system common as the second segment. The third segment is available for use by the task. Figure 1-6 shows possible arrangements of map file 1 schemes.

The link map of a generated system contains information on exactly which DX10 modules are included in each mapped segment.

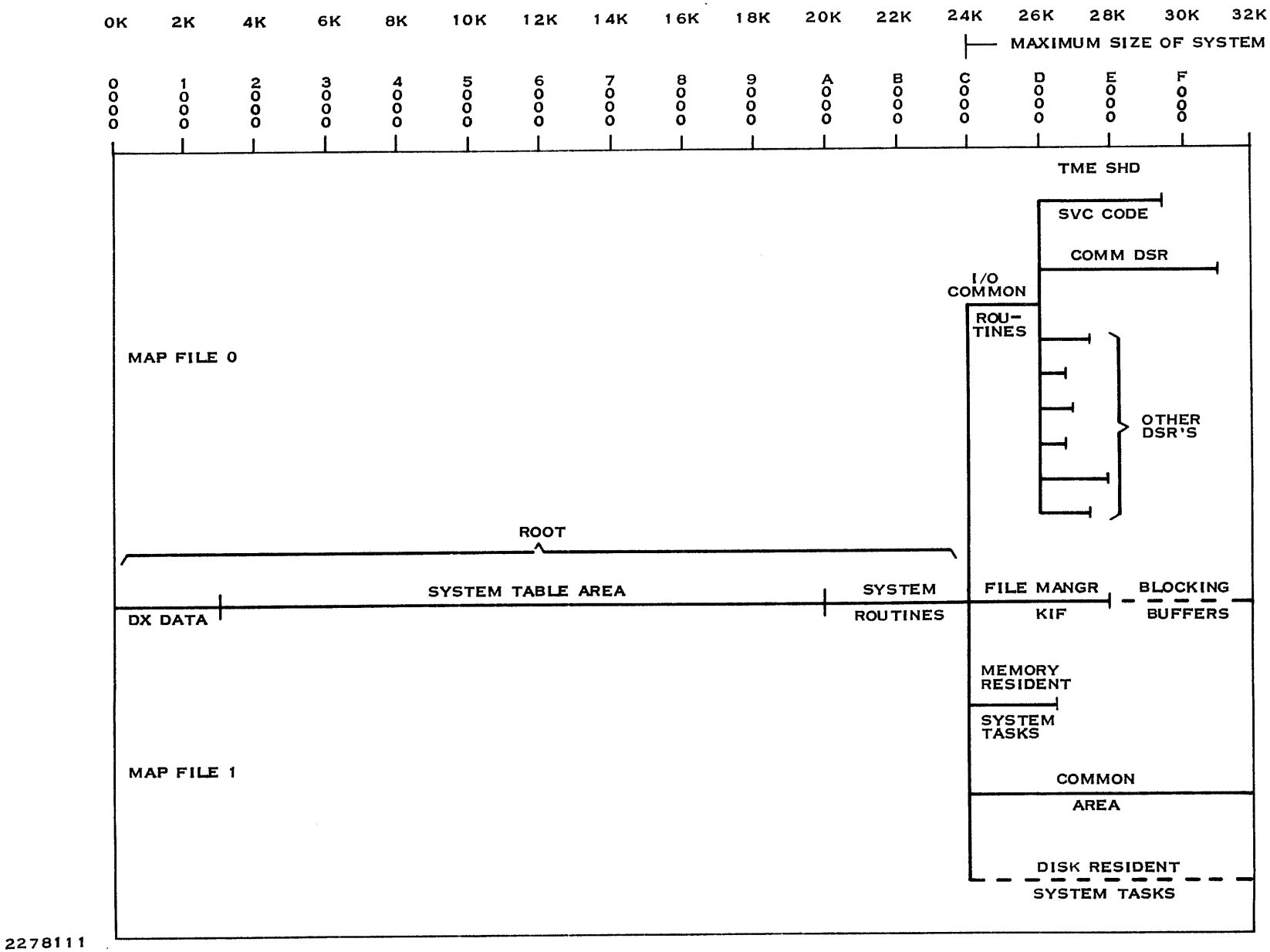


Figure 1-4 System Memory Mapping

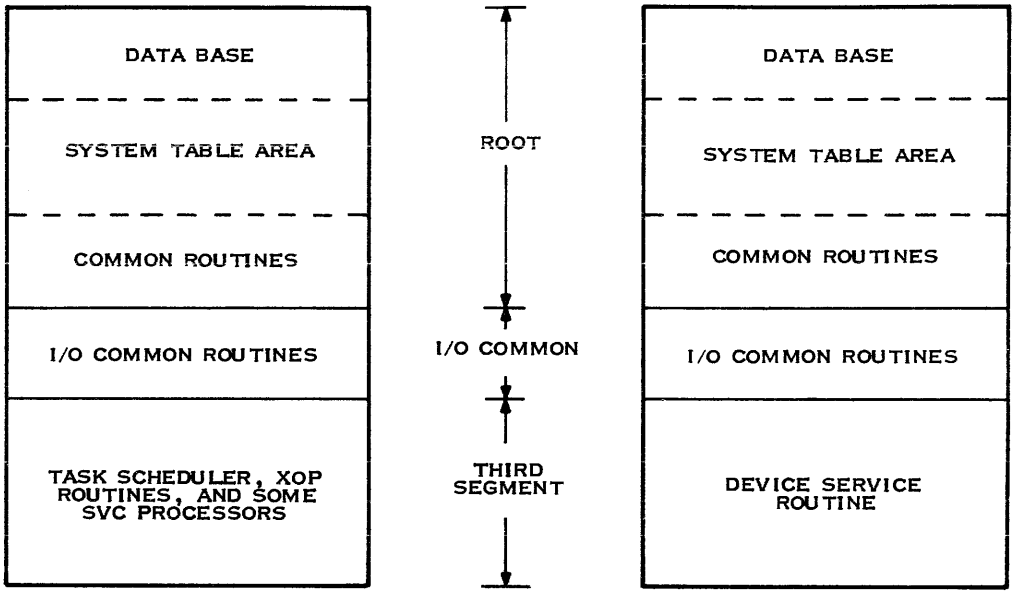


Figure 1-5 System Map File 0 Schemes

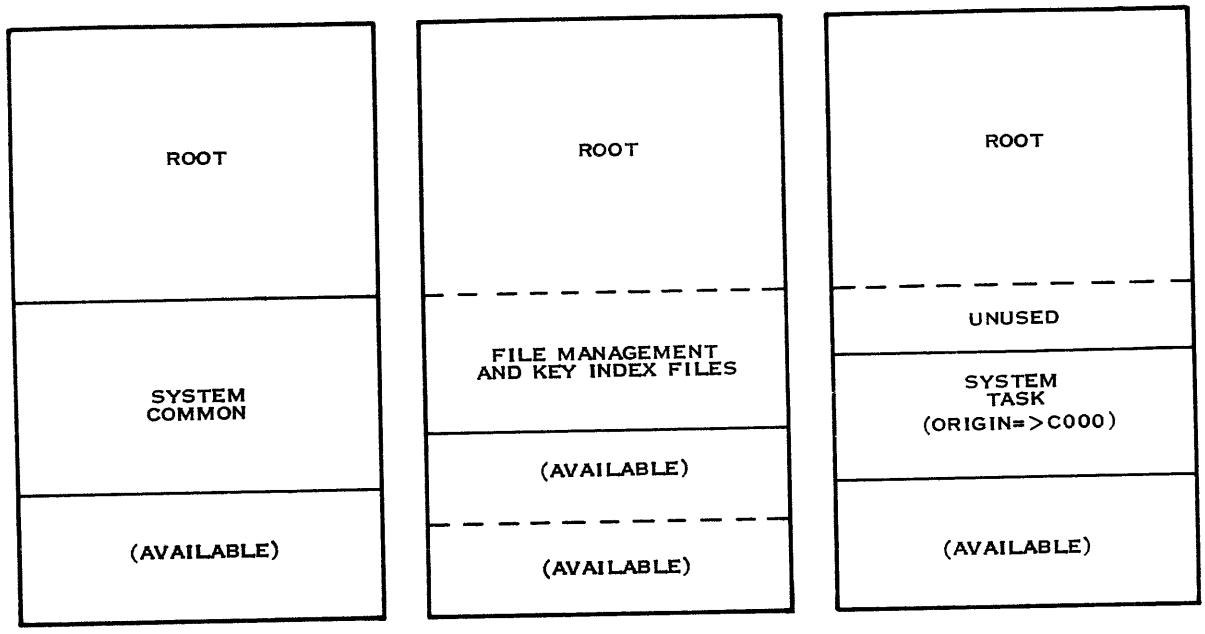


Figure 1-6 System Map File 1 Schemes

1.2 HOW PARTS OF DX10 FIT TOGETHER

The flow of control through DX10 follows many separate paths, depending on the action currently being performed. The remainder of this section traces these control paths separately, following the general order of events caused by the execution of a task. The order begins with SVC processing and the task bid and ends with task termination.

1.2.1 SVC PROCESSING.

Under DX10, supervisor calls are implemented as extended operation (XOP) 15. When an SVC is issued, control is passed via the XOP trap vector table to the SVC decoding routine, SVCINT, which is the XOP processor for XOP 15. This routine determines which SVC is desired by decoding the SVC code in the call block, and relinquishes control to the SVC processor.

If the SVC processor is a queue server, control passes from SVCINT to the SVC buffering routine, SVCBUF. This routine buffers the call block into the system table area and queues the buffered call block on the proper queue, thereby activating the associated queue server task. The task that issued the SVC is suspended.

If the SVC is for I/O (code 00), the SVC must go through yet another stage of decoding by the I/O supervisor, DXIOS. This routine buffers the user I/O data block and supervisor call block into the system table area, and determines whether the call is for file management (disk file I/O), file utility, or device I/O. File Management and File Utility calls are queued for the file management task or file utility task, respectively. Both are queue servers. Device I/O requests are handled by DXIOS and the device service routine directly, if the device is not busy. If the device is busy, the request is queued in the device queue for later processing.

The return of control to the task that issued the SVC is different for queue serving SVC processors and nonqueue serving processors. Non-queue servers simply return control to the calling task via the system return point XOPRT1, allowing the calling task to resume execution. Queue servers do not immediately return control to the calling task, but continue to process queue entries. When an entry has been processed, it is queued for the SVC cleanup routine, SVCCLN. SVCCLN unbuffers each entry from the system table buffer into the calling tasks memory, releases the system table area, and reactivates the task.

Figure 1-7 shows the flow of control through the DX10 modules involved in SVC processing.

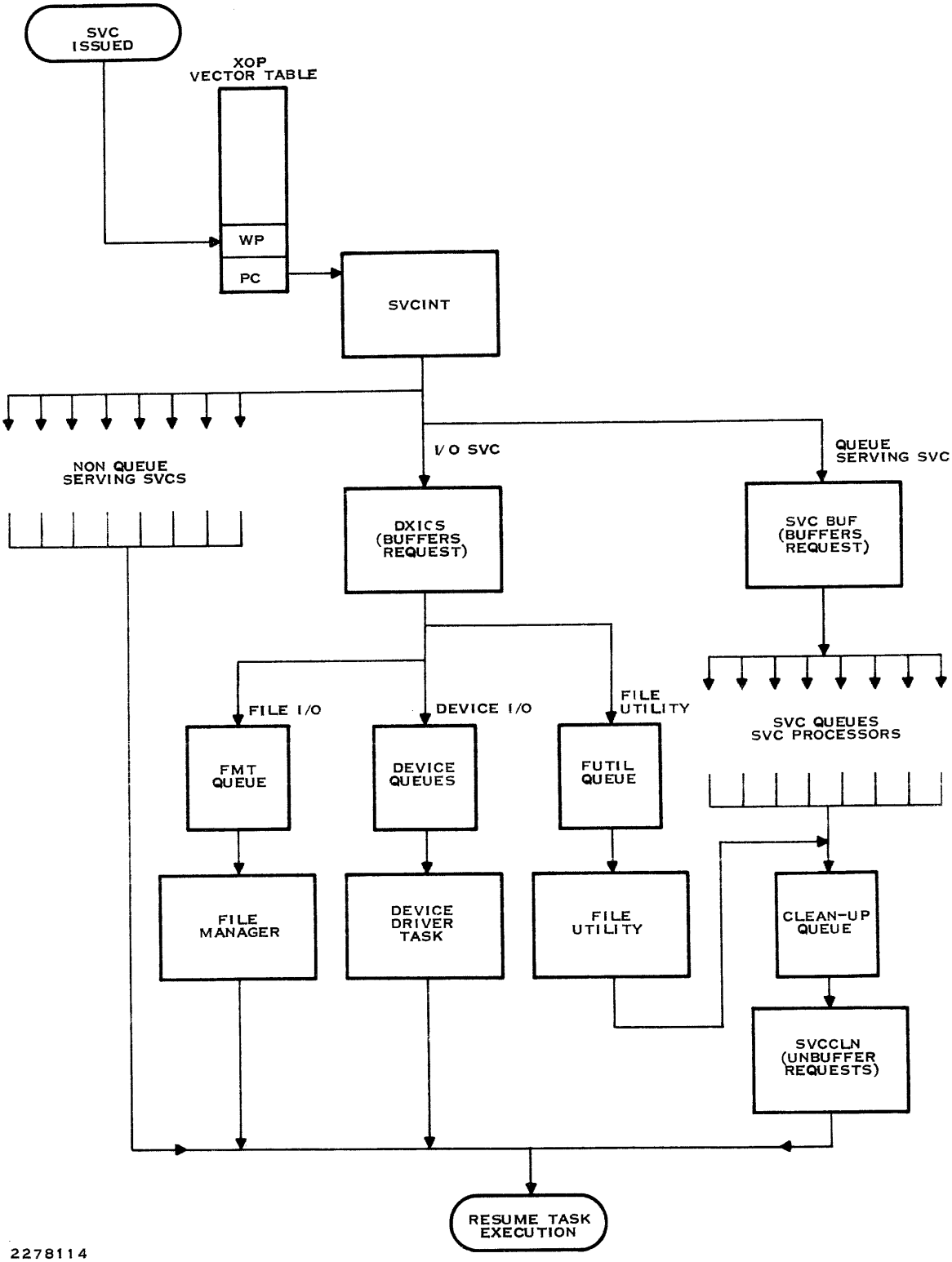


Figure 1-7 SVC Processing Flow of Control

1.2.2 BIDDING A TASK.

The process of bidding or executing a task under DX10 involves building a task status block to describe the task, queueing the TSB for processing by the bidder task, TM$BID, and then queueing it for the task loader, as shown in Figure 1-8.

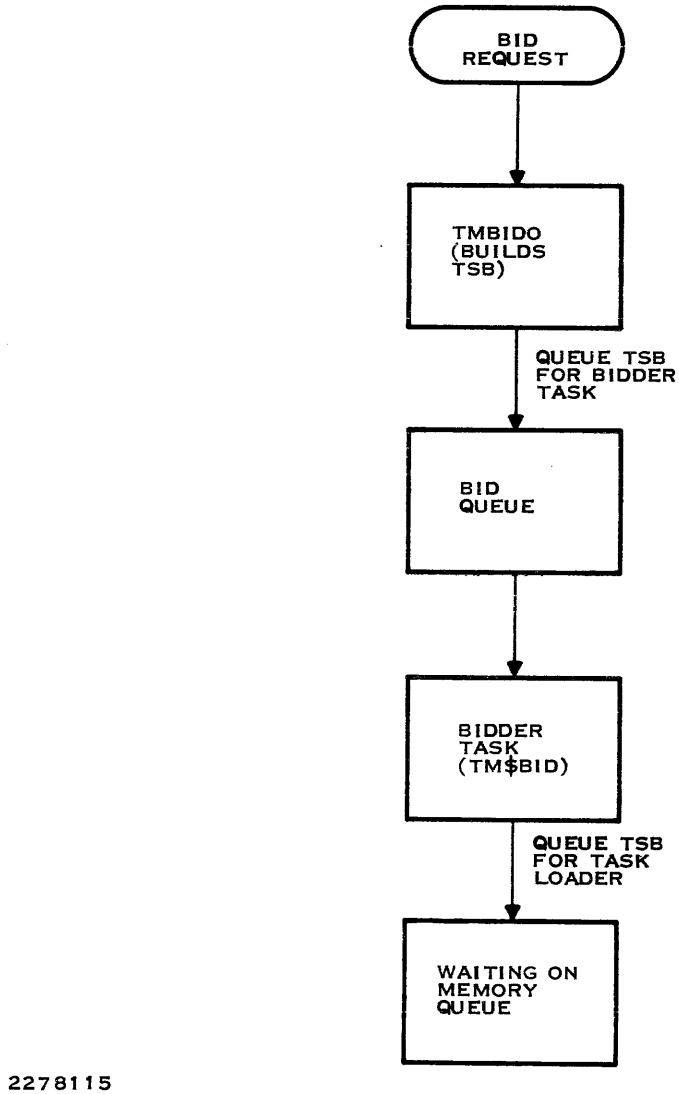


Figure 1-8 Bidding a Task

The first action taken by DX10 when bidding a task is to build a task status block (TSB). A TSB is a block of overhead data maintained by the operating system to describe each task currently running (either executing, waiting for CPU time, or rolled-out). The TSB contains pointers to other system overhead blocks associated with a task (e.g., logical device tables for each LUNO assigned by the task, procedure status blocks for any attached procedures), as well as other information describing the task to the operating system (see the section on data structures for more detail on task status blocks and procedure status blocks). From the time a task is bid until it terminates, the task is represented within DX10 by its TSB; all actions taken by the system in order to execute the task (e.g., roll-in, roll-out, memory allocation) reference the TSB.

To build a TSB, the system routine TMBIDO calls memory management to reserve a block of memory from the system table area. It initializes some of the fields in the TSB, generates a run-time ID for the task, and then queues the TSB for further processing by the bidder task TM$BID.

The bidder task is a memory-resident server. It dequeues a TSB from its queue, and fills in more fields (e.g., the task's priority) using data from the program file on which the task is installed. The bidder task also fills in the "family tree" pointers in the TSB. These four pointers link the newly bid task with other tasks, according to the structure shown in Figure 1-9. The parent task is the task that issued the Execute Task SVC to bid the task. If the task was bid from a terminal through an SCI command, SCI is the parent task. The brother tasks are other tasks that were bid by the same parent task. The oldest son is the first task that is bid by the task. This pointer is given a zero value when the task is first bid.

Other pointers that are initialized by the bidder task are pointers to the procedure status blocks of any procedures associated with the task. A procedure status block serves a similar function for procedures as a task, although the pointers are not as complex (see Section 6, Data Structures, for more detail on PSBs). If a procedure attached to the task being bid is not currently in memory (i.e., does not have a PSB), the bidder task gets a block of system table area and builds the PSB for the procedure; the bidder task accesses the program file for necessary information. Figure 1-10 shows how TSBs and PSBs are linked by pointers under DX10. Note that PSBs may be linked with more than one TSB, since a single procedure may be attached to more than one task.

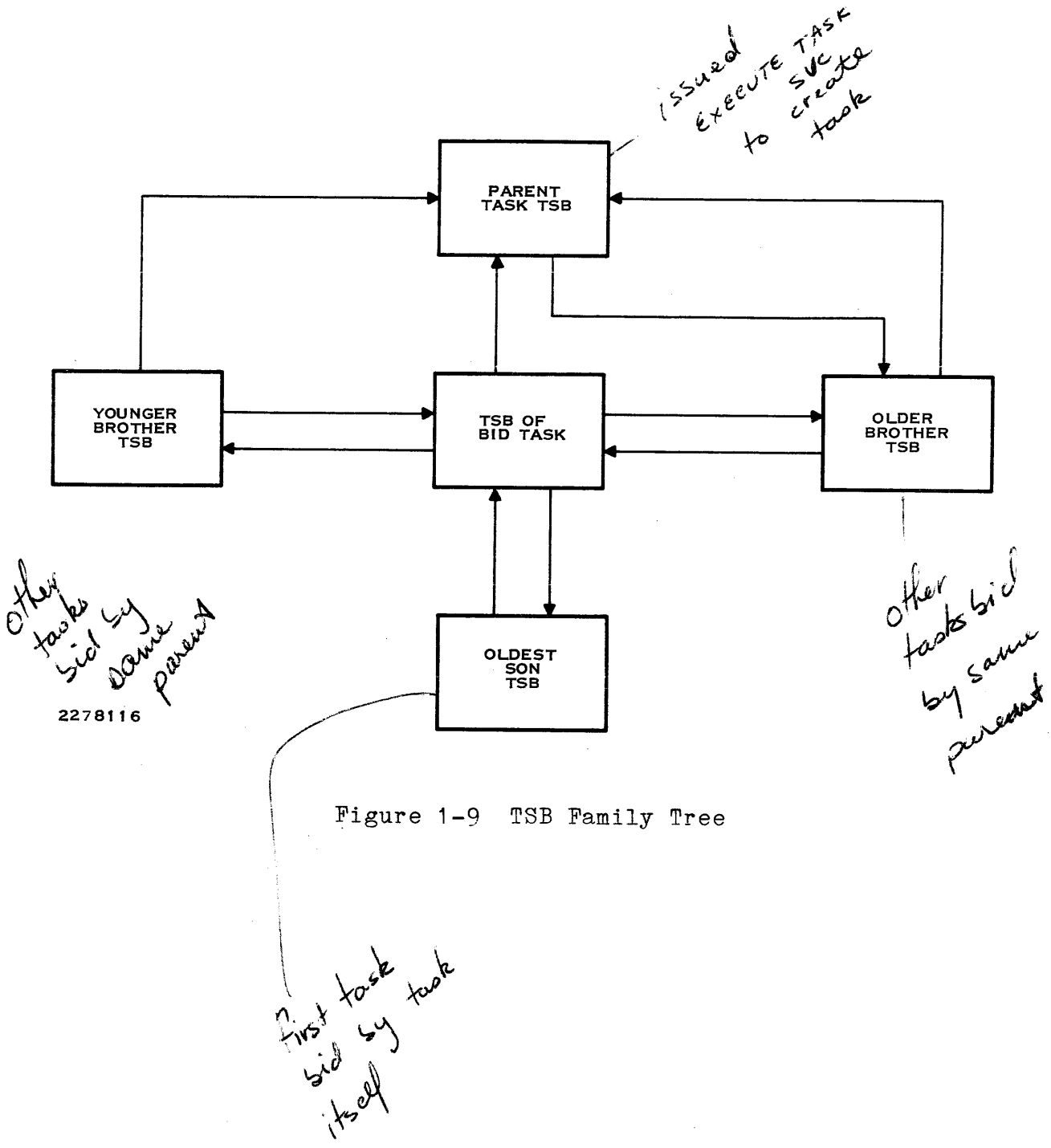


Figure 1-9 TSB Family Tree

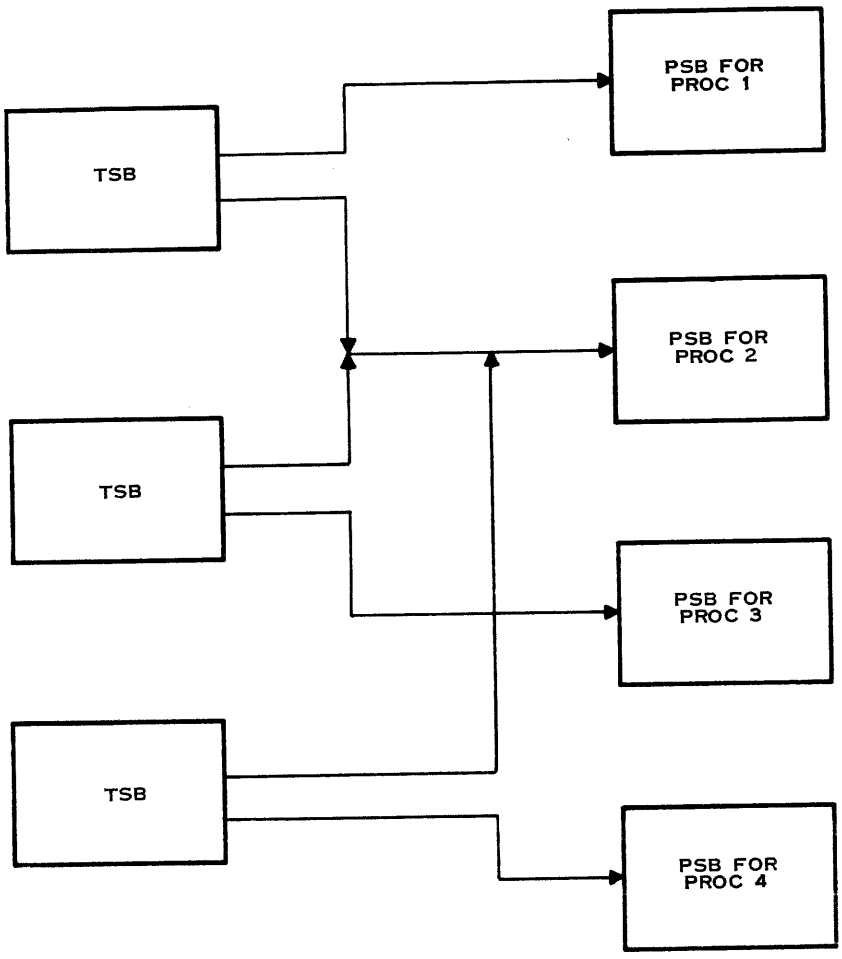


Figure 1-10 TSB/PSB Relationship

Finally, the bidder task queues the TSB on the waiting on memory queue, which is serviced by the task loader. The bidding task (parent task), which had been suspended while TM$BID processed the SVC, is reactivated.

The scheduler and operating system provide a means for tasks to be bid from interrupt processors. The bid-task interrupt processor resets the interrupt, sets the bid-task-in-progress flag of the associated physical device table, saves the ID of the task to be bid (the task to be bid must reside in the system program file), sets up the processor interrupt vector for handling multiple interrupts before the first bid is complete, sets the reenter me flag, and returns through the trap return module. The scheduler scans the PDT list for reenter me flags every time the scheduler runs (every 50 milliseconds), so that when the scheduler executes, the interrupt processor is reentered and the scheduler inhibit flag (TM$EXT) is cleared.

When the processor is reentered, all lower interrupts are disabled. The interrupt processor resets the reenter me flag, initializes the registers required, and calls TMBID0 to bid the required task. Upon return to the interrupt processor from TMBID0, the returned error code is checked. Appropriate action is taken when the error code is nonzero; when the error code is zero, the processor is exited.

To exit the processor because of an error or bid complete, the bid-task-in-progress flag is reset and the interrupt entry vector is set up for processing subsequent interrupts. Exit is made by an RTWP instruction which returns to the scheduler for any necessary scheduling.

The following list contains register definitions for a call to TMBID0:

```
R0       = Error Return
R1       = Installed ID/SVC flag
           SVC flag:  0 = SVC call
                      NOT 0 = Not an SVC call
R2       = Bid Parameter #1
R3       = Bid Parameter #2
R4       = Station ID/Program File LUNO
R5-R9    = Not used
R10      = Stack Pointer (The Scheduler Stack is used.)
R11      = Branch and Link Return Address
R12-R15  = Not used
```

The following list contains register definitions upon return from TMBID0:

```
R0      = Error Return
R1      = Run ID/QUE - NO QUE flag (Bit 15)
          Bit 15:  0 = Request not queued,
                   1 = Request queued
R2      = TSB address of bid task
R3-R15  = Not used
```

The following error codes are returned from TMBIDO:

0 = No error
1 = Illegal station number on bid task
2 = No runtime task IDs available
3 = No system table area available
4 = Illegal program file LUNO

1.2.3 SCHEDULING, LOADING, AND ROLLING A TASK.

Once a task has been bid and a TSB has been built for it, that task must be loaded into memory before it can be scheduled for execution. The business of loading and executing multiple tasks is managed by the task scheduler in conjunction with the task loader and memory management.

1.2.3.1 Scheduling.

Tasks running under DX10 may be found in a variety of states, including: active, suspended, queued for a SVC processor, and waiting for various types of I/O. TSBs of tasks that are waiting for CPU time (i.e., active tasks), and are in memory, are queued on the priority-ordered active queue. The task scheduler always picks the highest priority task off the active queue. Before actually giving the CPU to this task, the scheduler first checks the queue of tasks waiting on memory.

This queue is a priority-ordered queue (highest priority first) of TSBs of tasks which are either rolled-out or have just been bid, and are waiting to get memory in order to execute. The scheduler determines if any task waiting on memory is of higher or equal priority to the task it has chosen from the active queue by searching the TSBs in the waiting on memory queue. If a higher priority task is waiting, and the task loader is not busy, the scheduler gives the next time slice to the loader rather than the chosen task. If an equal priority task is waiting for memory, and the chosen task has already had a minimum number of time slices, the loader is likewise awarded the time slice. Otherwise, the chosen task gets the time slice.

Figure 1-11 shows a simplified version of how the scheduler chooses a task to execute. The scheduler is described in more detail in Appendix E.

1.2.3.2 Loading And Rolling.

The task loader is responsible for loading tasks into memory and rolling them out. Tasks that are to be loaded may be either rolled-out tasks or tasks that have just been bid. The task status blocks of all tasks to be loaded are on the waiting on memory queue. The queue is sorted so that higher priority tasks are processed first. The queue is first-in, first-out for tasks of the same priority.

Tasks that are to be rolled-out by the task loader are tasks that have either been selected by memory management (see paragraph 1.2.3.3) to be rolled-out, but had TILINE I/O in progress, or issued a Get Memory SVC. Since TILINE I/O accesses the task's memory directly, the task could not be immediately rolled-out; therefore, memory management flags the TSB to indicate that it is being "quieted" (waiting for the I/O to complete). The TSB remains on the active queue until the I/O is complete, at which time the scheduler queues the TSB onto the quieted queue. TSBs of tasks which issue Get Memory SVCs are immediately queued on the quieted queue. All tasks that are on the quieted queue are rolled-out by the task loader.

The task loader is a dedicated server of the quieted queue; that is, it is automatically bid when an entry is made on the queue. The loader may also be scheduled to execute when the task scheduler decides to try to roll a task into memory, although the loader does not "serve" the waiting on memory queue. Figure 1-12 shows how the loader processes the two queues.

When the loader is executed, it first processes all of the entries on the quieted queue, rolling them out of memory, requeueing the TSBs onto the waiting on memory queue, and releasing the now vacant memory used by the tasks. When the quieted queue is empty, the loader checks to see if there is a task waiting on memory (i.e., a TSB on the waiting on memory queue). If not, the loader issues a Suspend Awaiting Queue Input SVC, returning control to the scheduler.

If a task is waiting on memory, the loader checks for attached procedures, and tries to allocate memory for them (the allocation logic first checks to see if the procedure is already in memory). If memory is found, the loader calls memory management to allocate memory for the task segment. If no memory is available, the load is left pending, and the loader checks to see if any more entries have been made on the quieted queue. At the end of the allocation phase, the loader checks to see if all of the memory required is still allocated. This might not be true if memory management has rolled out procedure 1 when allocating memory for procedure 2.

If all of the memory is safely allocated, the loader loads the task and procedures (unless they were already in memory) from either the roll file (for roll-ins) or a program file (for initial bids). If the loaded task is flagged as memory resident, the loader assumes that it was part of the initial program load (system boot) and puts the task in a terminated state. Otherwise, the loader queues the task on active queue priority 0, slot 2 (head of the queue). This is done, regardless of the task's assigned priority, in order to insure that the task will get a time slice before it could be rolled-out by a higher priority task.

After the task has been loaded, or some error has interrupted the load, the loader once again checks the quieted queue for entries. If there are any, the loader starts all over at the top of the cycle; otherwise, the loader suspends itself, returning control to the scheduler.

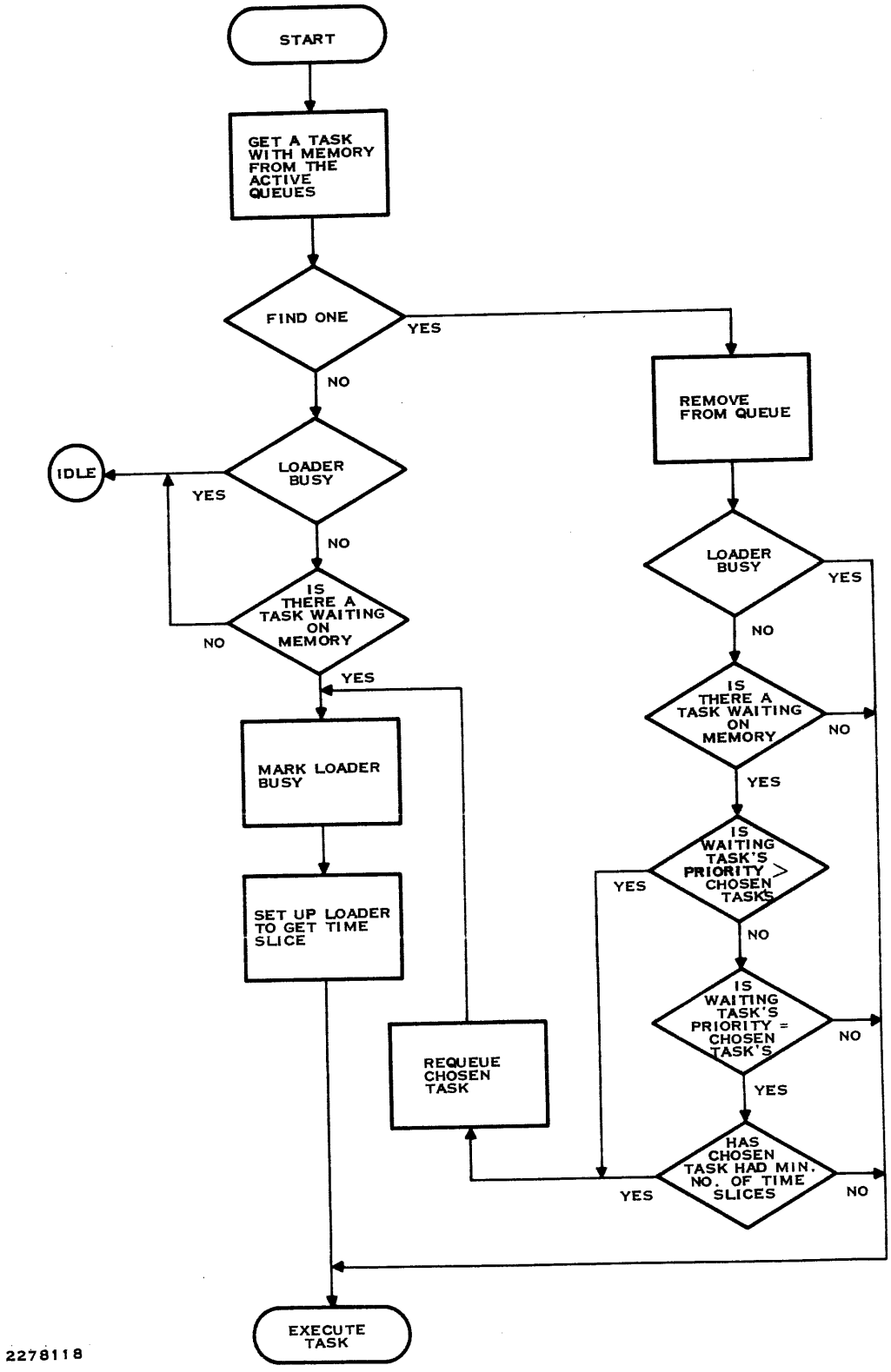


Figure 1-11 Simplified Flow of Scheduler

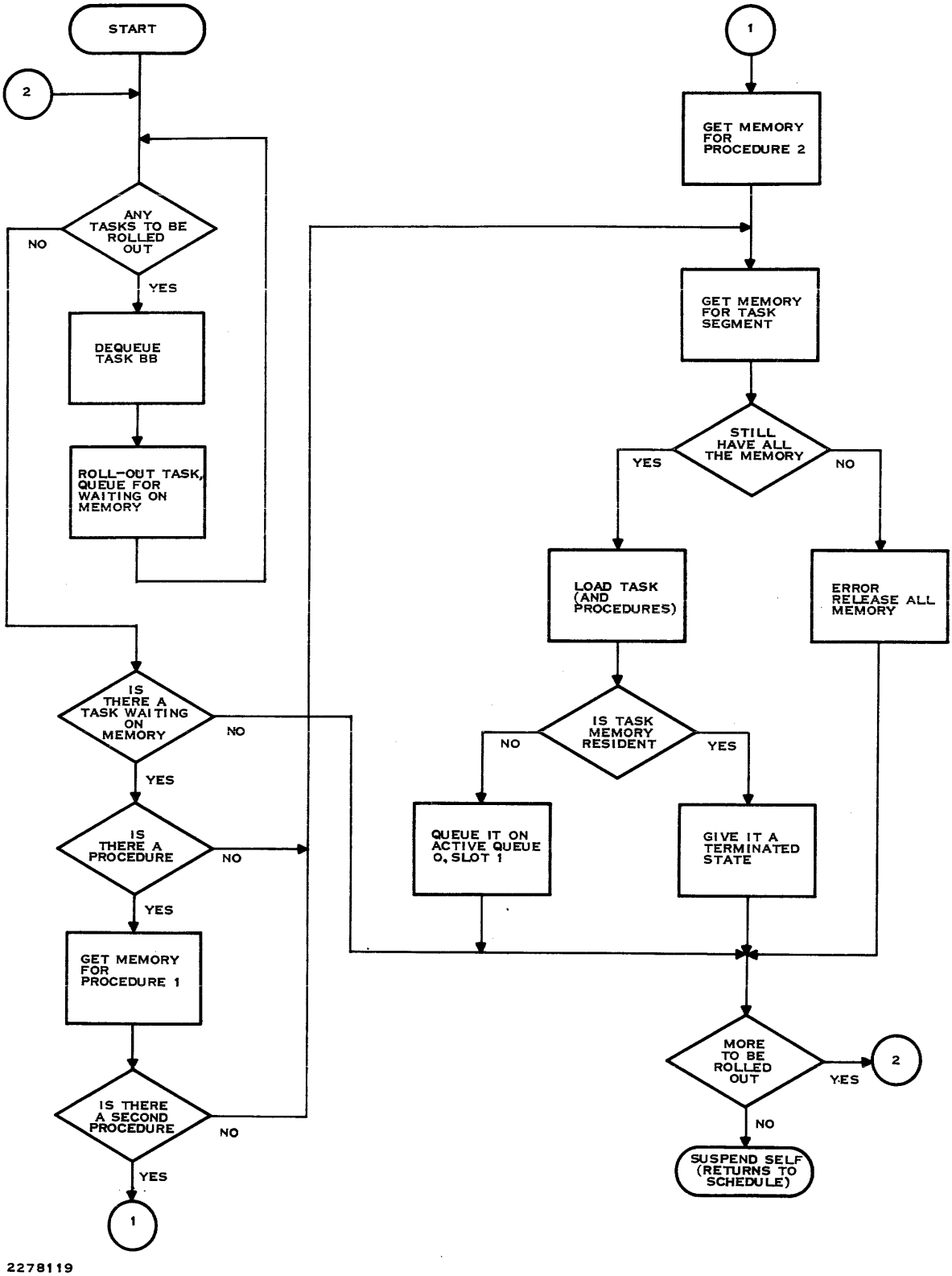


Figure 1-12 Simplified Flow of Loader

1.2.3.3 Memory Management.

Available memory under DX10 is divided into system table area and user area (see Figure 1-1). All of this memory is dynamically allocated and deallocatd by a collection of nucleus routines called memory management. Memory is organized in four separate groups:

...Free user memory

...Allocated user memory

...Free system table area

...Allocated system table area

The allocated blocks of system table area are the various system data structures, such as task status blocks, and are not all linked on a single list. Free blocks of system table area are all linked on a single list, headed by SAHEAD, an anchor contained in the DX10 data base module (described in Section 7).

Free blocks of user memory are linked on a single list headed by UAHEAD, another anchor in the data base module. Allocated blocks of memory, which may contain tasks, procedures, or file blocking buffers, are linked together on a time-ordered list. The time-ordered list (TOL) is maintained such that, whenever a block is accessed (i.e., the task executes, or the blocking buffer is read or written to), it is removed from its current position on the TOL and relinked at the head of the list when it is no longer being used. Thus, the list is ordered, the first blocks on the list being the most recently used and the last blocks being the least recently used. Figure 1-13 shows how the time-ordered list is structured.

Requests for system table area are serviced immediately by the table area allocating routine, MM$GSA. Since nothing in the system table area can be rolled (all system tables and device buffers are essential), the request is filled from available area only, and must immediately fail or succeed depending on how much table area is free. The allocation routine does a first-fit, linear search of the free table area list, starting at the list header. If the search is unsuccessful, the routine scans the TSB list, checking for disk resident queue servers which are suspended awaiting queue input. If such a task is found, its memory is released, its TSB is deallocated, and the search for table area is restarted.

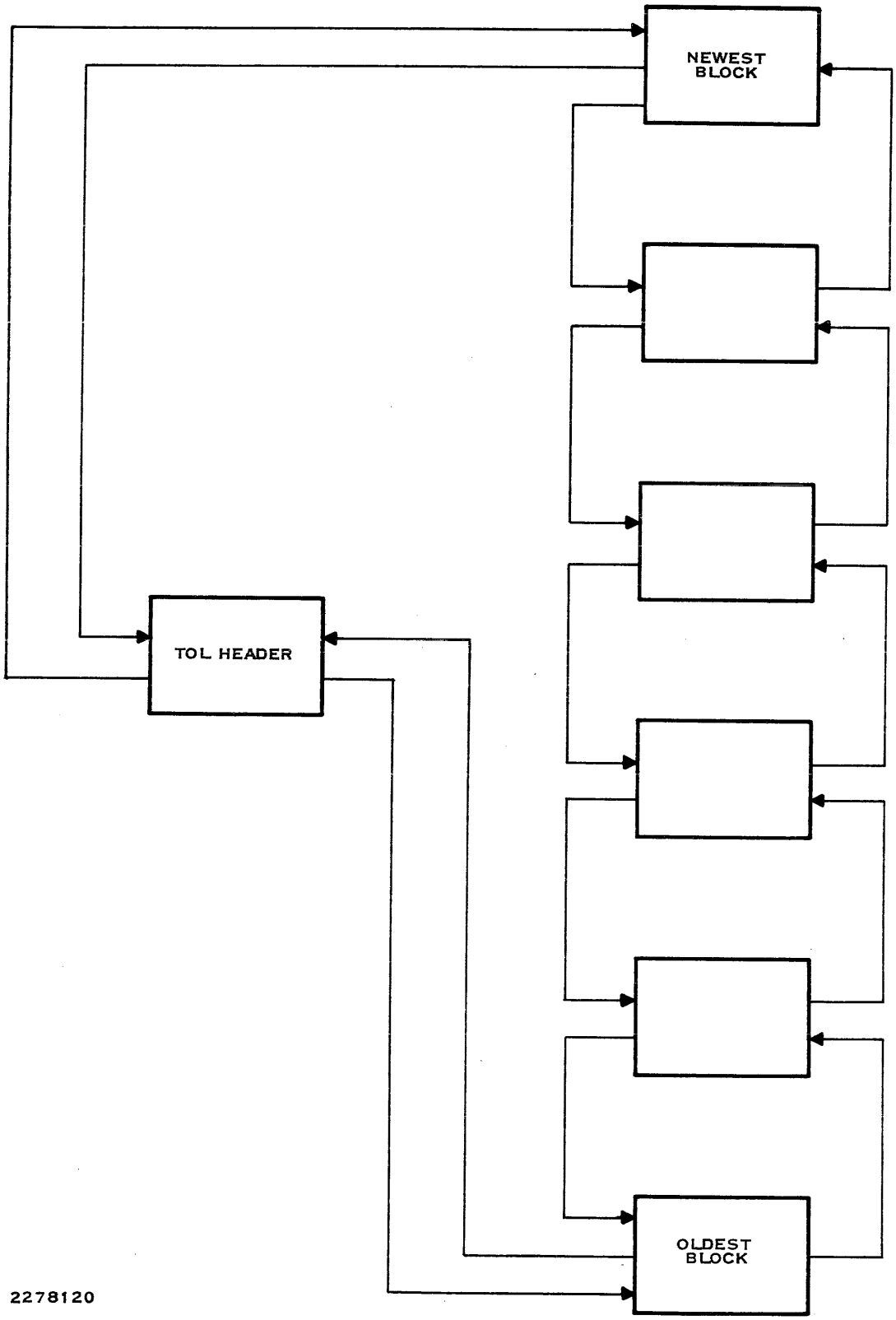


Figure 1-13 Time-Ordered List

Requests for user memory (e.g., for tasks and procedures) are not always serviced immediately. The find memory routine, MM$FND, only processes one request at a time; further requests are queued until they can be serviced. The find memory routine first searches the available memory list, using a first-fit search. If a free block of adequate size is found, the starting address of the block is returned to the requester. If no free block is available, the time-ordered list is scanned (by MM$SCN) for a rollable block of memory.

The time ordered list (TOL) scan is from oldest (least recently used) block to newest (most recently used). Blocks are chosen for roll-out according to the following rules:

1. Any buffer except the memory resident buffer; if the requester is buffer management, then the memory resident buffer may also be chosen.
2. Tasks with lower priority.
3. Higher priority tasks that have been suspended longer than a time threshold.
4. Equal priority tasks that have received a minimum number of time slices since being loaded into memory.
5. Any procedure with no currently attached tasks in memory.
6. Tasks that have TILINE I/O in progress may be flagged for quieting, and subsequent roll-out, if the memory requester is not buffer management.
7. Queue servers that are suspended for queue input.

Tasks that may not be rolled-out are:

1. Tasks that have an alternate task (see Section 6, Data Structures for a description of alternate tasks).
2. Tasks that are queued for the system overlay loader.
3. File utility task (FUTIL) when it is accessing a directory.

If the TOL scan is successful, the block chosen is rolled-out (if it is a task or procedure), written to its file (if it is a blocking buffer and has been modified), or simply released (if it is a queue server's memory or an unmodified buffer). The rolled block of memory is released from the TOL and put back on the free memory list. As blocks are added to the free memory list, they are merged with any immediately preceding or following blocks to reduce fragmentation. After a successful scan of the TOL, the find memory routine again searches the free block list. If a large enough block is still not available, the TOL is scanned again for another rollable block. Figure 1-14 shows how user memory is found.

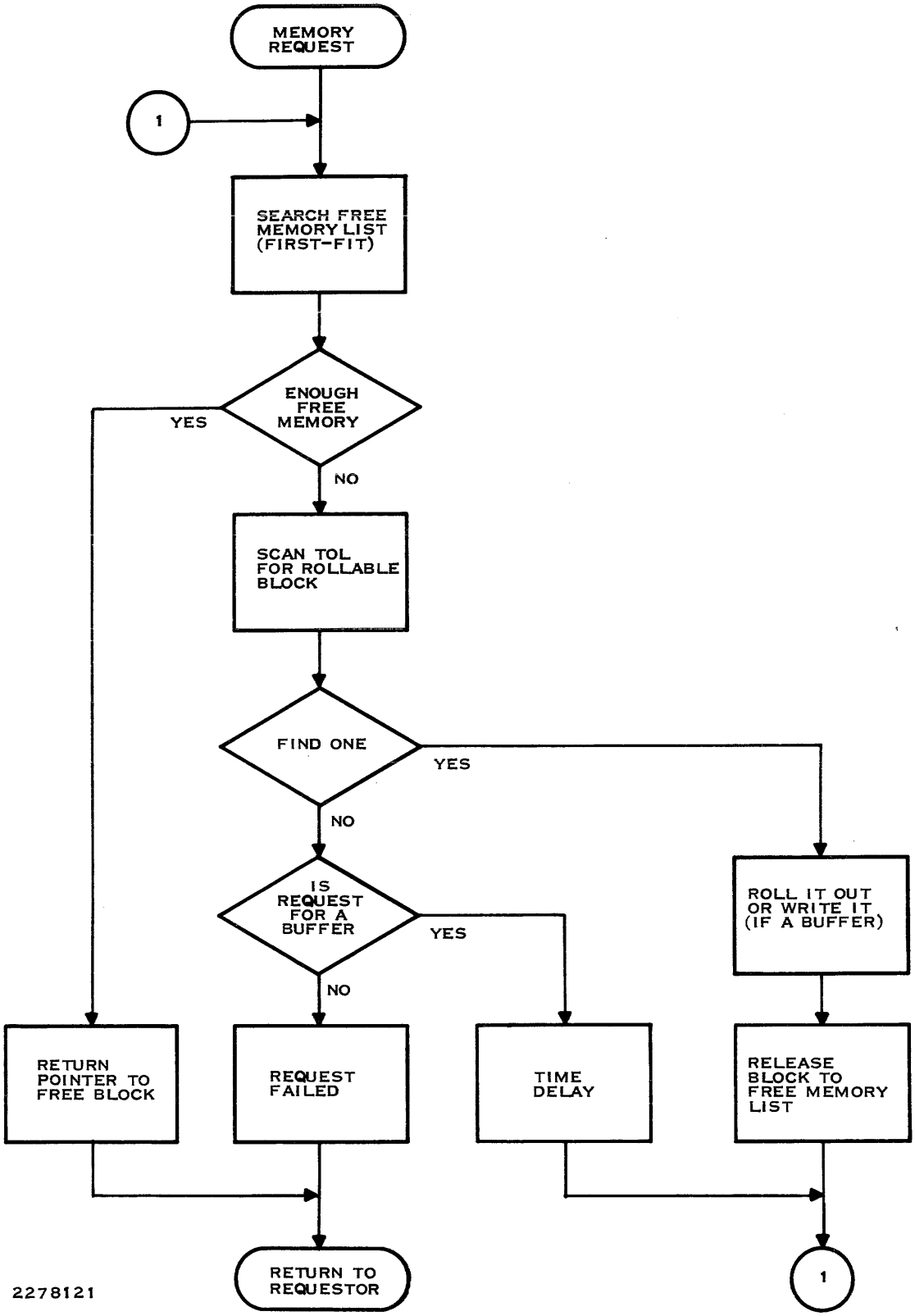


Figure 1-14 Find Memory Flow

1.2.4 DEVICE I/O FLOW.

When an executing task performs I/O, it must be done via an I/O SVC, and directed to a Logical Unit Number (LUNO). The I/O SVC is decoded by the SVC decoder, as described in paragraph 1.2.1, and control passes to the I/O supervisor, DXIOS. The I/O supervisor determines from the I/O opcode that the call is not for file utility. It then searches the logical device table tree for the logical device table (LDT) corresponding to the LUNO to which the I/O is being directed.

A logical device table (LDT) is a system data structure that is created for each LUNO assigned. The LDT describes the logical unit, whether it be assigned to a file or a device (see Section 6, Data Structures, for a detailed description of an LDT). All LDTs in the system are linked in a hierarchy according to the type of LUNO which they describe (Figure 1-15). LDTs for task local LUNOs are linked on a list of LDTs anchored in the task status block for the task. This list is in turn linked to the terminal local LDTs for the terminal with which the task is associated (if any), which are in turn linked to a single list of LDTs for global LUNOs.

When the I/O supervisor searches for the LDT of a LUNO, it searches for the LDT starting at the anchor in the calling task's TSB. The list is searched linearly through the task local LDTs, then the terminal local LDTs (if any), and finally the global LDTs, until an LDT for the desired LUNO is found. Note that this causes task local LUNOs to mask global or terminal-local LUNOs.

When the I/O supervisor finds the LDT, it determines whether the I/O is to a device or file. Device I/O call blocks are buffered into the system table area. If there is a data buffer (e.g., for read or write operations) and the I/O requested is non-TILINE I/O, it is also buffered in the system table area. At this point, the I/O supervisor checks the physical device table (described in Section 6) to see if the device is busy. If not, control is immediately passed to the device service routine (DSR) for the device. Otherwise, the buffered call block is placed on the device queue. If the I/O call is Initiate I/O and the calling task has not exceeded a certain threshold number of Initiate I/Os, control returns immediately to the calling task. If the call is not Initiate I/O, the task is suspended until completion of the I/O.

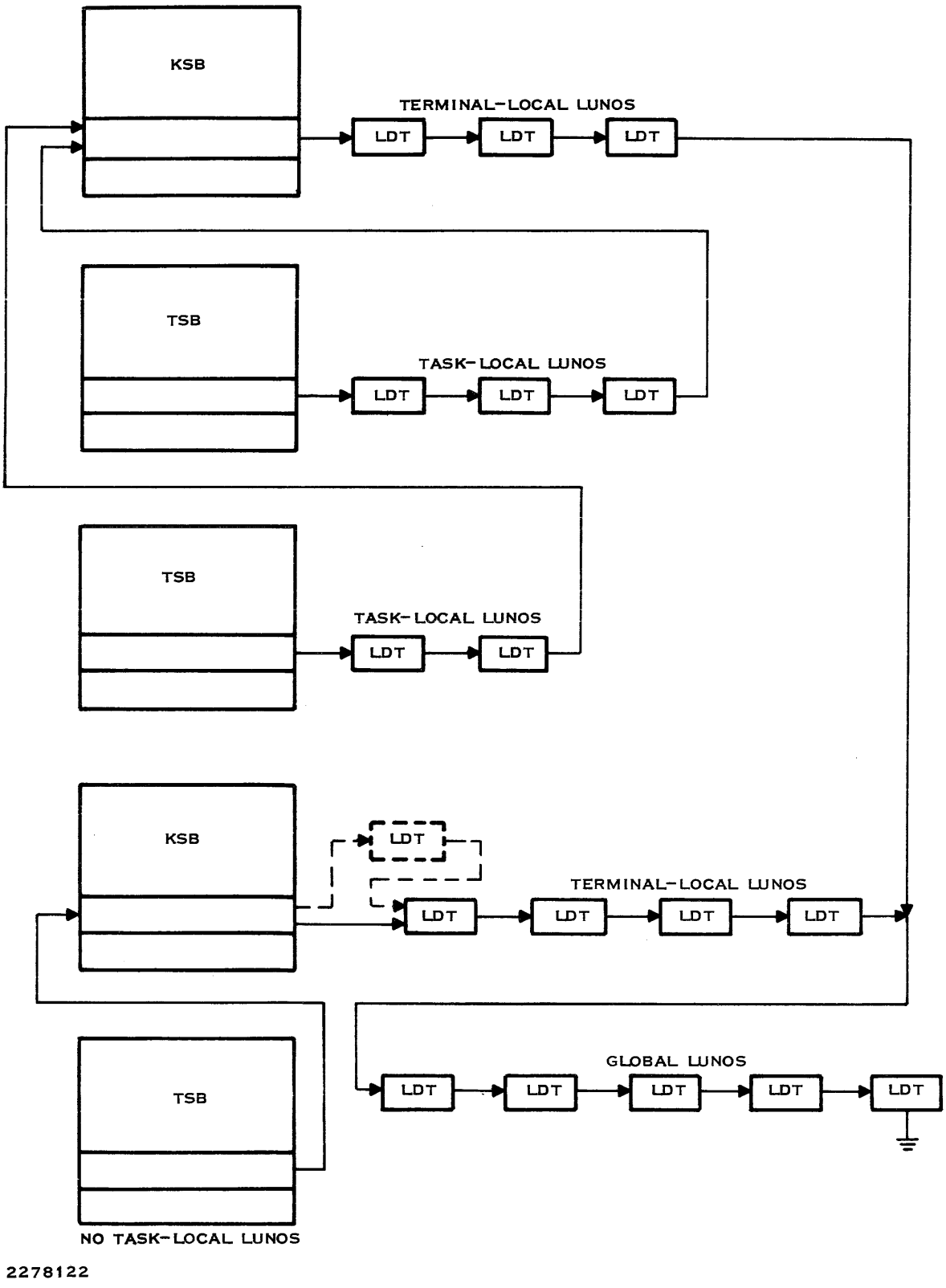


Figure 1-15 Logical Device Table Hierarchy

When processing an I/O request, the I/O supervisor checks the calling task to determine whether it has dynamic priority or not. If it does, the priority is updated according to the type of I/O being done.

When the DSR finishes the I/O operation, it calls an end-of-record (EOR) routine which increments a counter in the calling tasks TSB and places the TSB on the active queue. When the scheduler allots a time slice to the task, it checks the EOR count in the TSB. If the count is non-zero, the device drive task (DDT) is allowed to execute in the task's place.

When DDT executes, it scans the list of physical device tables (PDTs) for a device that needs end-of-record processing. When such a device is found, DDT processes the end-of-record; it unbuffers the call block and data block (if any) into the calling task's memory (causing the task to be rolled-in first, if necessary), and activates the task. After processing an EOR for a device, DDT checks the device queue for queued I/O requests. If a request exists, DDT transfers control to the device service routine initial entry point. When the DSR finishes, it returns to DDT. When the DSR returns, or if the device queue is empty, DDT proceeds to the next PDT on the list.

When DDT has checked all the PDTs, it suspends itself via a Suspend Awaiting Queue Input SVC, returning control to the scheduler. Figure 1-16 shows a simplified flow of the DX10 logic involved in processing a device I/O SVC.

1.2.5 FILE UTILITY FLOW.

The initial processing of a file utility SVC (code 0, opcodes >90 through >9C) is similar to that for device I/O. The SVC is decoded by SVCINT, and control passes to the I/O supervisor. The I/O supervisor determines from the "ninety" opcode that the call is for file utility. The I/O supervisor preprocesses the file utility call by checking for illegal opcodes, buffering the I/O call block, and queueing the buffered call block for the file utility task. The calling task is suspended with a state of "waiting on task driven SVC processor" (state >14). The file utility task, FUTIL, is a queue server and is automatically bid when an entry is placed on its input queue.

When the file utility task gets CPU time, it dequeues an entry from its queue and processes that entry. File utility calls may be made in the old librarian or DX10 2.2 "FUR" formats. The file utility task converts such calls into DX10 3.0 file utility call format and then processes them normally. Normal processing (performed in module UC$) involves a table look-up of the correct processor for the specified file utility opcode (range >90 through >9C), and the transfer of control to that processor. When the processor finishes, it returns control to the file utility task. FU$ then queues the buffered call block for the SVC cleanup task, SVCCLN, which unbuffers the call block into the calling task's memory, and reactivates the task. Figure 1-17 shows the flow of control for file utility SVCs.

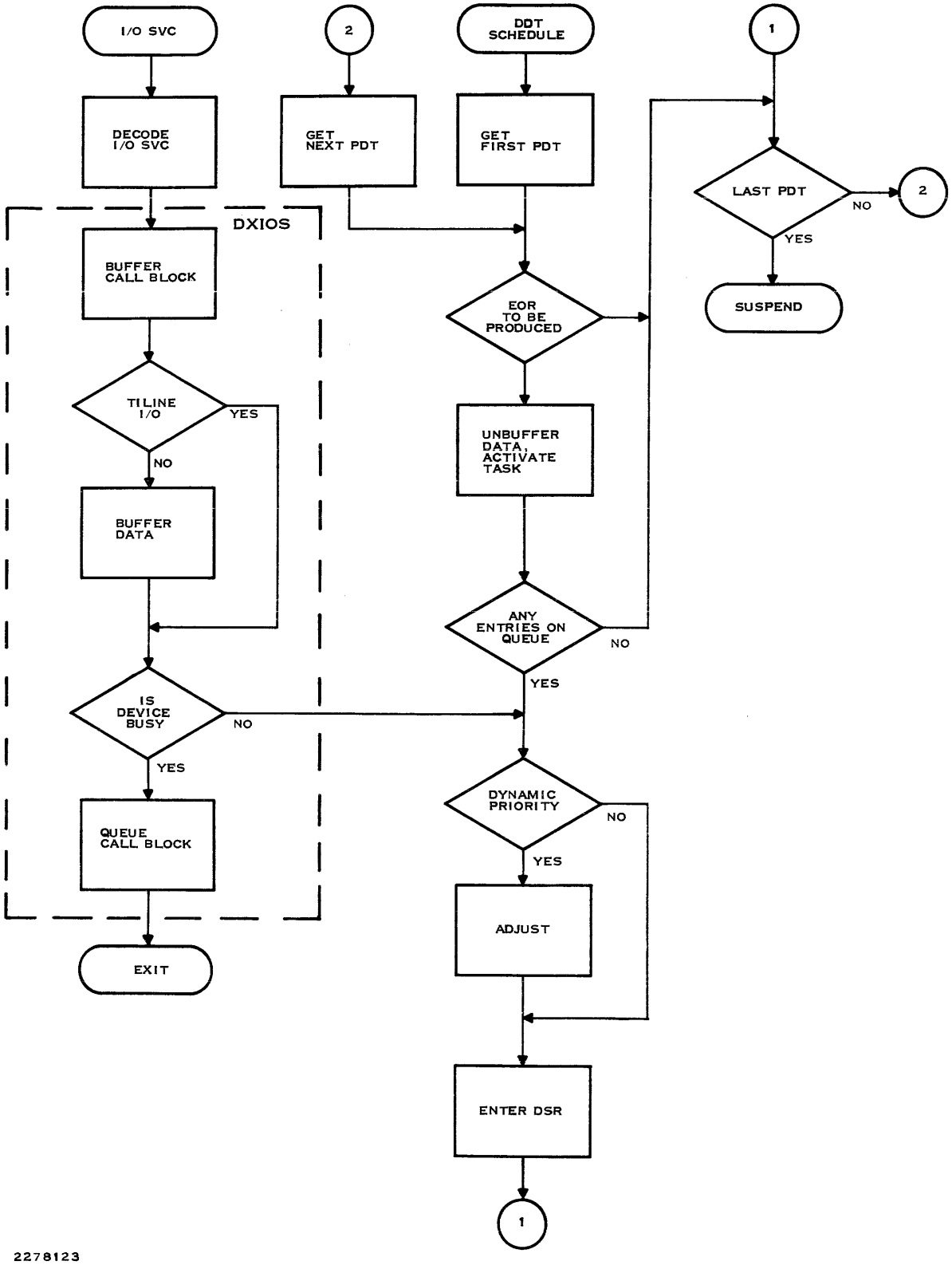


Figure 1-16 Device I/O Processing Flow

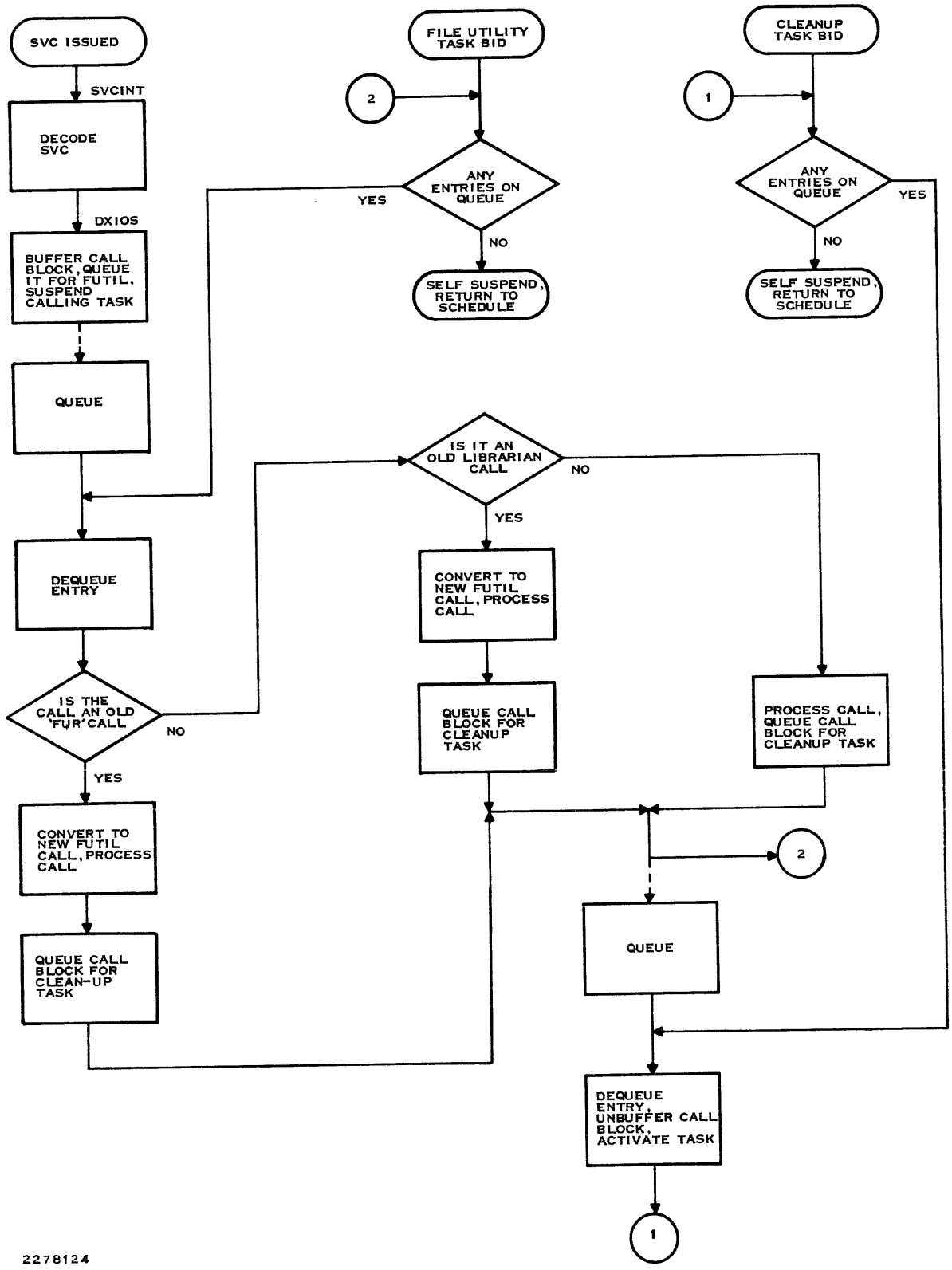


Figure 1-17 File Utility Calling Processing

1.2.5.1 Assigning and Releasing LUNOs.

Within DX10, a LUNO is represented by a data structure called a logical device table. Since the file utility opcodes include Assign LUNO and Release LUNO, file utility is responsible for building logical device tables and maintaining the LDT links. Whenever a LUNO is assigned, an LDT must be built in the system table area, and its pointers defined. Figure 1-18 shows the different pointers which may be set in an LDT.

In addition to the LDT link which links all LDTs in memory into the structure described in the device I/O section, all LDTs have a pointer to the physical device table of the device to which the LUNO is assigned. LDTs of LUNOs assigned to files point to the disk PDT of the volume (drive) that contains the file. All LDTs also have a pointer to the task status block of the task opening the LUNO.

File LDTs (LDTs of LUNOs assigned to files) also have two additional pointers. All of the LDTs of LUNOs assigned to the same file are linked on a common list. Also, each file LDT has a pointer to the file control block (FCB) of the file to which the LUNO is assigned. A file control block is another data structure maintained in the system table area to describe a file (see Section 6). For every file that currently has LUNOs assigned to it (i.e., is being referenced), file utility maintains an FCB tree for all pathname components (i.e., higher level directories) of the files pathname. For example, if a LUNO is assigned to the file VOLUME4.USER05.SOURCE.DEBUG, an FCB for USER05, SOURCE,and DEBUG will be in memory, linked together. The LDT for the LUNO assigned to DEBUG points to the DEBUG FCB, and is also linked on the list of LDTs for the file (see Figure 1-19). Note that the FCB for the volume directory (VCATALOG), an assumed component of every pathname, is always in memory when the volume is installed. A pathname can have a maximum of 49 bytes, which is 1 byte for the length of the pathname and 48 bytes for the pathname itself.

When a LUNO is released, file utility searches the FCB/LDT tree for the LDT of the LUNO being released. The LDT is delinked from the various lists it is on and released to the free system table area list; any blocking buffers associated with the LDT are released. If the LDT is linked to a file FCB, file management checks to see if any more LDTs are linked to the file. If not, the file must not be used currently, and the FCB is delinked from the FCB tree and released to the system table area. The search continues up the FCB tree. As long as an FCB has no LDTs linked to it (i.e., no LUNOs assigned to the file), and has no descendant FCBs (i.e., if the FCB represents a directory file that is not being accessed and has no cataloged files being accessed), it is delinked from the FCB tree, and the release LUNO search continues at the parent FCB.

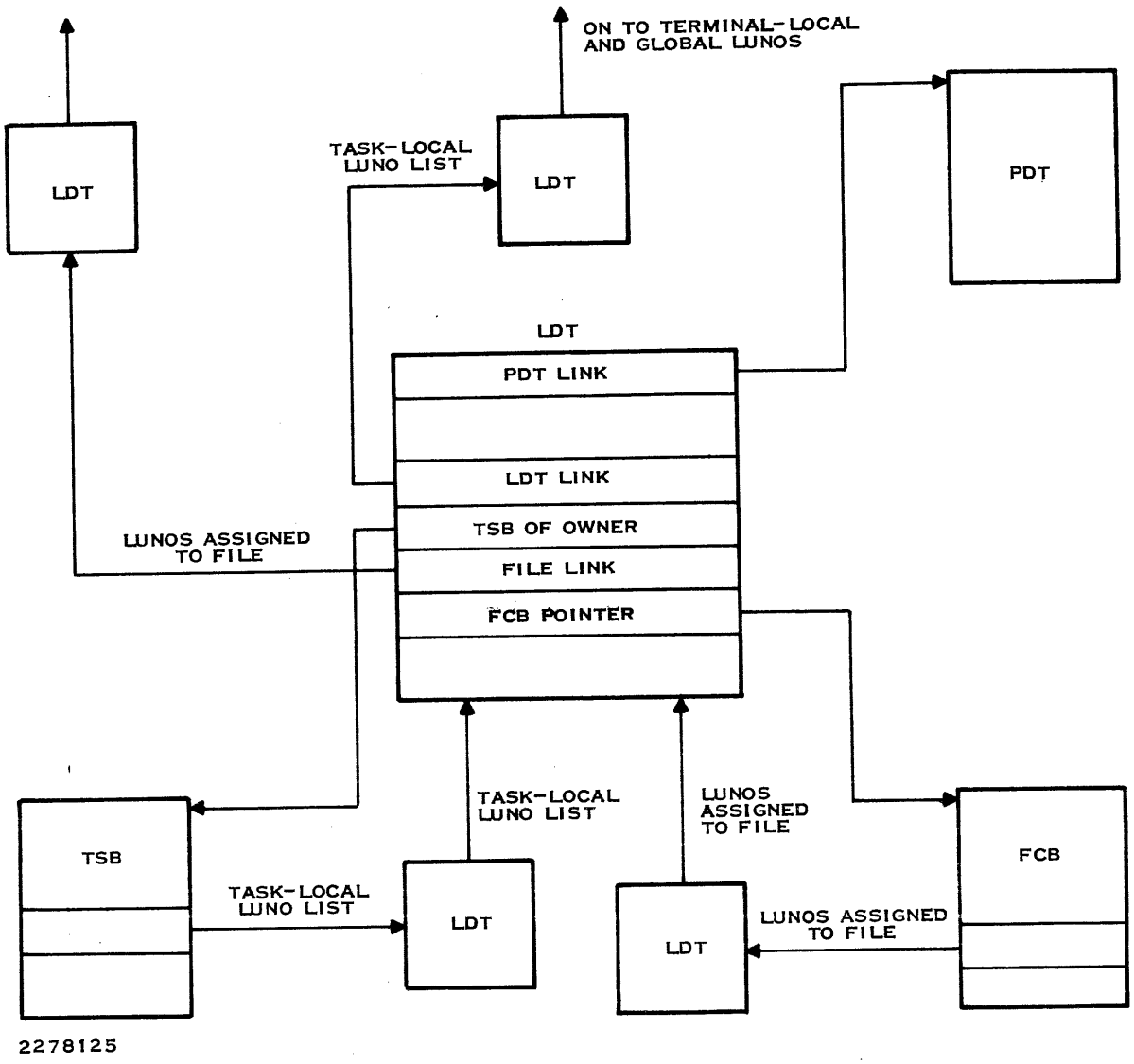


Figure 1-18 Logical Device Table Pointers

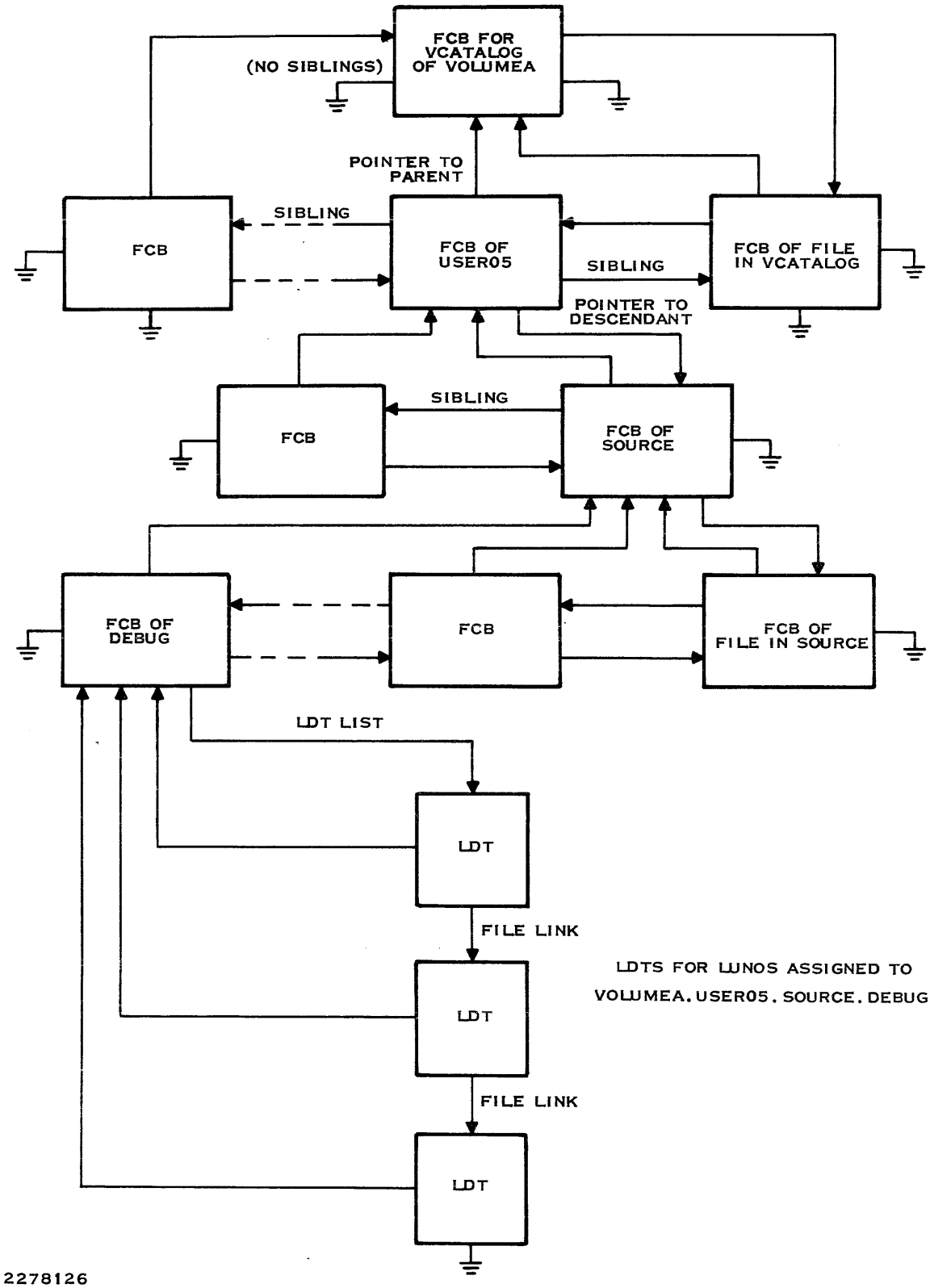


Figure 1-19 FCB and LDT Tree

1.2.5.2 Creating and Deleting Files.

Creation of a file under DX10 involves the following process:

1. The FCB tree of the directory under which the file is to be created is built in memory. For example, if the file VOLUMEA.USER05.SOURCE.DEBUG is being created, an FCB must be in memory for the VCATALOG, USER05, and SOURCE directories. This structure is necessary in order to access the directory VOLUMEA.USER05.SOURCE on the disk. A pathname can have a maximum of 49 bytes, which is 1 byte for the length of the pathname and 48 bytes for the pathname itself.

2. File utility searches the disk directory to see if the file being created already exists. If so, an error is returned; otherwise, processing continues.

3. A file descriptor record, which is a directory entry (see paragraph 4.3.4.2) for the file being created, is built in memory, and inserted into the directory on the disk.

Deletion of a file involves a process similar to the one described above. The FCB tree must be built in order to access the directory in which the file is cataloged (note that the FCB tree may already by in memory). The file descriptor record in the directory is released and made available. The FCB representing the deleted file is released if it is currently in the system table area. In addition, the path up the FCB tree from the deleted file is searched for FCBs with no descendants (directories which are no longer being accessed). Any such FCBs are delinked from the FCB tree and released to the system table area.

1.2.6 FILE I/O FLOW.

The initial processing of a file I/O SVC is also similar to that for device I/O. The I/O supervisor determines from the I/O opcode that the call is for file management services. It then tests the I/O request to see if a "fast transfer" is possible. A request is eligible for "fast transfer" if the following conditions are met:

...The file is not opened for unblocked I/O

...The I/O is for a sequential or blocked relative record file

...The operation is a read or write (not forced write)

...The file is not currently being accessed

...The record desired is already buffered in memory (i.e., it has recently been accessed and the buffer has not yet been destroyed).

If all of the conditions are met, control is transferred to the file management read or write routine, the desired I/O is performed directly between the file buffer in memory and in the calling task's data buffer, and control returns to the calling task. Thus, the I/O operation is performed by XOP level code, and the calling task is not suspended. If the conditions are not met, the call block is buffered into the system table area and queued for file management. The file management request queue is served by the number of replicas of the file management task, FM$TSK, specified during system generation. The file I/O queuing routine searches a list of file management tasks for a suspended task, and activates one if possible.

FM$TSK is the main driver for file I/O processing. It dequeues an entry from the queue, and checks to see if the file is already being accessed by another FM$TSK. If so, it queues the request on a queue of I/O requests for that file, and dequeues another entry from the file management request queue. If not, FM$TSK checks the I/O opcode, and tranfers control to the correct processor. If the file being used is a key indexed file, control transfers to the key indexed file I/O driver, KI$BEG. When the processor returns, FM$TSK unbuffers the call block and key indexed file currency information, if necessary, into the calling task, and reactivates the task. It then checks for more queue entries, first on the queue for the same file, and then on the file management request queue. If more exist, it processes them. When the queue is empty, FM$TSK suspends via SVC >24. Figure 1-20 shows a flowchart of the top level of file I/O processing.

Some of the opcode processors (e.g., FMOPEN for open calls) do no more than access the logical device table (LDT) assigned by the calling task to the file. The read and write processors work in conjunction with buffer management to transfer data between the disk file and user buffer. The following paragraphs describe the file buffering scheme used by DX10 file management.

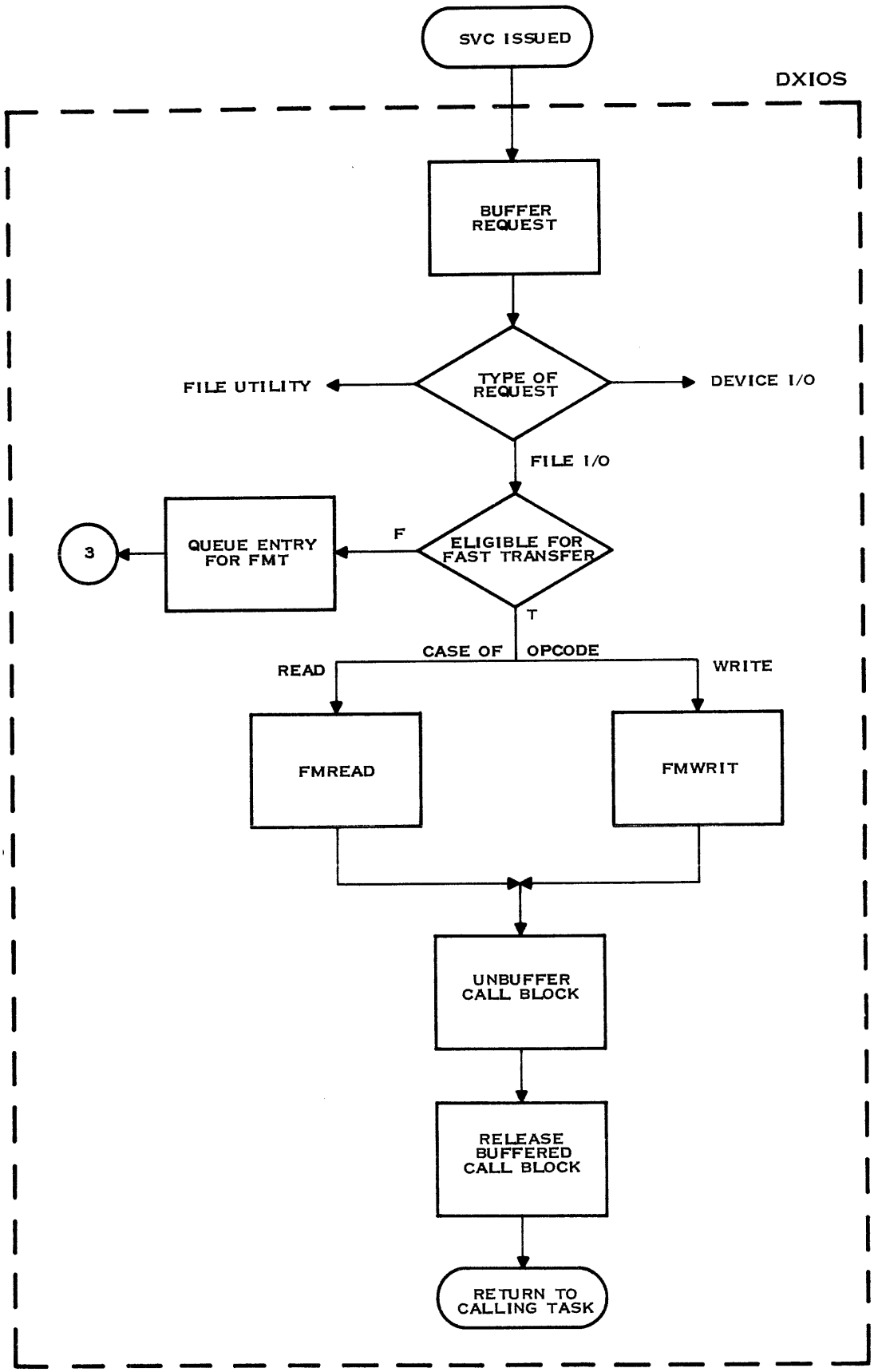


Figure 1-20 File I/O Flow

1.2.6.1 Blocked File I/O.

I/O operations to blocked files (any type of file except unblocked relative record - see Section 4 for a description of file types) are handled through a group of routines called buffer management.

To access a specified logical record of a file, the file physical record that contains the desired logical record is read into a blocking buffer. These buffers are allocated from user memory, and are linked onto the time ordered list when not being used. The buffers are linked and delinked, read and written, created and released by buffer management routines.

When file management receives an I/O request for a particular logical record of a file, it calls a buffer management routine to get the buffer that contains the physical record which in turn contains the desired logical record. If the specified physical record is already in a buffer on the TOL, the buffer management routine simply delinks the correct buffer and maps it into the file management task; if not, the buffer management routine creates a new buffer, reads the specified physical record from the file, and maps it into the file management task.

The file management processor also calls a buffer management routine, BM$MAP, to map the calling task's data buffer into file management. Having both buffers mapped in allows file management to avoid using long distance instructions when transferring data between the two buffers.

After the file management processor has completed transfering the specified logical record, it calls another buffer management routine to release the file blocking buffer, which is relinked onto the head of the TOL.

The use of blocking buffers in DX10 is an attempt to reduce the actual number of disk accesses required to perform a given number of file I/O operations. Since buffers are not immediately released, but rather stay on the TOL until their memory is required by memory management, a read or write request to a blocked file record may be made to a buffer already in memory, rather than necessitating a disk access.

1.2.6.2 Unblocked File I/O.

I/O operations to unblocked files (i.e., program, image, directory, unblocked relative record files) do not use intermediate blocking buffers. The file management processor calls BM$MAP to map in the calling task's data buffer, and then calls the file management disk I/O routine, FM$IO, to transfer the record directly between the disk and data buffer.

1.2.7 TASK TERMINATION.

There are four ways for a task to terminate execution:

* Issue an End Task or End Program SVC
* Issue a Suspend Awaiting Queue Input SVC (system tasks only)
* Create an error and go into end action
* Be killed by another task issuing a Kill Task SVC

All terminated tasks, except tasks which are suspended awaiting queue input, have their memory released and their TSBs queued for the termination task, TM$DGN.

TM$DGN performs such general clean-up actions as:

...closes LUNOs opened by the terminated task

...releases task local LUNOs assigned by the terminated task

...delinks the TSB from its family tree structure

...activates the parent task if specified

...sends any error message to the system log

...clears any breakpoints for the terminated task

...releases the TSB, if the task is not memory resident.

The following paragraphs describe what happens to a task at termination.

1.2.7.1 End Task/End Program SVC.

The End Task and End Program SVCs are both processed by a routine in the module TM$FUN. This routine checks the TSB of the executing task (i.e., the task issuing the SVC) for any outstanding I/O. If any exists, it is aborted, and the task is left on the active queue in order to allow the device driver task to process the task's end-of-records. The kill flag in the TSB is set, to notify the task scheduler that the task has been terminated.

When a task's kill flag is set, and it has no outstanding I/O, either the scheduler or the end task/end program routine queues the tasks TSB for processing by the termination task, TM$DGN. If the task is not memory resident, its memory is released.

1.2.7.2 Suspend Awaiting Queue Input SVC.

This SVC is also processed by a routine in the TM$FUN module. This routine simply enters a state >24 in the task's TSB, and resets the task's PW and WP register values to restart it. The task's memory is not immediately released, nor is its TSB queued for the termination task. Since the task is not processed by TM$DGN, it should release all task local LUNOs before issuing this SVC.

1.2.7.3 Error Termination.

When a task causes an internal error interrupt (e.g., address out of range, memory parity error), it is forced to go into end action. If the task has no end action, or when the end action terminates, the tasks memory is released and its TSB is queued for the diagnostic task.

1.2.7.4 Kill Task SVC.

When a task is killed, it is forced into end action, and then queued for the diagnostic task.

SECTION 2

ORGANIZATION AND STRUCTURE OF DX10 SOURCE LIBRARIES

2.1 GENERAL

This section is intended as a guide to help the user search a DX10 source disk for a particular module. It contains a tabularized description of the top level directories and their contents. A disk map of the DX10 Operating System source disk is contained in the Product Documentation Package manual for the DX10 source (part number 2250958-0001).

The DX10 source and object modules are cataloged under a directory structure that is generally organized as follows:

- ... The level one directories (directories under VCATALOG) break the DX10 code into specialized sections (e.g., task manager, disk-manager, GEN990, key indexed file processor).
- ... Each level one directory generally contains two sub-directories, OBJECT and SOURCE.
- ... The OBJECT and SOURCE directories contain the object and source modules of DX10 routines associated with the general function implied by the level one directory name (e.g., task management routines, memory management routines).

2.2 TOP LEVEL DIRECTORIES

Table 2-1 gives the names of the top level source directories (cataloged directly under VCATALOG) and a brief description of the contents of each one.

Table 2-1 Top Level Directories -- Part 1 of 2

Directory Name	Contents
ANALZ	The routines that make up the system crash analyzing utility, ANALZ.
BATCH	Several SCI batch command streams used to build a DX10 system disk.
BDLINK	Link Editor control streams, link maps, and linked object used by the disk build utility.
DEBUGR	The routines that make up the debugger.
DEVDSR	The device service routines for all supported devices.
DSCBLD	The routines that make up the disk build utility.
DSCMGR	The disk manager routines; the main driver is DM$TSK.
DXIO	The DX10 I/O routines, excluding file management and file utility; DXIOS is the main driver (i.e., processes all code 0 SVCs.)
DXLINK	Link Editor control streams, link maps, and linked object of various parts of DX10.
DXMISC	Miscellaneous routines and modules within DX10, including: D$DATA, DXDAT2, SVC processors, boot loader.
DXUTIL	DX10 2.X to 3.1 disk conversion utility routines.
FILMGR	File management routines; the main driver is FM$TSK. The routines for handling key indexed files and program files are cataloged elsewhere.
FUTIL	File utility routines; the main driver is FU$.
GEN990	Routines and data files that make up GEN990, the system generation program; the main driver is GEN.
GENPLINK	Link Editor control stream, link maps and linked object used by GEN990 to generate a system.
KIFILE	Key index file managing routines and overlays; the main driver is KI$BEG.
LINKER	Batch streams, control file, and Link Editor routines.
MEMMGR	Memory management and buffer management routines.

Table 2-1 Top Level Directories -- Part 2 of 2

Directory Name	Contents
NOSHIP	Dummy debugger.
PATCH	Patch files for sysgen and system program file.
PGFILE	Program file handling routines.
SCI990	Batch streams and control streams to generate SCI, and SCI routines.
SDSMAC	Batch streams and control streams to generate the macro assembler, and the assembler routines.
SYSTEM	Templates of system tables and data structures.
SYSTSK	Various system tasks, including: MVI, install/unload volume (IVOL, UVOL), initialize disk (IDSC), diagnostic task (TMDGN).
TI	The assembly language test program.
TSKMGR	Task management routines, including: scheduler, loader, rolling routines, etc.
TXLINK	Link editor control streams, link maps, and object for the TX/DX file conversion routines.
UTCOMN	Common routines used by various DX10 utilities.
UTDIRP	Directory utility routines (e.g., for copy directory, delete directory, backup directory, restore directory).
UTDXTX	DX/TX file conversion routines.
UTGENR	General utility routines, including DCOPY, SIS, STS, CKS, IDT, MAD, SAD, etc.
UTLINK	Link Editor control streams, link maps and object for linking most of the utility routines.
UTSVC	More utility routines, including Map Disk.

SECTION 3

SYSTEM LOADERS

3.1 GENERAL

The software for loading and initializing the DX10 operating system includes: a program image loader on track 1 of all initialized disks, a specialized loader for loading DX10 images and initialization of various parts of the operating system, and a system restart task which further initializes DX10 and which bids any user specified restart task (GEN990 ID parameter).

In addition, the new ROM bootstrap loader (multiwire - PN 945134-0013; printed circuit - PN 945134-0014) which can load a program from cards, cassette, magnetic tape, or disk, is used to load the program image loader.

The normal sequence for loading a system image is:

1. Initiate the boot loader (by pressing the HALT, RESET, and then LOAD buttons on the front panel)

2. The boot loader loads the disk program image loader, starting at location >A0.

3. The disk program image loader determines where the end of its address space (highest address) is, relocates itself to the high end of memory, then reads in the special system image loader, starting at location >A0.

4. The system image loader relocates itself to the high end of memory, loads the DX10 image specified in the disk volume information (track 0, sector 0), initializes system variables, bids the system restart task, and transfers control to the operating system.

5. The system restart task, executing under control of the operating system, performs more initialization of the operating system and then bids the user specified restart task, if one was specified when the loaded system was generated.

The following paragraphs describe each separate loader and restart task in detail, as well as options in the loading sequence that may be used.

3.2 THE BOOT LOADER

The bootstrap loader which is contained in ROM, is capable of loading a program from either cards, cassette, magnetic tape, or disk. The loader can be programmed to load from any of these media by inserting values in the boot workspace (memory locations >80 - >9F) via the front panel of the 990/10 computer.

Location >80 is used to specify the loading device, according to the following rules:

* If the content of >80 is positive, load from the card reader at the preferred location (CRU address >40).
* If the content is zero, load from a cassette at the preferred location (CRU address 00).
* If the content is negative, load from the TILINE device (magnetic tape or disk) at the address specified in location >82.

If the loading device is a TILINE device, locations >82 and >84 are used to specify the TILINE address and unit select, respectively. The unit select value specifies which device on a multiple-device controller is to be used. For magnetic tape controllers, the following values should be used:

>8000	-	unit 0
>4000	-	unit 1
>2000	-	unit 2
>1000	-	unit 3

For disk controllers, the following values should be used:

>0800	-	unit 0
>0400	-	unit 1
>0200	-	unit 2
>0100	-	unit 3

The default values, which are inserted into these locations by pressing the HALT button on the front panel, cause the ROM loader to boot from disk unit 0 at TILINE address >F800. Note that when loading from a disk, the ROM loader always loads the program image loader from track 1; however, when loading from cards, cassette, or magnetic tape, it loads whatever program image is read from the device.

3.3 THE DISK PROGRAM IMAGE LOADER

The loader SYSLD, which resides on track 1 of every DX10 3.0 formatted disk, is capable of loading any standalone program from an image file on disk into memory. After the program is loaded by the boot loader at location >A0, it relocates itself to the high end of its address space (e.g., the high end of 32K words). It then determines what program to load by using the volume information in sector 0 of track 0 on the system disk (see paragraph 4.2.1 for a description of the volume information).

The program to be loaded is chosen according to the following rules:

1. If the diagnostic flag in the volume information is Y (i.e. non zero), the diagnostic task, which can be any standalone task, will be loaded, and the flag reset to N (zero).

2. If no diagnostic is specified, the loader checks to see if the file pathname of either a primary or a secondary system loader is specified. If so, the image loader loads whichever system loader is indicated by the flag (0=load primary, 1=load secondary, -1=load secondary and reset flag to zero).

3. If no system loader is specified, the image loader loads the system image indicated by the image flag, which is used in the same manner as the system loader flag.

The program image loader normally loads a program image starting at memory location >A0. This default load bias is stored in the second word of the loader, and may easily be changed by a Modify Absolute Disk (MAD) command. Note that, since the image loader neither changes memory mapping nor uses long distance instructions, it always loads in its own address space. When the loader is booted by the ROM loader, it is always mapped into the first 32K words of memory.

3.4 THE SYSTEM LOADER/INITIALIZER

The DX10 system loader, module STARTR, is normally located on file .S$LOADER. This program assumes it has been loaded starting at location >A0 by the program image loader, and relocates itself to the high-order 8K bytes of physical memory. It then determines the name of the operating system to be loaded from the program file .S$IMAGES, by reading the volume information on track 0, sector 0. The system select flag indicates which system image is to be loaded (0=primary, 1=secondary, -1=load secondary and reset flag to zero).

After obtaining the name of the system to be loaded, the system loader searches .VCATALOG and finds the program file, .S$IMAGES. It then loads the system image from disk, starting at location >A0.

After loading the system image, STARTR performs the following initialization functions:

* Initializes the map files for all of the system segments.
* Renames the disk drives (if necessary) to make the system disk DSO1.
* Initializes memory beyond the end of the loaded system to a constant pattern (>B00B).
* Initializes memory size parameters.
* Initializes the interrupt and XOP trap vectors in memory locations >00 - >80.
* Creates file control blocks and logical device tables for VCATALOG, the system program file, the system overlay file, and the roll file.
* Enters the power up code of each device service routine in the system.
* Loads memory resident procedures and bids memory resident tasks.
* Initializes free memory pointers.
* Bids the system restart task, SYSRST.

After initialization, the system loader releases control to the task scheduler.

3.5 THE SYSTEM RESTART TASK

When the operating system starts up, the first task scheduled for execution is the system restart task, SYSRST. This task performs initialization functions that are easier to do under a running operating system than in the system loader. SYSRST also bids the user specified restart task if there is one (the user restart task must be on the system program file, and is specified during system generation).

The initialization functions performed by SYSRST are:

* Delete all temporary files on the system disk
* Assign global LUNO >10 to the language program file, S$SDS
* Assign global LUNO 1 to the foreground TCA file, S$FGTCA
* Assign global LUNO 2 to the background TCA file, S$BGTCA
* Assign global LUNO 3 to the master TCA library file, S$TCALIB
* Enables the SCI bidding logic within the operating system
* Bids the system log command processor to start file logging.

SECTION 4

DISK ORGANIZATION

4.1 DISK FORMAT

Under DX10, all tracks on disks are initialized in one sector per record format. Note that this record is a disk characteristic and is not the same as the physical record size specified when creating files.

DX10 disks are logically divided into allocable disk units (ADUs), as described in the DX10 Operating System Systems Programming Guide. An ADU is defined to be an integral number of sectors on the disk, the number of sectors per ADU varying according to disk size (see table 4-1), such that the number of ADUs is less than 65,536 (i.e., each ADU on the disk can be addressed in a 16-bit word) and the sectors (ADU is 1 or a multiple of 3). ADUs are numbered from zero, with the first ADU starting on track 0, sector 0.

Table 4-1 Format Information for Supported Disks

Disk Type	No. of Sectors	No. of ADUs	Sectors/ ADU	Bytes/ ADU
DS10	16320	16320	1	288
DS25	77520	25840	3	864
DS31/DS32	9744	9744	1	288
DS50	154850	51616	3	864
DS200	588430	65381	9	2592
CD1400/32				
Removable	52544	52544	1	256
Fixed	52544	52544	1	256
CD1400/96				
Removable	52544	52544	1	256
Fixed	262716	43786	6	1536

4.2 PHYSICAL ORGANIZATION OF THE DISK

To prepare the disk for use, surface analysis and initialization of the disk must be performed. Surface analysis is performed by using the IDS (Initialize Disk Surface) command. After execution of this command, the disk state word in track zero, sector zero contains a value of two. Additionally, bad tracks (physical imperfections) on the disk are indicated. Each bad track is indicated in pairs of words. The first word indicates the first of any contiguous group of bad tracks, and the second word indicates the number of contiguous tracks. Initialization

of a disk is performed by using the INV (Initialize New Volume) command. When a disk is initialized, the disk state word in track zero, sector zero, contains a value of three, indicating that the disk is now ready for use. Bad disk areas are indicated by ADUs in pairs of words. The first word contains the ADU address of the first of any contiguous group of bad ADUs. The second word contains the address of the last ADU in the group.

All disks that have been initialized under DX10 have the following physical layout:

* Track 0, sector 0 -- contains information about the disk volume, such as the volume name and pointers, to the volume directory (VCATALOG).

* Track 0, sector 1 -- contains a list of bad (in the sense of physical imperfections) areas on the disk. Each entry is two words: the first word is the address of the first bad ADU; the second word is the address of the last bad ADU. A zero word terminates the list.

* Track 0, sector 2+ -- the remainder of track 0 contains disk allocation information, in the form of bit maps.

* Track 1, sector 0+ -- is reserved for the disk program image loader described in Section 2.

* Track 1, penultimate sector -- a copy of track 0, sector 0.

* Track 1, last sector -- a copy of track 0, sector 1.

* The remaining tracks are available for file allocation.

The following paragraphs describe the track 0 information in greater detail.

4.2.1 VOLUME INFORMATION.

The information contained in track 0, sector 0 of all disks intialized under DX10 is called volume information. Figure 4-1 shows the format of the 140-byte block of information.

The following is a list of the volume information contained in track 0, sector 0. Note that some fields have zero values when the disk is initialized.

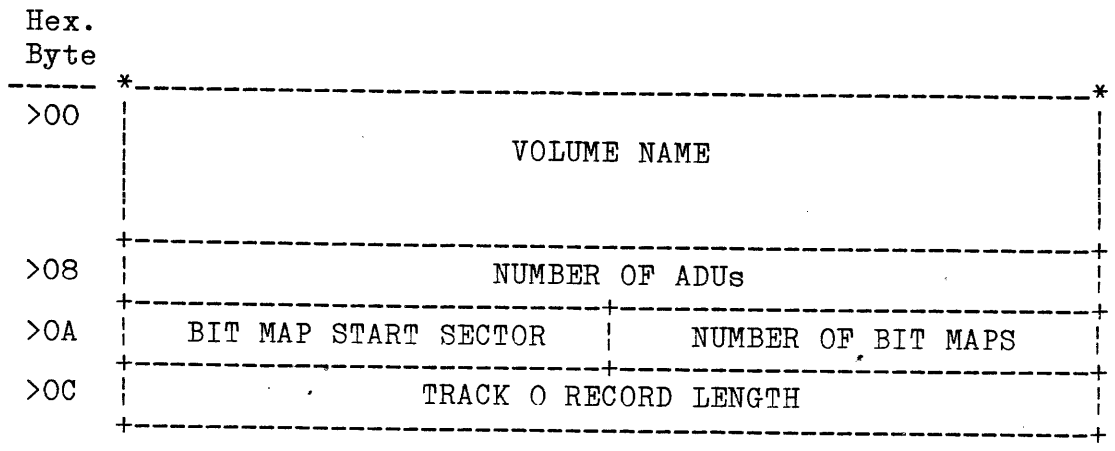


Figure 4-1 Volume Information Format (VIF) -- Part 1 of 4

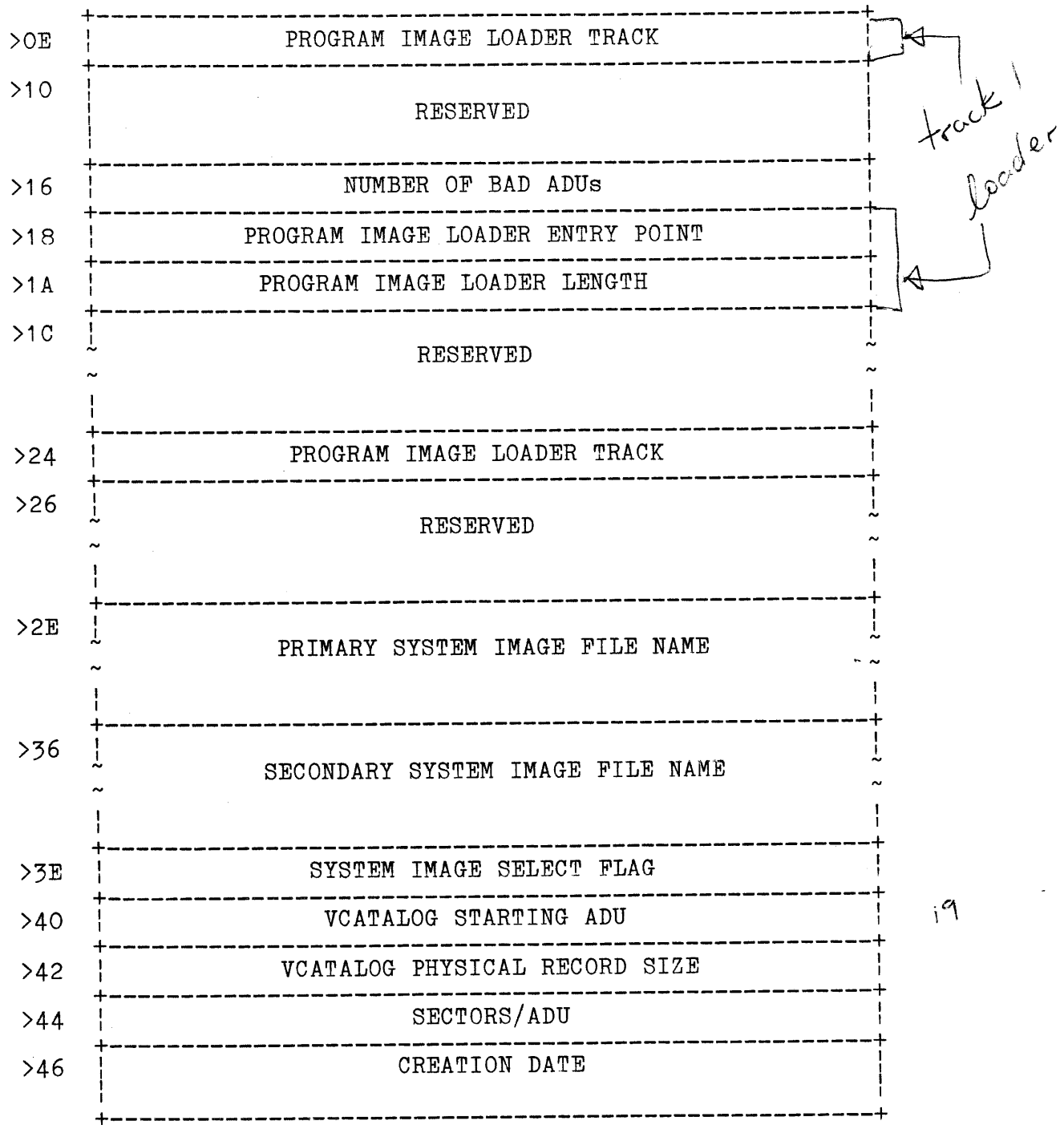


Figure 4-1 Volume Information Format (VIF) -- Part 2 of 4

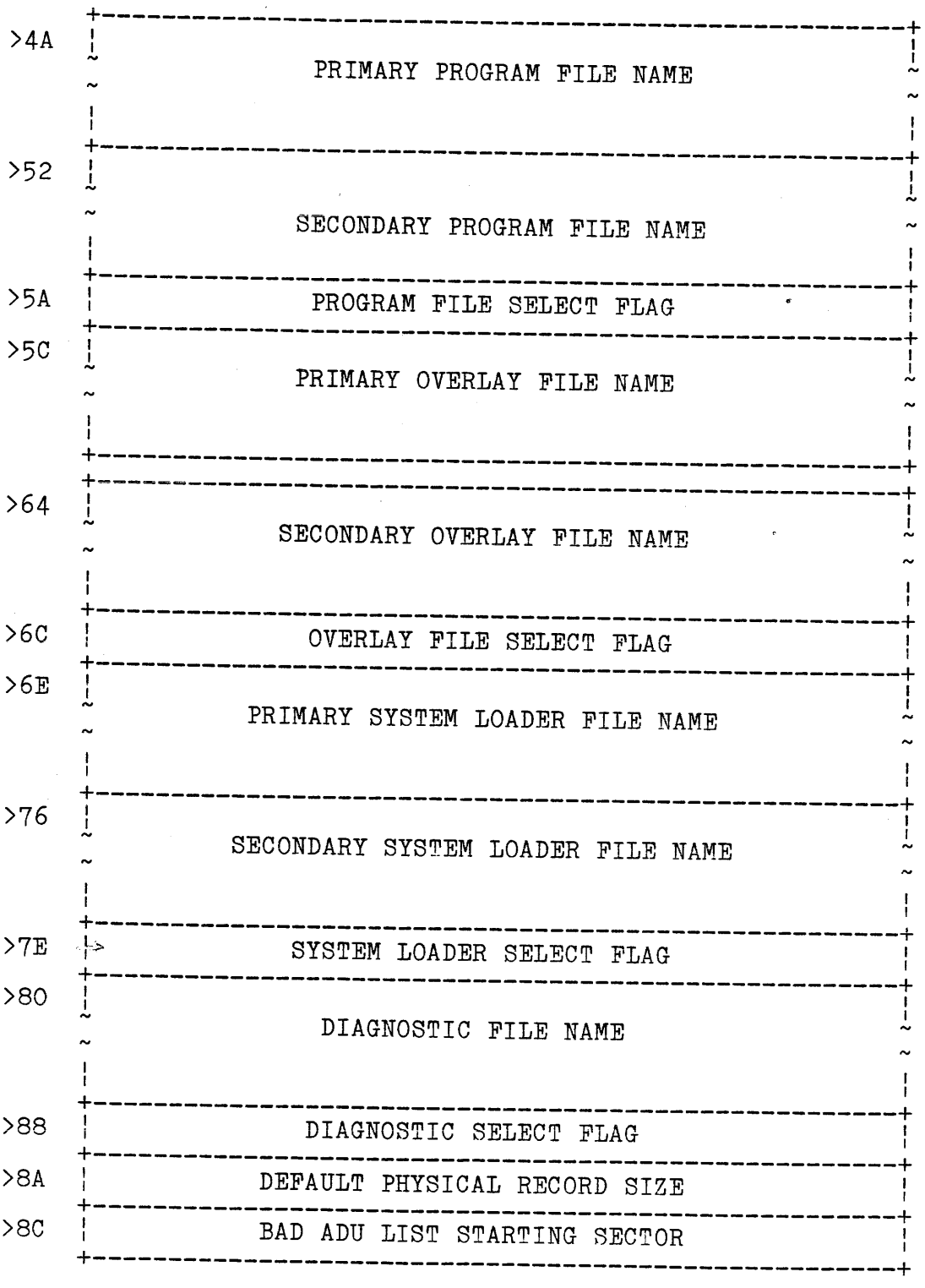


Figure 4-1 Volume Information Format (VIF) -- Part 3 of 4

1 for DX10

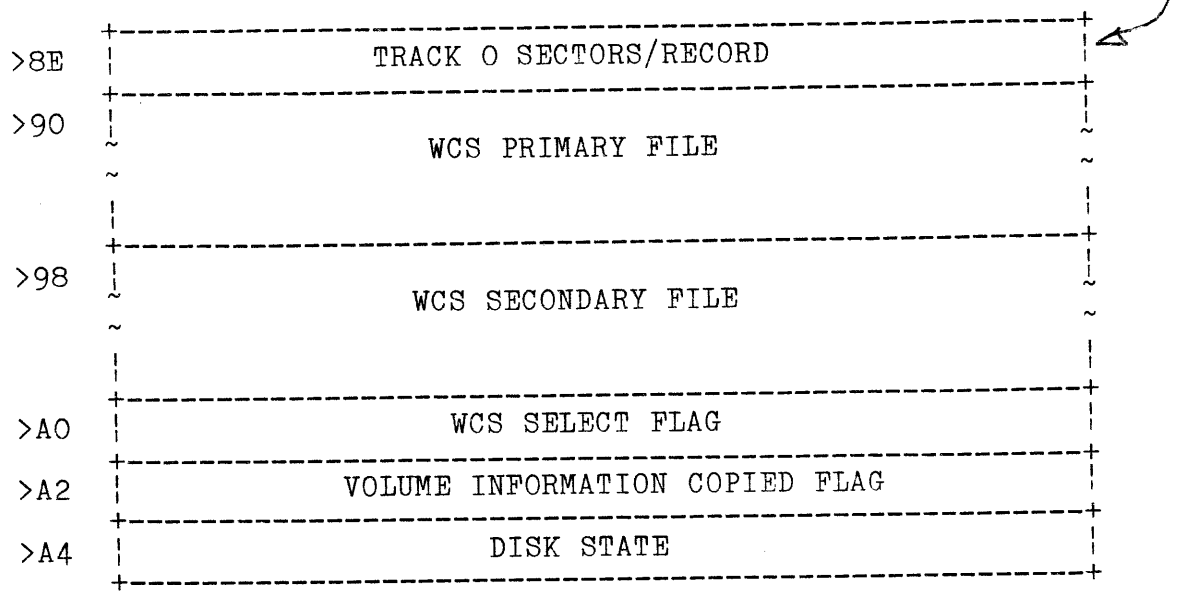


Figure 4-1 Volume Information Format (VIF) -- Part 4 of 4

has track 0, sector 1 & 0
been copied to track 1,
sector n, n-1

1 = copied
0 = not copied

Hex. Byte	Description
>00	A 1-8 character volume name, blank filled to the right.
>08	Total number of allocable disk units contained in the volume. This field varies by disk type.
>0A	The number of the sector on track 0 in which the first bit map resides.
>0B	Total number of bit maps.
>0C	The number of bytes per physical record (i.e., sector) on track 0. This value is also disk dependent.
>0E	The number of the track that contains the disk program image loader. This field is initialized to one.
>10	Reserved.
>16	Total number of bad ADUs on the disk.
>18	Entry point address of the disk program image loader (initialized to >A4, the entry point of the loader when it is loaded at location >A0).
>1A	Total length, in bytes, of the disk program image loader.
>1C	Reserved.
>24	Second copy of the number of the track that contains the disk program image loader (initialized to one).
>26	Reserved.
>2E	The 1-8 character name of the primary system image file. Zero at initialization.
>36	Name of the secondary system image file. Zero at initialization.
>3E	System select flag. Zero at initialization.
>40	Number of the ADU in which the volume directory (VCATALOG) begins.
>42	Physical record size of the VCATALOG directory file (initialized to >86 bytes).
>44	Number of sectors per ADU (disk dependent).
>46	Disk creation date.

>4A Primary system program file name.

>52 Secondary system program file name.

>5A System program file select flag.

>5C Primary system overlay file name.

>64 Secondary system overlay file name.

>6C System overlay file select flag.

>6E Primary system loader file name.

>76 Secondary system loader file name.

>7E System loader select ~~file~~. flag

>80 Diagnostic file name.

>88 Diagnostic select flag.

>8A Default physical record size for the disk (not implemented).

>8C Sector in track 0 in which the load ADU list begins (equals 1 for DX10 disks). (Not implemented).

>8E Number of sectors per record on track 0 (not implemented).

>90 Primary writable control storage (WCS) file name.

>98 Secondary writable control storage (WCS) file name.

>A0 Writable control storage select flag.

>A2 Volume information copied flag.

>A4 Disk state IDS DONE
INU DONE

4.2.2 Allocation Bit Map

To keep track of which areas on the disk are allocated and which areas are free, the DX10 disk manager maintains a bit map of allocated ADUs. The bit map is located on track 0 of each disk, starting at sector 2 and continuing through as many sectors as necessary.

The bit map is divided into 128-word partial bit maps. Each partial bit map is located in a separate sector on track 0. The first word of each partial bit map contains the number of th ADU that begins the largest block of free disk space located in that part of the disk that is mapped by the partial bit map. Each bit in the remaining 127 words represents an ADU. If the bit is zero, the ADU is free; a one bit indicates that the ADU is allocated (or bad).

Figure 4-2 shows a partial bit map. Note that, since each partial bit map contains 127 16-bit words of information, it maps 2032 ADUs.

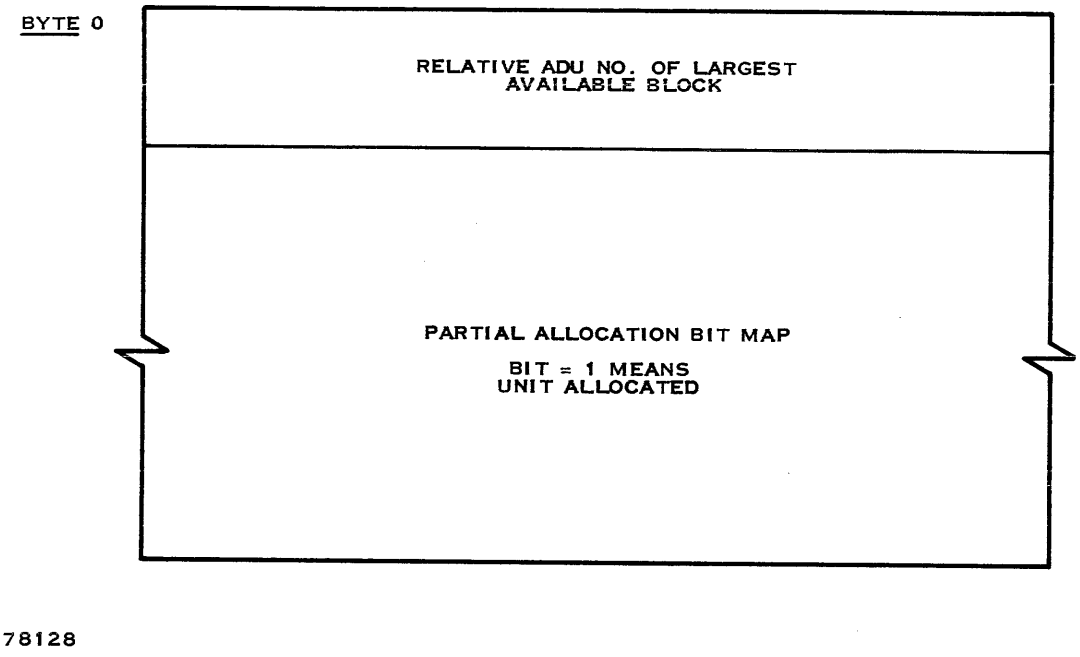


Figure 4-2 Partial Bit Map

4.3 FILE STRUCTURES

DX10 supports three file types: relative record files (block and unblocked), sequential files, and key indexed files. All file types are based on the unblocked relative record type, with extra system overhead needed to implement sequential and key indexed files. In addition, there are three special usages of the relative record file: program files, directory files, and image files.

In the following discussion of file types and file structures, a physical record of a file is the amount of data actually transferred by the operating system during an I/O operation to the file; a logical record of a file is the amount of information the user desires to transfer in an I/O operation. The ratio of the physical record size to the logical record size is called the blocking factor.

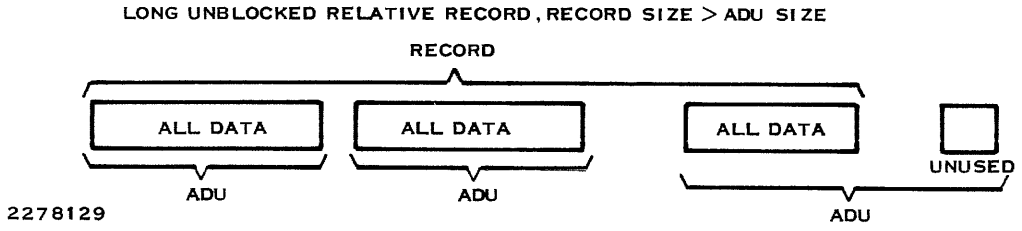
4.3.1 RELATIVE RECORD FILES.

A relative record file is a file in which all logical records are a fixed length and each record can be randomly accessed by its unique record number. Relative record files may be unblocked (logical record size equal to physical record size) or blocked (logical record size less than physical record size).

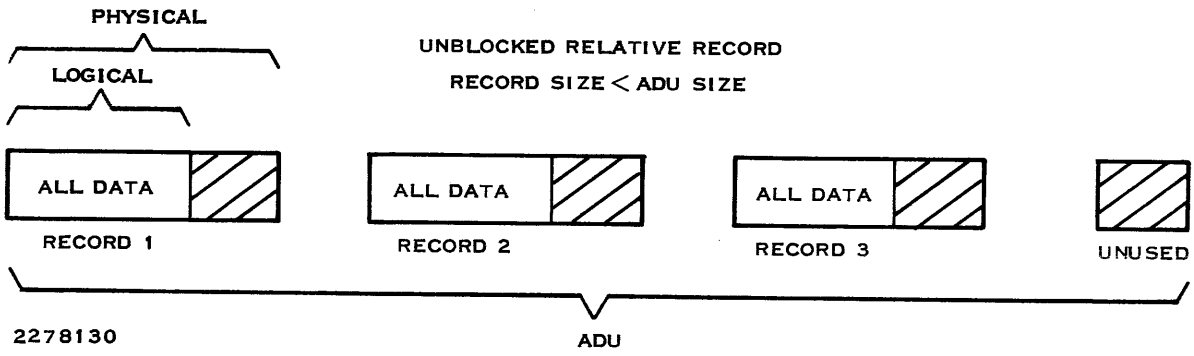
4.3.1.1 Unblocked Relative Record Files.

Each logical record of a file of this type occupies one physical record of the file. A physical record may be any integral multiple of contiguous sectors. File accesses require reading or writiog this many sectors (reads and writes of multiple contiguous sectors can be accomplished via one disk access). Records read from unblocked relative record files are transferred directly from the disk to the user buffer, without intermediate system buffering. When the user specifies a particular record of the file, the record number is converted by file management to an absolute allocable disk unit number and a sector offset within the ADU. The absolute disk address is then passed to the disk device service routine (DSR) to perform the actual data transfer. The disk DSR converts the ADU and relative sector to a physical track and sector disk address to communicate with the disk controller hardware.

Long Unblocked Relative Record
Record Size > ADU Size



Unblocked Relative Record
Record Size < ADU Size

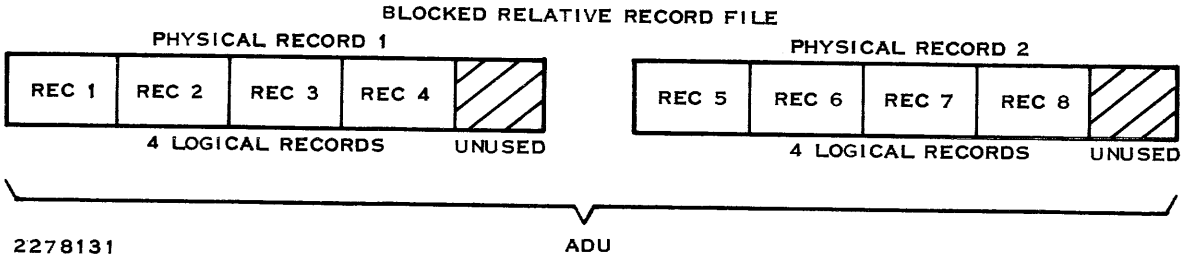


Note that each physical record must begin on a sector boundary and that a physical record that starts in te middle of an ADU may not span the ADU boundary.

4.3.1.2 Blocked Relative Record File.

These files are the same as unblocked except that multiple logical records may be stored in each physical record. Logical records may not span physical records. Records are transferred via intermediate blocking buffers which are furnished from the general pool of user space by buffer management.

Blocked Relative Record File



4.3.2 SEQUENTIAL FILES.

Sequential files are blocked relative record files with variable length logical records. Logical records may span physical record boundaries regardless of ADU boundaries. When a logical record spans a physical record boundary, it is broken into partial records which are contained in separate blocks. The first word of each physical record has two flags indicating whether the first logical record is continued from the

preceding physical record and whether the last logical record is contained in the following physical record. Set flag bits (bit = 1) have the following meaning:

Bit	Meaning When Set
0	First logical record in this physical record is continued from the preceding record.
1	Last logical record in this physical record continues in the next record.

Each logical record or partial record is preceded by a header word and followed by a trailer word. The content of the header and trailer is the number of bytes of user data between them. An end-of-file is signified by a zero length record (zero header and trailer).

A special condition exists when a record or last partial record ends with only one or two words remaining in the physical block. Since there is not room for another partial record (header/data/trailer), the next record will begin in the following block. The last word of the current block contians the number in the last trailer plus the number of unused bytes (two or four). Figure 4-3 shows how a sequential file is arranged.

Logical records of a sequential file may be blank-suppressed (i.e., the sequential file is created blank-suppressed). In blank-suppressed files, all double blanks are removed. A blank-suppressed logical record has the following format:

1. Header word
2. Byte containing a count of words with double blanks*
3. Byte containing a count of words with no double blanks*
4. Data characters*
5. Trailer word

* Items 2 through 4 above are repeated as necessary.

Figure 4-4 shows a blank-suppressed record.

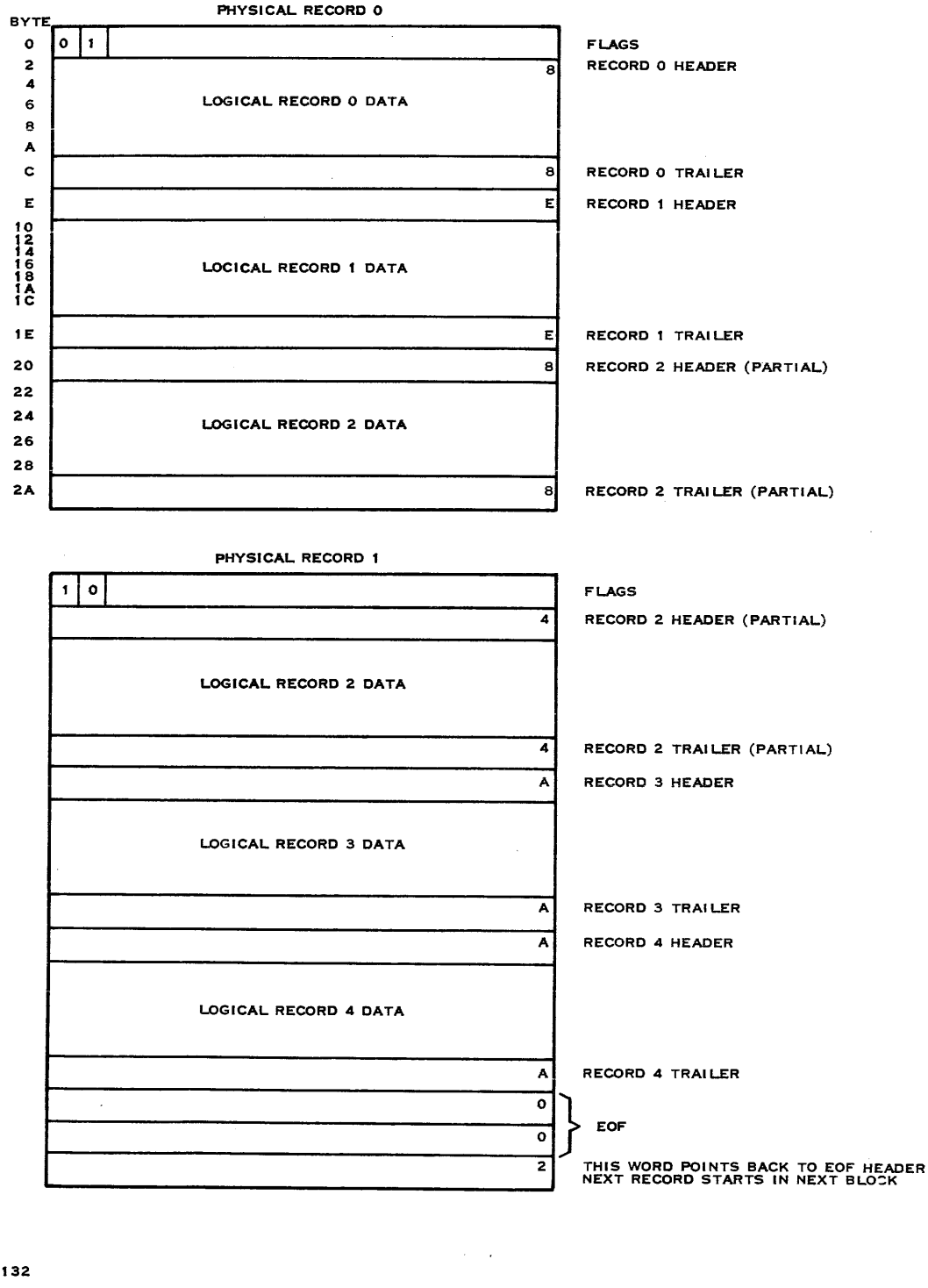


Figure 4-3 Sequential File Format

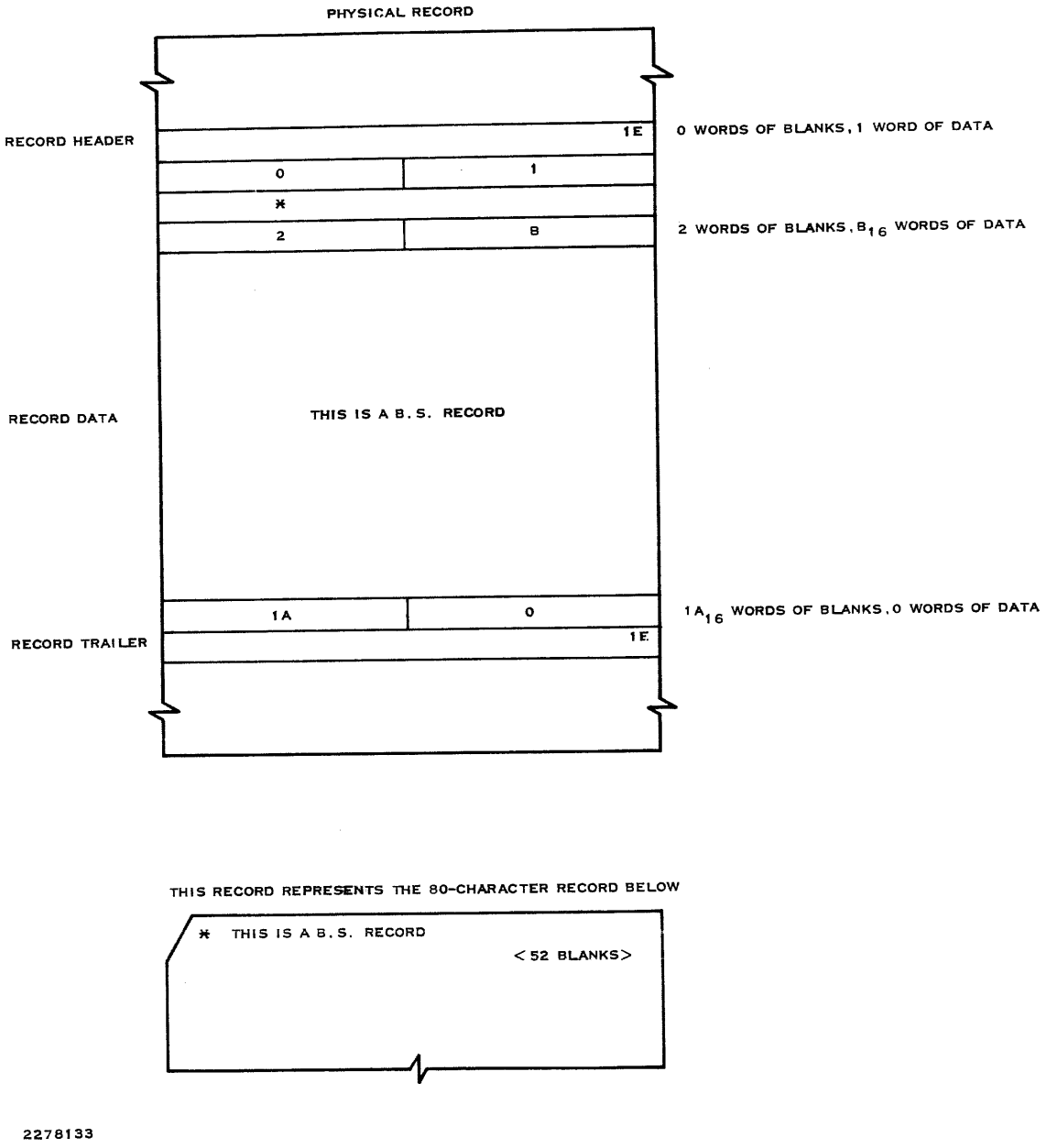


Figure 4-4 Blank-Suppressed Record

4.3.3 KEY INDEXED FILES.

Key indexed files have variable length logical records that can be accessed either randomly, by any one of up to 14 alphanumeric keys, or sequentially, in the sort order of any key. On the disk, a key indexed file with n keys is arranged as follows:

* The first 18n+3 physical records are the KIF log blocks. Before modifying any record within the file, it is written into a log block, to prevent data loss in case of an error (e.g., power failure) during the data transfer. In the event of such an error, the logged image is written back into the original file record, when the file is next opened, and the file operation may be retried.

* The next n physical records are the roots of the balanced trees (B-trees) that are used to locate each logical record within the file by key. There is a B-tree for every defined key (i.e., up to 14 B-trees); therefore, there are n B-tree roots.

* Following the B-tree root nodes are physical records that contain data as well as physical records that contain other B-tree nodes.

The following paragraphs describe the structure of B-trees and data records in detail.

4.3.3.1 B-Trees.

B-trees are made up of a root node, branch nodes, and leaf nodes. Root nodes are just the first node of the tree. Leaf nodes are the nodes that contain pointers to the data records. Branch nodes are all the nodes between the root and leaf nodes.

A B-tree, under DX10, is a multi-way (having multiple branches per node), balanced tree; that is, all leaf nodes are at the same level. DX10 B-trees may not exceed nine levels. Figure 4-5 shows a sample B-tree, in which the key values are single letters.

Each node of a B-tree occupies one physical record of a key indexed file, and is called a B-tree block. Each B-tree block contains a few words of overhead and several pointer/key value pairs. Figure 4-6 shows a B-tree block.

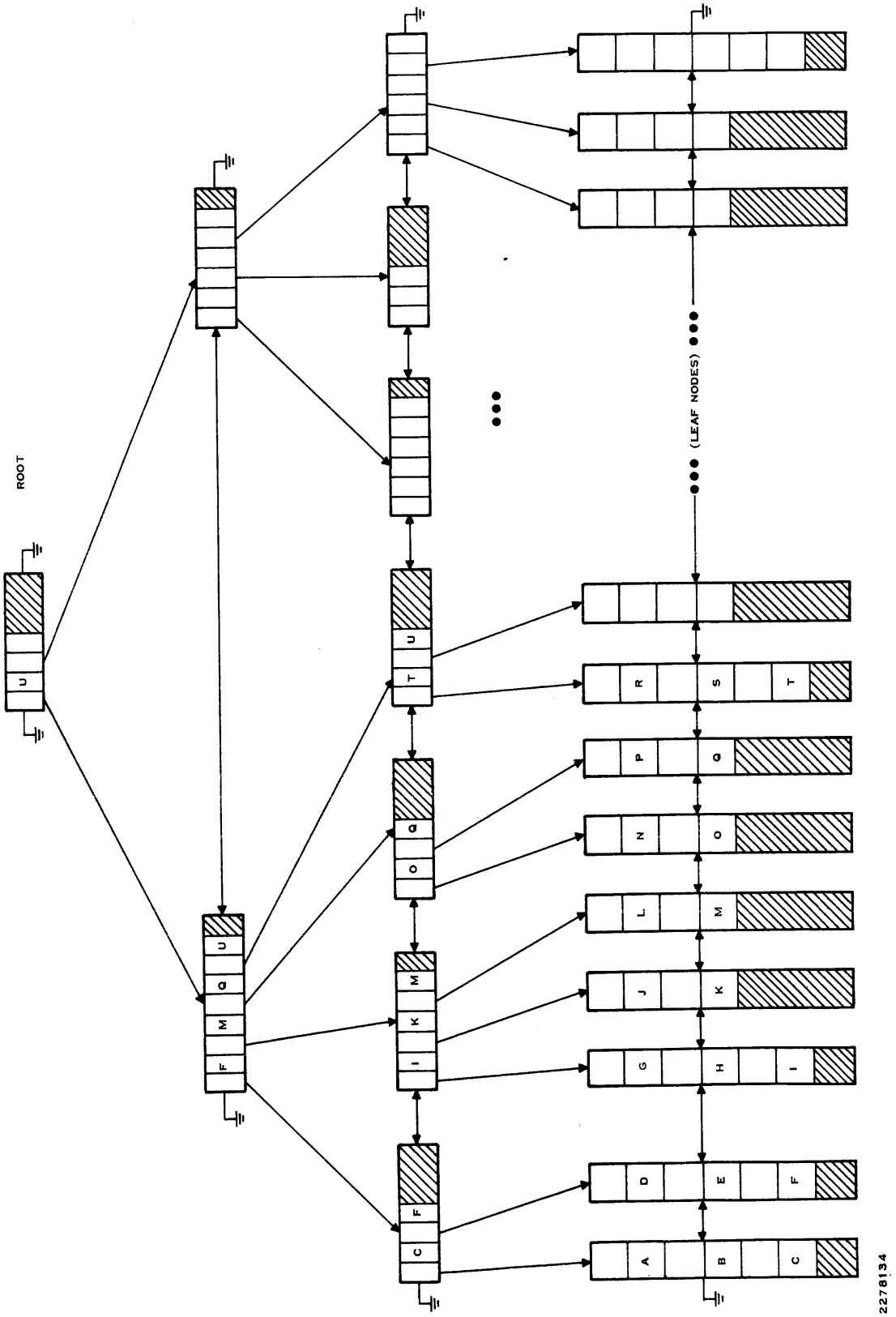


Figure 4-5 Key Indexed File B-Tree

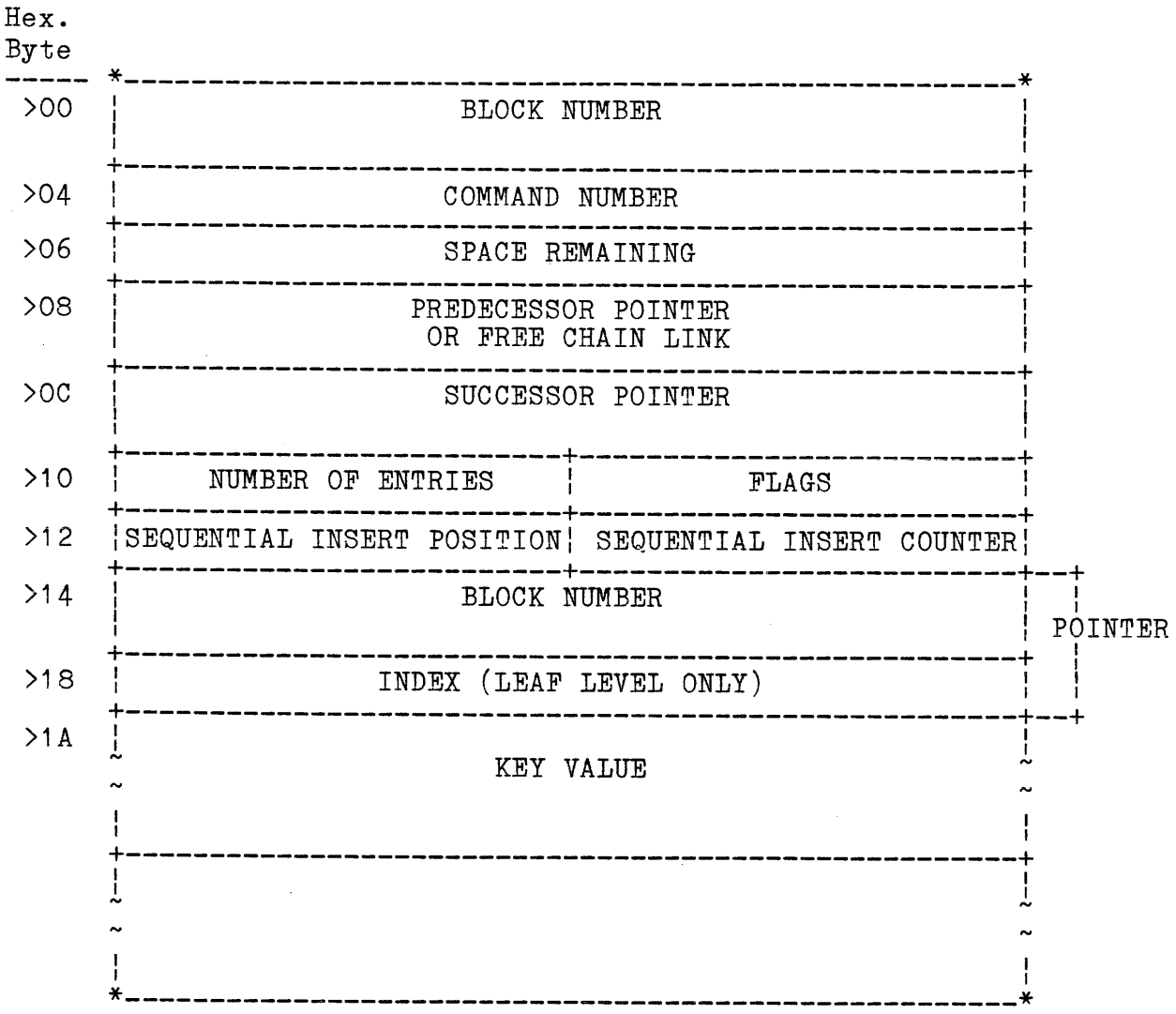


Figure 4-6 B-Tree Block

Hex. Byte	Description
>00	Physical record number of this B-tree block. This field is maintained so that, should a system crash occur while this file block is being modified, the logged image can be restored to the correct file record.
>04	The opcode of the file operation being performed. This field is also maintained for logging purposes.

>06 The number of available bytes remaining to be used in this B-tree block.

>08 If this block is currently being used as a B-tree node, this field points to the preceding node on the same level (zero if this is leftmost node). The address is a physical record number. If this block is available for use, this field points to the next available block (free blocks are kept on a linked list).

>0C If this block is a B-tree node, this field points to the following node on the same level; otherwise, this field is unused.

>10 The number of pointer/key value pairs currently contained in this block.

>11 Flags. Bit 7 is set (byte 17 = 1) if this B-tree block is a leaf.

>12 This byte is zero when the block is initialized due to a B-tree split. When the first entry is made to the block, this byte contains the number of entries in the block that are greater than the new entry. (This applies to sequential placement only; otherwise, byte >14 is moved up here).

>13 When the block is initialized due to a B-tree split, this value is the maximum entries that may be inserted in the block plus one. For each subsequent entry to this block, if the number of entries in the block that are greater than the new entry equals the number in byte >12, byte >13 is decremented by one. When this B-tree block is about to split, if byte >13 is zero, the lower 90% of the entries are in one block and the upper 10% in the other. Otherwise, the split is 50-50. (This applies to sequential placement only; otherwise, byte >15 is moved up here.)

>14 These six bytes are the first pointer. If this is a non-leaf node, the first four bytes contain the record number of a branch or leaf node and the last two bytes are meaningless. If this is a leaf node, the first four bytes contain a record number of a data record and the last two bytes contain the key ID of the logical record within the data record.

>1A The key value of the first pointer/key value pair.

The remainder of the B-tree block contains more pointer/key value pairs. These entries in a B-tree block are kept sorted in increasing order of key value (smallest key value is first entry).

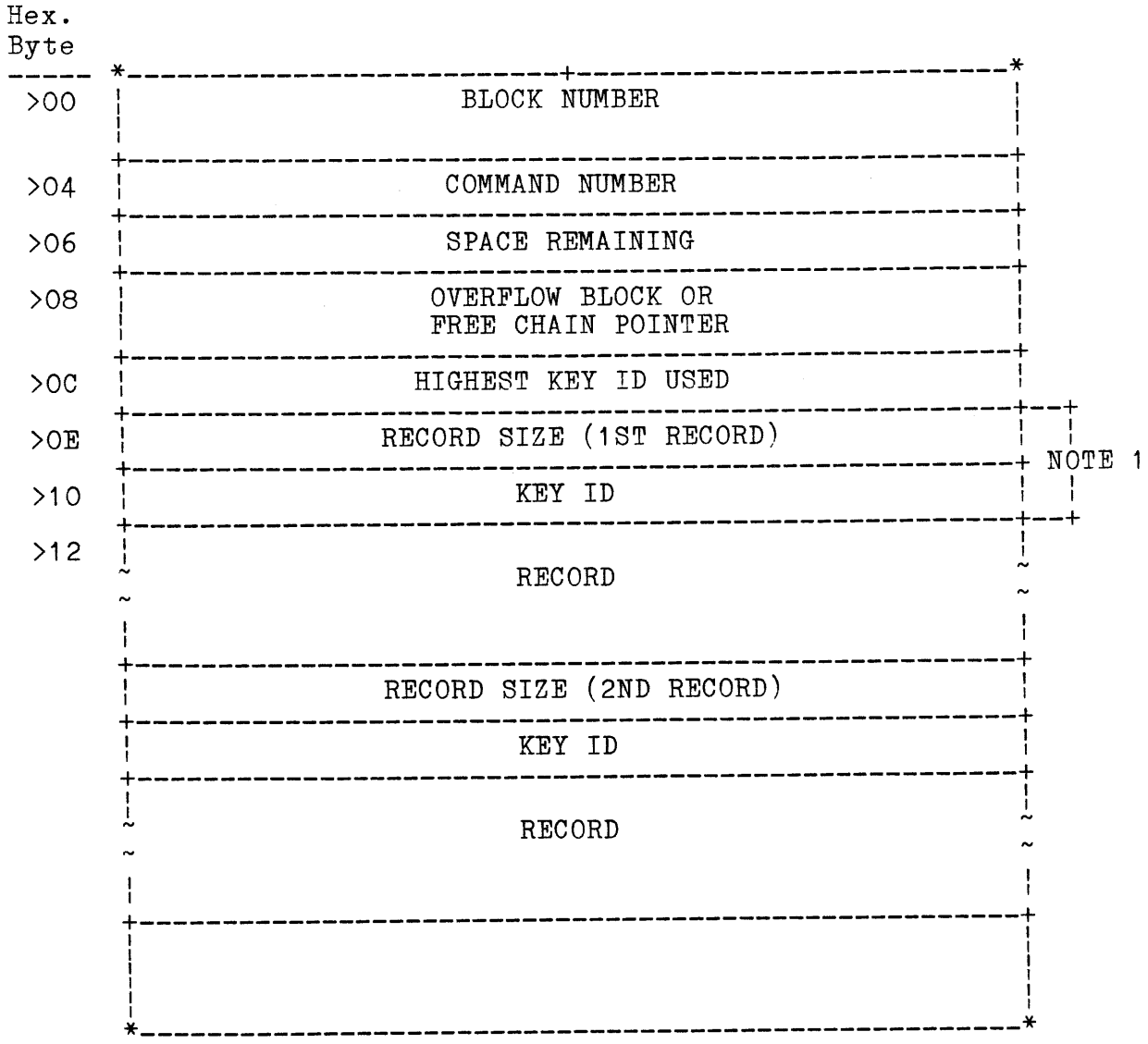
If the block is not a leaf entry, each pointer field points to a subtree that contains all key values less than or equal to the key value associated with the pointer. In fact, the highest key value

contained in the subtree is the key value associated with the pointer (as shown in the sample B-tree).

Further information on general B-tree structure is available in The Art of Computer Programming, Volume III by Donald Knuth.

4.3.3.2 Data Blocks.

All of the data records (logical records) of a key indexed file are contained in data blocks. A data block is a physical record of the file and contains a few words of overhead and several logical records, as shown in Figure 4-7. The word following the last logical record has a zero value.



NOTE 1 -- Four Overhead Bytes per Logical Record

Figure 4-7 Data Block

Hex. Byte	Description
>00	Physical record number of this block. This field is maintained so that, should a system crash occur while this block is being modified, the logged image can be restored to the correct file record.
>04	The opcode of the current command. This field is also maintained for logging purposes.

>06 The number of bytes remaining in the physical record.

>08 This block is used to link the block on the free block chain.

>0C The highest ID assigned to any logical record with the block.

>0E Size, in bytes, of the first logical record including this word.

>10 The ID assigned to the first logical record.

>12 First logical record.

Whenever a data record is to be inserted in a data block, it is assigned an ID that is unique within the block. The data record is then inserted in the first available place in the block.

4.3.4 SPECIAL RELATIVE RECORD FILES.

In addition to the three basic file types, three special uses of the relative record file warrant description: program files, directory files, and image files.

4.3.4.1 Program Files.

Program files are unblocked relative record files having a logical record size of one sector. The smallest sector size allowed is 256 bytes. Figure 4-8 shows the format of a program file. The sections of information describing the contents of the program file (see Figure 4-8 will not always start at the beginning of records or be in the same place for all program files. The following set of equations define the record number and the offset into the record of the beginning of the sections of information. In the equations, R designates a record and O designates the offset.

$$R1 = 1$$
$$O1 = 0$$

$$R2 = R1 + \frac{((\text{MAX \# TASKS} + 2)/2) * >10) + 01}{>100}$$

$$O2 = \text{remainder of } \frac{((\text{MAX \# TASKS} + 2)/2) * >10 + 01}{>100}$$

$$R3 = R2 + \frac{(\text{MAX \# TASKS} + 1) * >10 + 02}{>100}$$

$$O3 = \text{remainder of } \frac{(\text{MAX \# TASKS} + 1) * >10 + 02}{>100}$$

$$R4 = R3 + \frac{((\text{MAX \# PROCS} + 2)/2) * >10 + 03}{>100}$$

$$04 = \text{remainder of } \frac{((\text{MAX \# PROCS} + 2)/2) * >10 + 03}{>100}$$

$$R5 = R4 + \frac{(\text{MAX \# PROCS} + 1) * >10 + 04}{>100}$$

$$05 = \text{remainder of } \frac{(\text{MAX \# PROCS} + 1) * >10 + 04}{>100}$$

$$R6 = R5 + \frac{((\text{MAX \# OVLYS} + 2)/2) * >10 + 05}{>100}$$

$$06 = \text{remainder of } \frac{((\text{MAX \# OVLYS} + 2)/2) * >10 + 05}{>100}$$

$$R7 = R6 + \frac{(\text{MAX \# OVLYS} + 1) * >10 + 06}{>100}$$

$$07 = \text{remainder of } \frac{(\text{MAX \# OVLYS} + 1) * >10 + 06}{>100}$$

$$R8 = R7 + \frac{((\text{MAX \# HOLES} * 4) + 2) + 07}{>100}$$

$$08 = \text{remainder of } \frac{((\text{MAX \# HOLES} * 4) + 2) + 07}{>100}$$

If 08 is not equal to zero, then R8 = R8 + 1

R1,01: Record number and offset for names of tasks.
R2,02: Record number and offset for task directory entries.
R3,03: Record number and offset for names of procedures.
R4,04: Record number and offset for procedures directory entrie
R5,05: Record number and offset for names of overlays.
R6,06: Record number and offset for overlay directory entries.
R7,07: Record number and offset for unused space directory.
R8: Record number of first image record.

The first record (record number 0) of a program file contains six bit maps. These bit maps, in order of occurrence within record 0, are for memory resident tasks, memory resident procedures, all tasks, all procedures, all non-replicatable tasks, and all overlays.

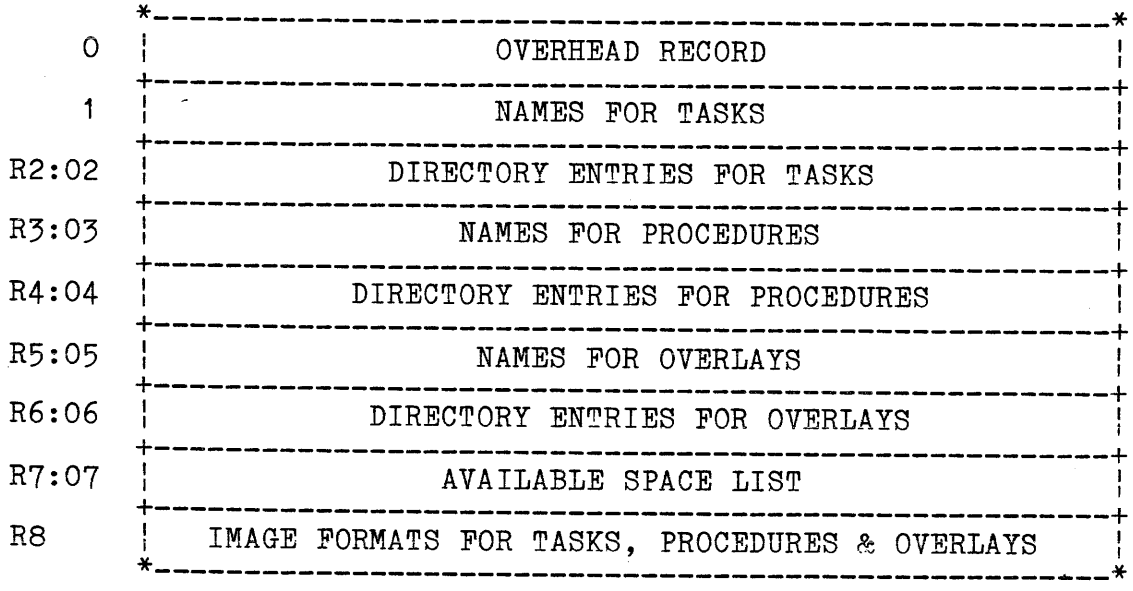


Figure 4-8 Program File Format

When record zero is initialized, all the bits in the bit map are zero except the first bit in the tasks, procedures, overlays, and non-replicatable tasks bit maps (the bit maps occupying bytes >54->D3). The first bit of these is a one, restricting user tasks from allocating ID zero.

Each bit map is 16 words by 16 bits per word, and thus is able to represent 256 IDs. A bit set to one indiates the ID corresponding to the bit position (i.e., 0 through 255) is assigned to a task, procedure, or overlay segment that is installed in the file. Figure 4-9 shows the format of record zero of a program file.

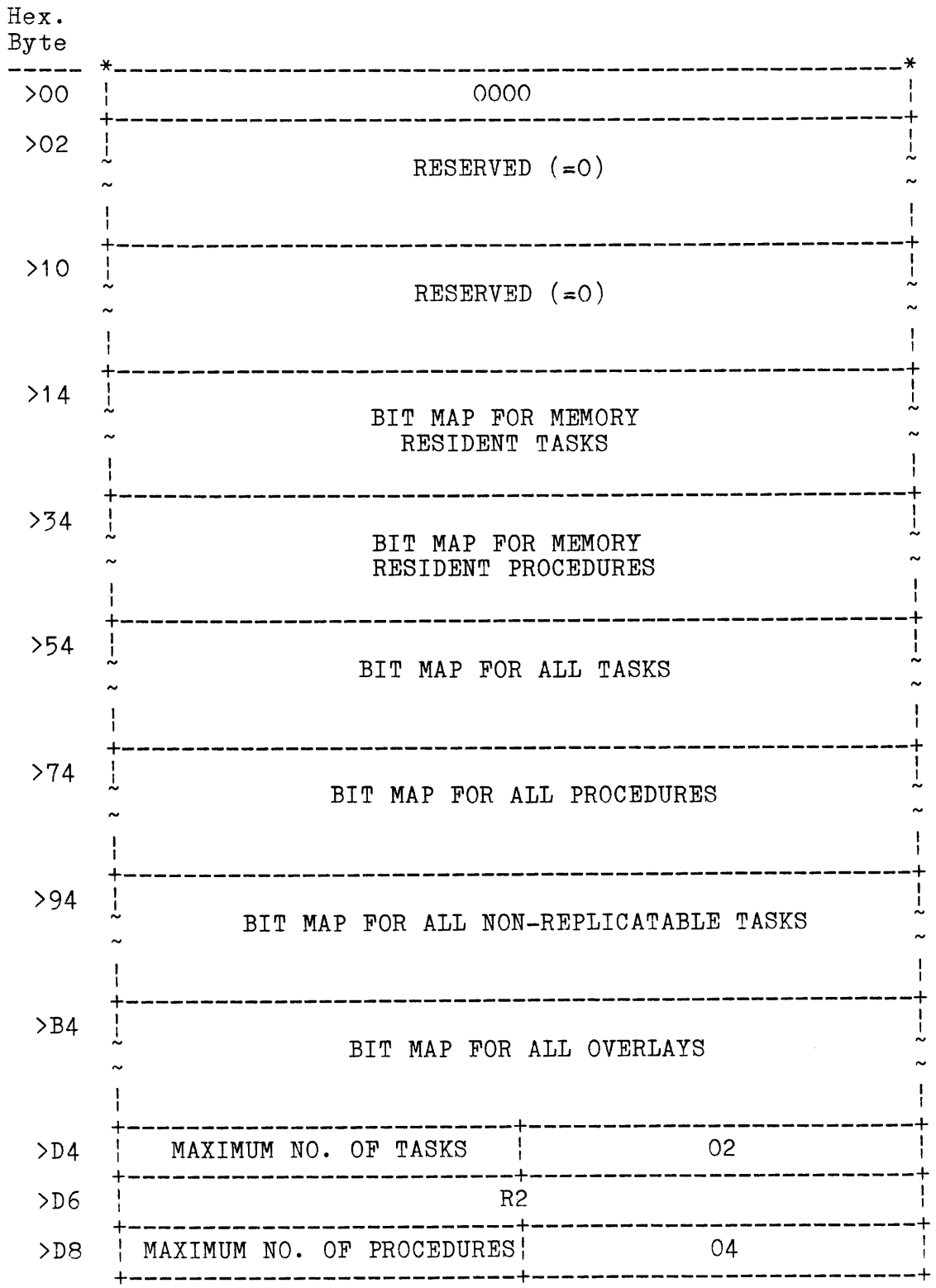


Figure 4-9 Program File Record Zero -- Part 1 of 2

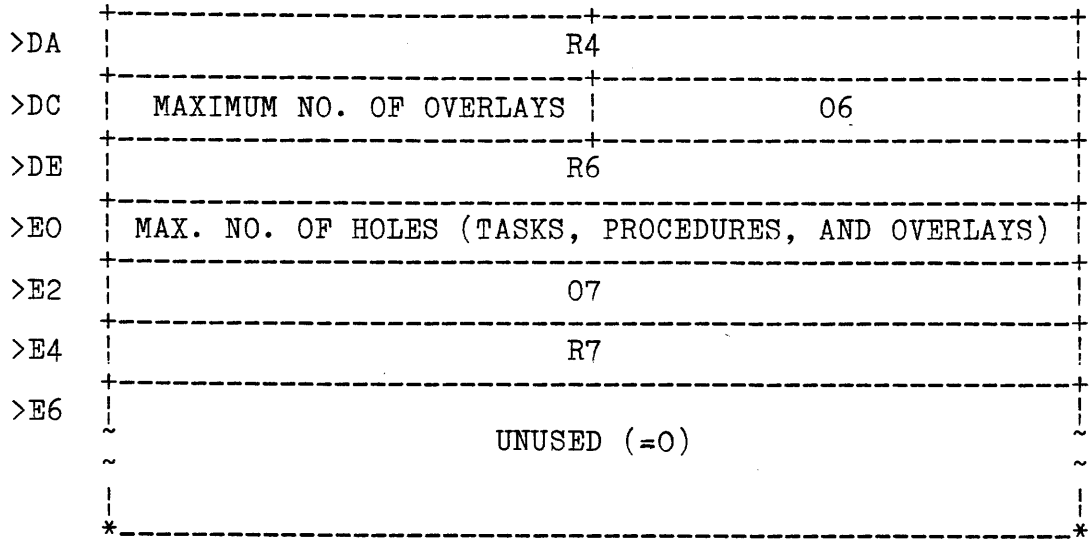
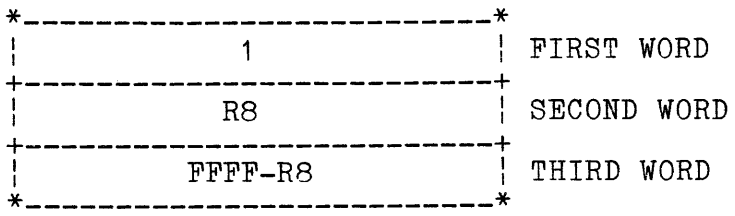


Figure 4-9 Program File Record Zero -- Part 2 of 2

At program file creation time, the maximum number of tasks, procedures, and overlays contained in bytes >D4, >D8 and >DC of record 0 are defined by the creator of the program file. The maximum number of holes, which equals the sum of the above three values, is used to calculate the number of bytes required in the overhead records for the available space list. This list is headed by a word that contains the number of entries in the list. The rest of the list consists of 2-word entries that describe the unallocated spaces (holes) in the image portion of the program file. Each entry contains the starting record number and the number of available records in each unused portion of the program file. These spaces appear when an image is deleted. This space is recorded to be used again if a new image is installed in the program file that is the same size or smaller than the one that was deleted. Adjacent images, when deleted, create only one hole. Figure 4-10 shows the format of the available space list.

The available space list uses the entire record, not 256 bytes of it as the other overhead records do. Therefore, if the list spans records, an entry is split across two records. (The first word of the entry is the last word of one record and the second word of the entry is the first word of the next record). The available space list is initialized at the same time record 0 is initialized. Its values are as follows:



R8 is the first record following the available space list.

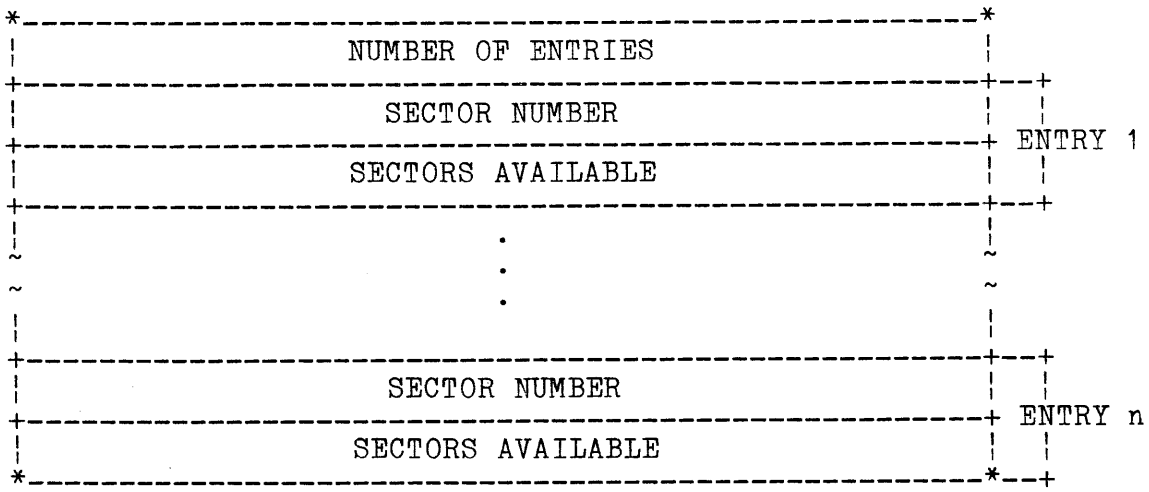


Figure 4-10 Program File Available Space List

The maximum number of records permitted in a program is FFFF. Thus, the maximum number of image records permitted in a program file is FFFF minus the number of overhead records. The actual image of a task, procedure, or overlay must start on a record boundary in the program file. If the segment has a relocation bit map, it begins at the first word following the program segment image.

The task, procedure, and overlay name blocks in the program file contain the names of all tasks, procedures, and overlays installed in the program file. A name is eight bytes long, blank-filled to the right. The names are placed in the position in the name block that corresponds to the ID assigned to that segment. For example, if task GENTX is assigned ID 1, the name GENTX is entered in bytes 8-15 (second position) of the task name block.

The task, procedure, and overlay directory blocks in the program file contain information about all segments installed in the program file, as well as pointers to the segment images. Each directory is 16 bytes long. The figures that follow show the formats of the program file directory entries, with the field description following their respective formats. Figure 4-11 shows the format of the task directory block.

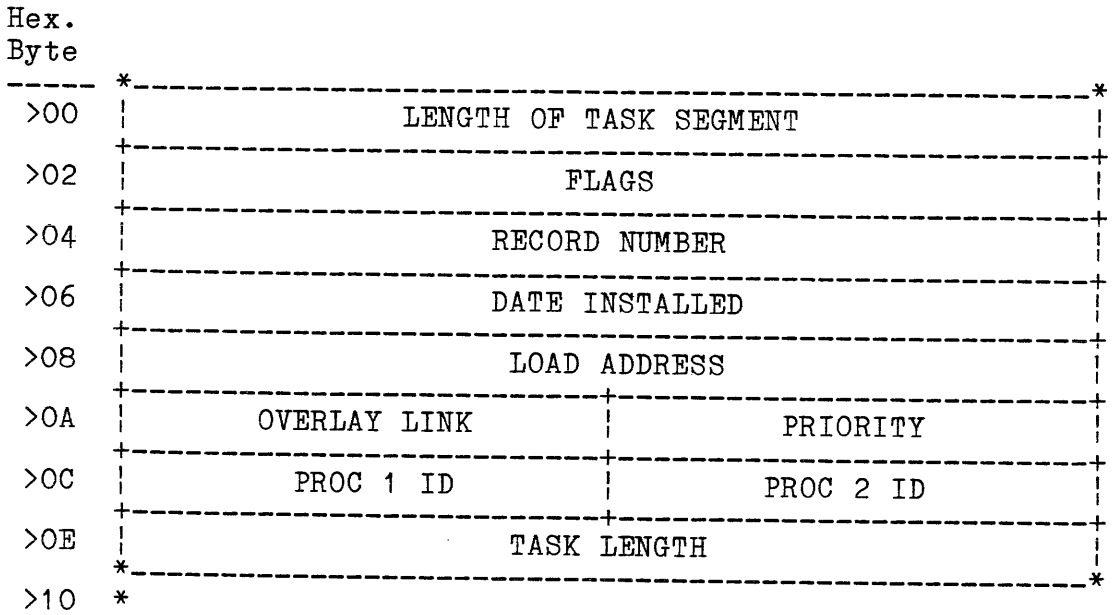


Figure 4-11 Task Directory Block

Hex. Byte	Description

>00 Length of task segment in bytes. Length of task root plus the length of the tasks longest overlay path.

>02 Flags, which mean the following when set:

Bit	Meaning When Set
0	Privileged
1	System
2	Memory resident
3	Delete protected
4	Replicatable
5	Procedure 1 is on the system program file
6	Procedure 2 is on the system program file
7	Directory entry in use
8	Overflow
9	Writable control store
10	Execute protected
11-15	Unused (set to zero)

>04 Record number. Logical record number of the start of the task image in the program file.

>06 Date installed. Date is in the format:

Bit	Meaning
0-6	Year (Displacement)
7-15	Julian date

>08 Load address. Relative starting address within a mapped task segment. Must be on a beet boundary.

>0A Overlay link. The ID of the most recently installed overlay associated with the task. Each overlay entry is in turn linked to the next entry so that tasks can be associated with their overlays when status or delete commands are executed. A value of 0 is used to terminate the list.

>0B Priority of the task.

>0C Procedure 1 ID.

>0D Procedure 2 ID.

>0E Length (in bytes) of the difference between the last defined location and the first defined location of a task. If a BSS is the last instruction in the task, its length is not included in this value.

>10 *

Figure 4-12 shows the format of the Procedure Directory Entry.

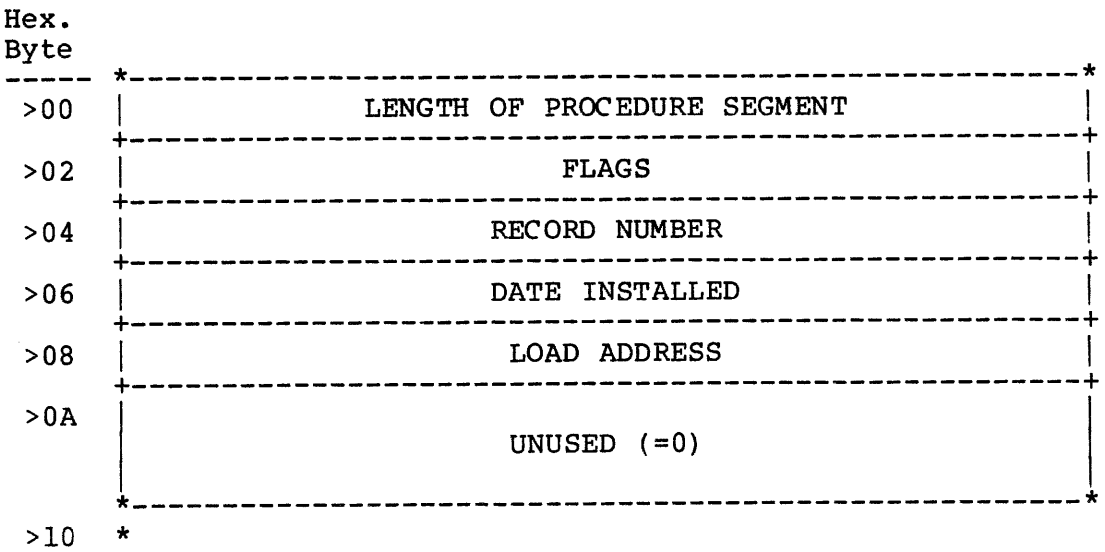


Figure 4-12 Procedure Directory Entry

Hex. Byte	Description
>00	Length of procedure segment in bytes.
>02	Flags, which mean the following when set:

Bit	Meaning When Set
0-1	Unused (set to zero)
2	Memory resident
3	Delete protected
4-6	Unused (set to zero)
7	Directory entry in use
8	Unused (set to zero)
9	Writable control store
10	Execute protected
11	Write protected
12-15	Unused (set to zero)

>04	Record number. Logical record number of the start of the procedure image in the program file.
>06	Date installed. Date is in the format:

Bit	Meaning
0-6	Year (Displacement)
7-15	Julian date

>08	Load address. Relative starting address within a mapped procedure segment. Must be on a beet boundary.
>0A	Unused.
>10	*

Figure 4-13 shows the format of the Overlay Directory Entry.

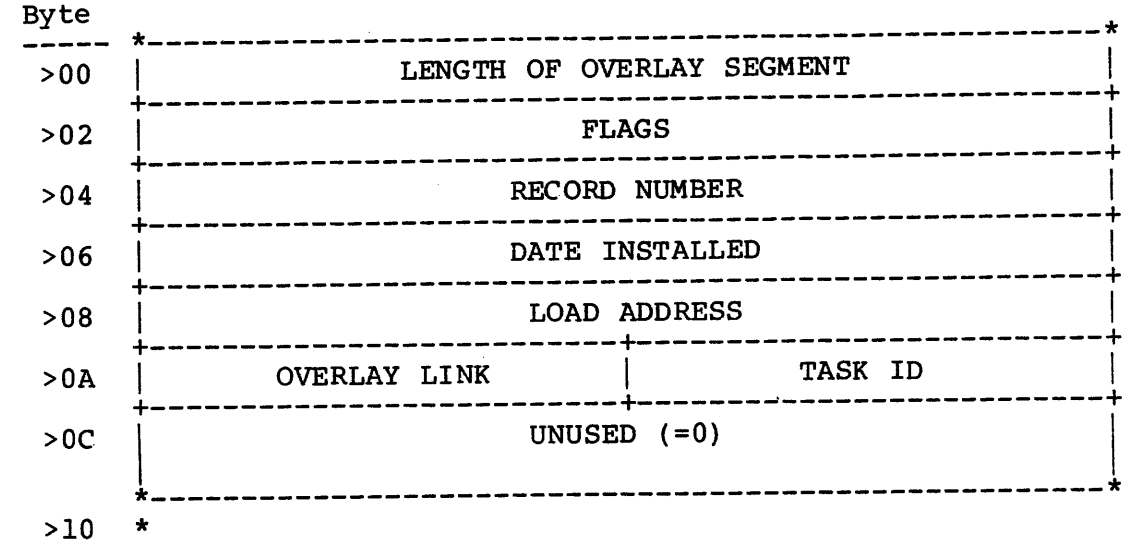


Figure 4-13 Overlay Directory Entry

Hex. Byte	Description
>00	Length of overlay segment in bytes.
>02	Flags, which mean the following when set:

Bit	Meaning When Set
0	Relocation bit map is present
1-2	Unused (set to zero)
3	Delete protected
4-6	Unused (set to zero)
7	Directory entry in use
8-15	Unused (set to zero)

>04 Record number. Logical record number of the starting address of the overlay image in the program file.

>06 Date installed. Date is in the format:

Bit	Meaning
0-6	Year (Displacement)
7-15	Julian date

>08 Load address. Relative starting address within a mapped overlay segment. Must be on a beet boundary.

>0A Overlay link to the next overlay.

>0B Task ID of associated task.

>0C-0F Unused (set to zero).

>10 *

4.3.4.2 Directory Files.

Directory files are unblocked relative record files and always have a record length of one sector. Record 0 of the directory file contains an overhead record. The remaining records in the file may contain one of the following types of data blocks:

* File Descriptor Record (FDR) -- every file cataloged in the directory is represented by an FDR, which describes the file and its location on the disk.

* Alias Descriptor Record (ADR) -- every alias of a file cataloged in the directory is represented by an ADR, which gives the location of the file and points to the FDR of the actual file.

* Key Descriptor Record -- each key indexed file cataloged in the directory is represented by an FDR, which in turn points to a key descriptor record. The key descriptor record describes all of the keys (1-14) that are defined for the file. Note that the use of the key descriptor record implies that each key indexed file cataloged in a directory uses two directory entries.

Figure 4-14 shows the general structure of a directory file. Entries are made in the directory file by hashing the name of the file being entered. The hash algorithm results in a record number from one through n, where n is the last record in the directory file.

Figure 4-15 shows the hash algorithm. If the directory file record is unused, an FDR for the file being inserted is placed in that record. If the record is already used, a free record is found by a linear search from the hashed record.

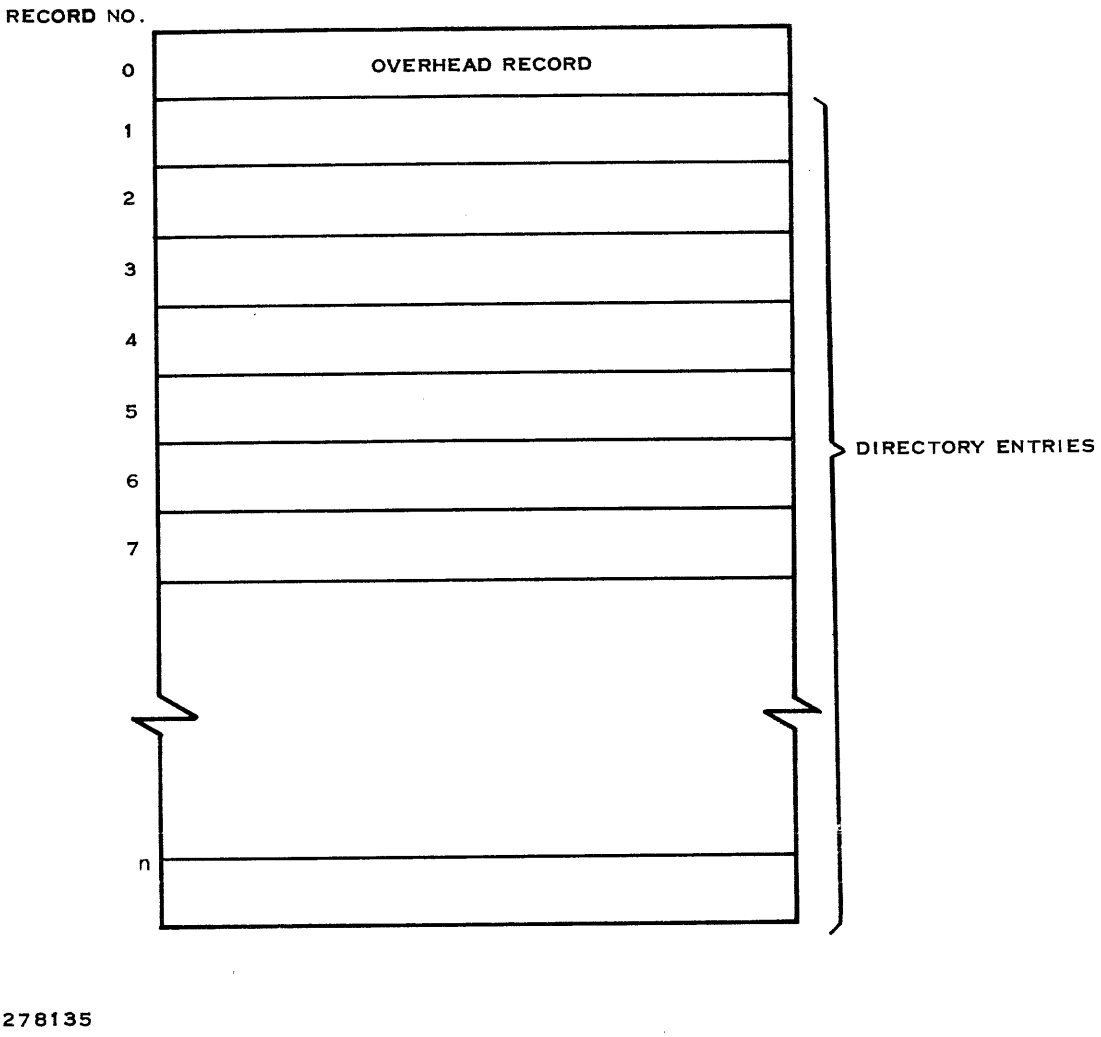


Figure 4-14 Directory File Structure

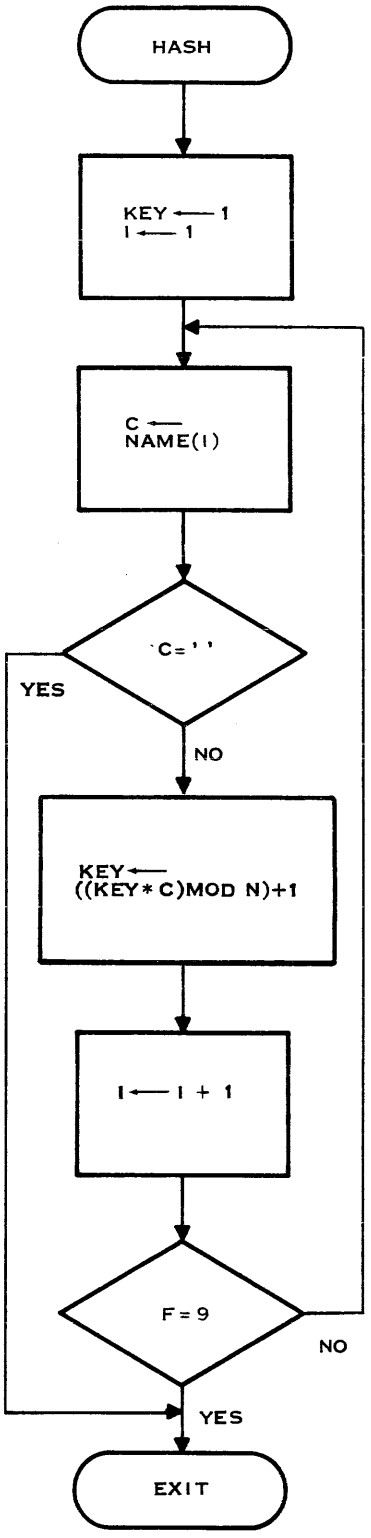


Figure 4-15 Computing Hash Key

If the file being inserted is a key indexed file, another directory record must be found to contain the key descriptor record. This record is found by searching linearly from the file descriptor record for the file. The key descriptor record is inserted in the first available directory record following the file descriptor record.

The different types of directory records are described in the following paragraphs.

The directory overhead record (record 0 of all directories) contains:

* The maximum number of records (entries) in the directory.
* The number of currently defined files.
* The number of available records (entries).
* The filename of the directory.
* The level number of the directory in the disk hierarchy (VCATALOG) is level 0)
* The filename of the parent directory.
* The default physical record length.

Figure 4-16 shows the format of a directory overhead record.

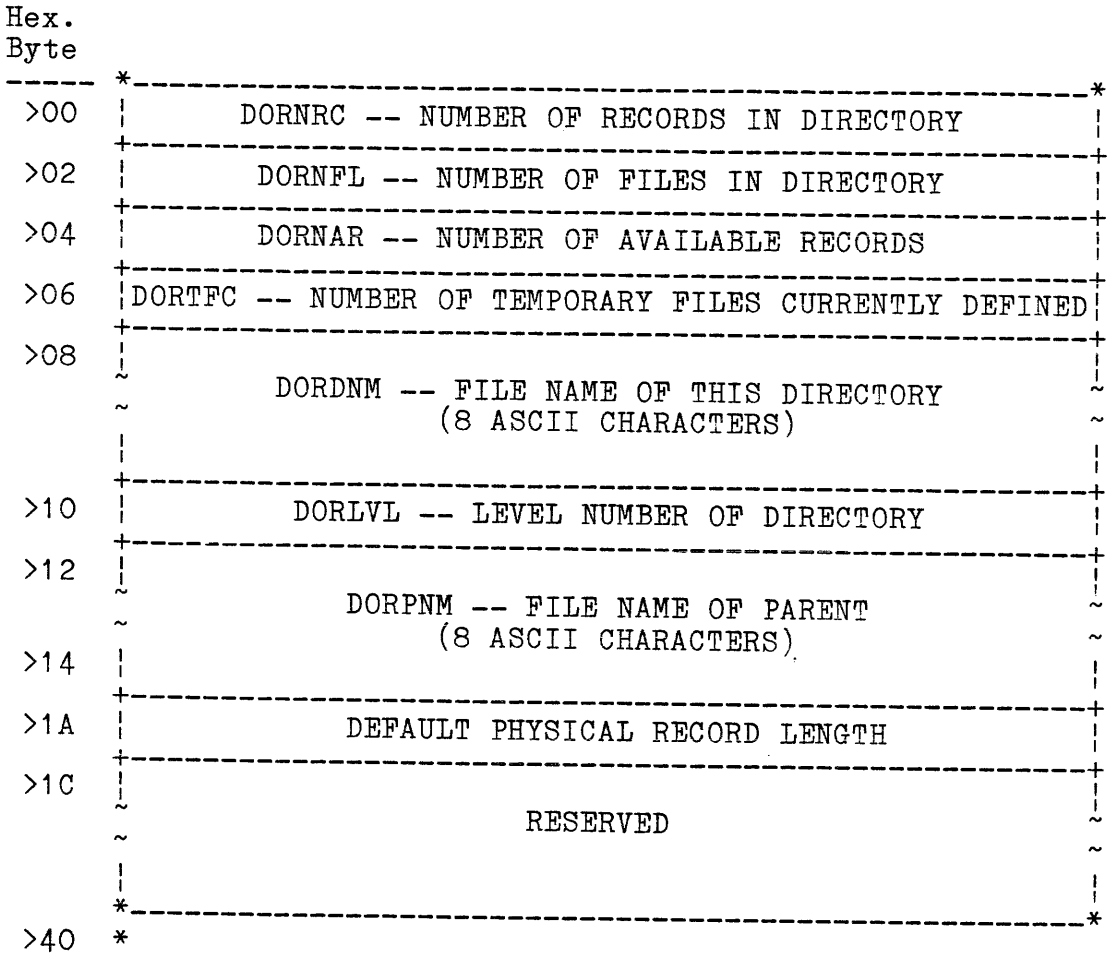


Figure 4-16 Directory Overhead Record Format

Each file cataloged under the directory is represented by a file descriptor record. Figure 4-17 shows an FDR.

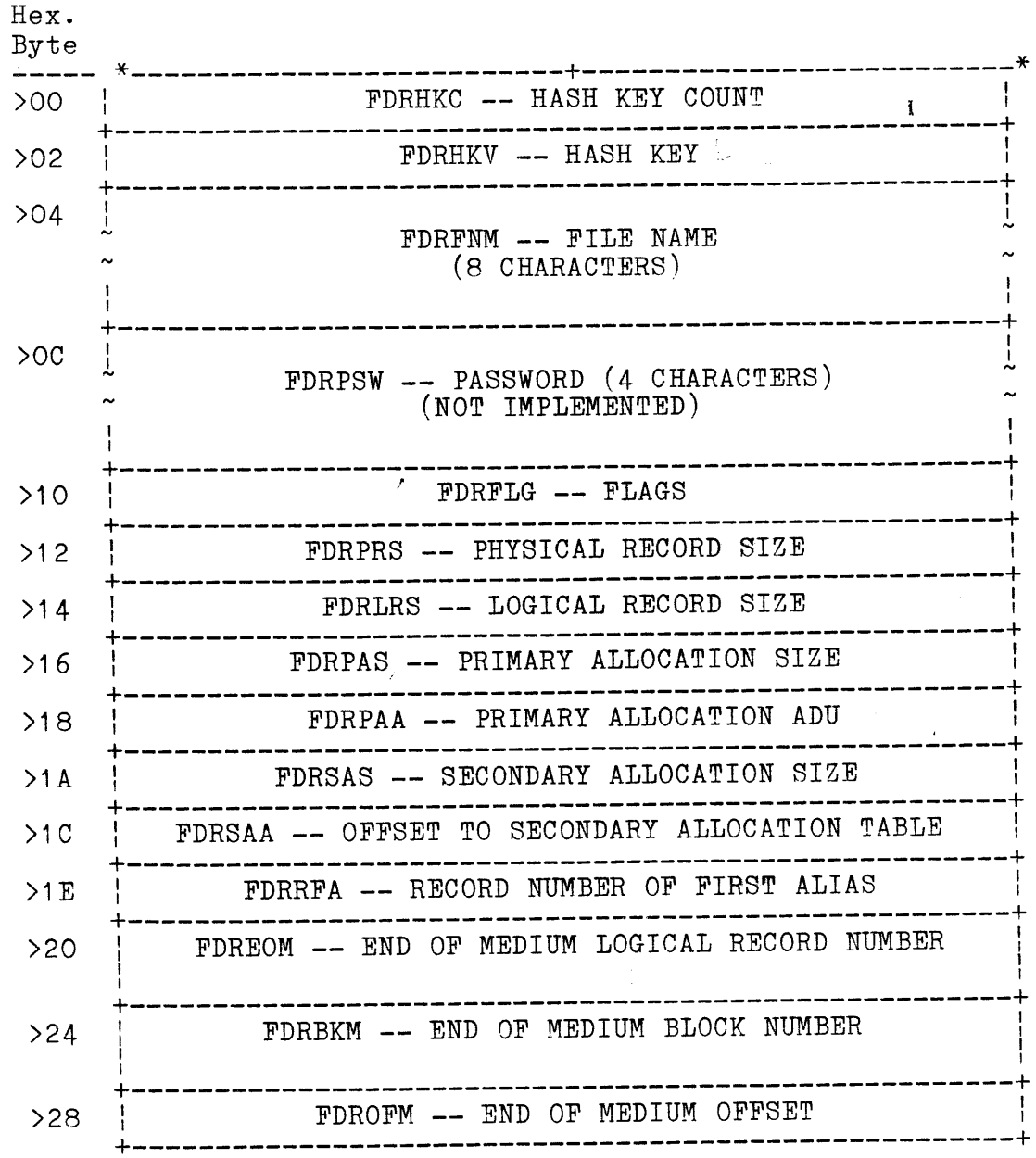
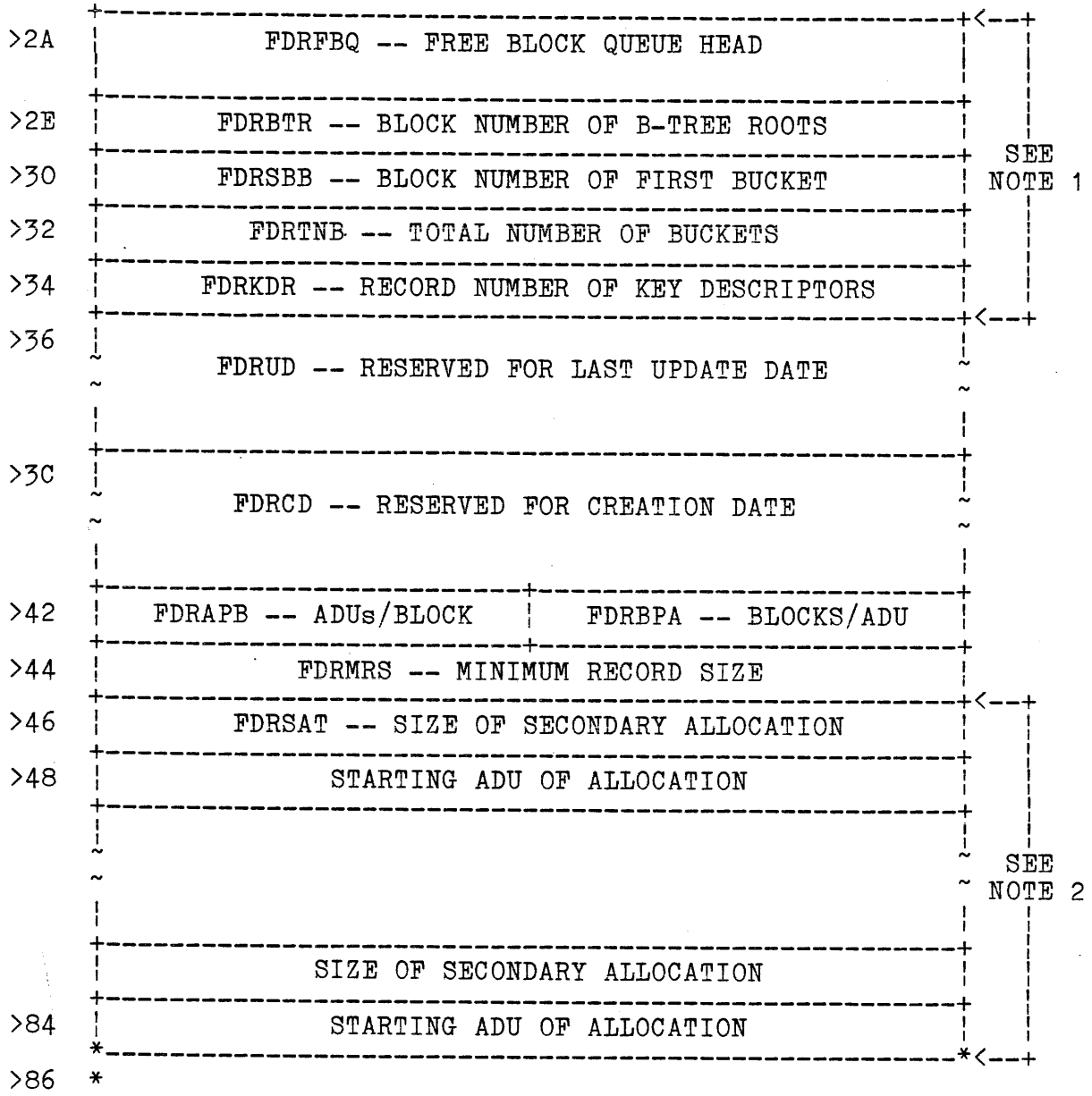


Figure 4-17 File Descriptor Record -- Part 1 of 2



NOTE 1 -- Used only for key indexed files.
NOTE 2 -- Secondary allocation table (up to 16 allocations).

Figure 4-17 File Descriptor Record -- Part 2 of 2

Hex. Byte	Field Name	Description
>00	FDRHKC	Hash Key Count. The number of file descriptor records (which may or may not include this one) that are present in the directory that hashed to this record number.
>02	FDRHKV	Hash Key. The result of the hash algorithm for the file name actually covered in this record. The value might not be this record number since the data may have arrived here via the linear search used when the hashed address is occupied.
>04	FDRFNM	File Name. Eight characters.
>0C	FDRPSW	Password. A future feature.
>10	FDRFLG	Flags as follows:

Bit	Meaning
0-1	File usage flags: 00 No special usage 01 Directory 10 Program File 11 Image File
2-3	Data Format: 00 Binary 01 Blank suppressed 10 Reserved for ASCII & print form control 11 Reserved
4	Allocation type: 0 Bounded 1 Unbounded
5-6	File type: 00 Reserved (for device) 01 Sequential 10 Relative record 11 Key indexed
7	Write protection flag: 0 Not write protected 1 Write protected
8	Delete protection flag: 0 Not delete protected 1 Delete protected

9 Temporary file flag:
0 Permanent flag
1 Temporary flag

10 Blocked file flag:
0 Blocked
1 Unblocked

11 Alias flag:
0 Not an alias
1 An alias file name

12 Force write flag:
0 Write buffers when memory is required
1 Write buffers when updated

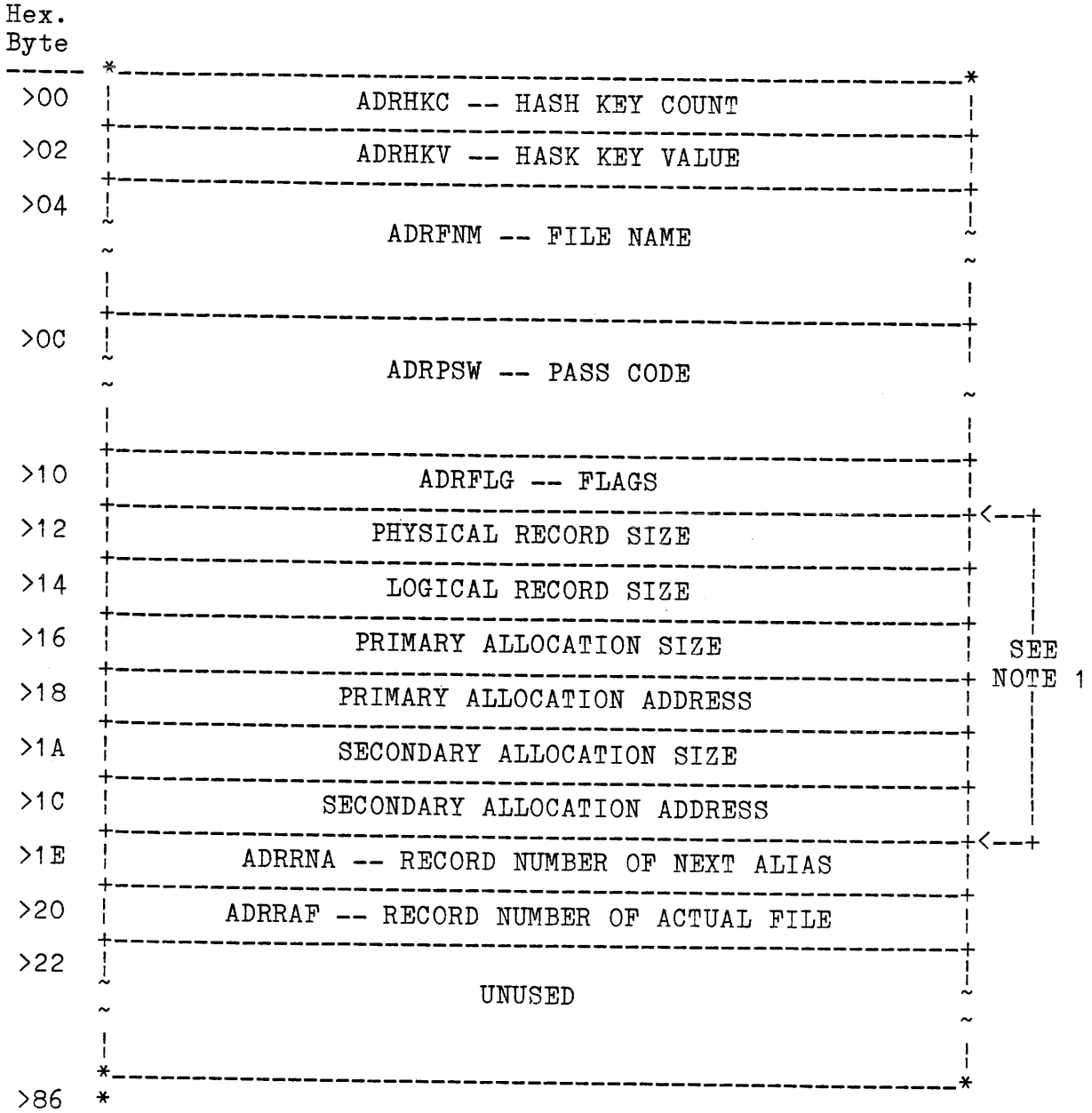
13 Reserved for FCB changed flag (see 6.4)

14-15 Reserved

>12	FDRPRS	Physical record size in bytes. Must be an even number.
>14	FDRLRS	Logical record size in bytes. Must be an even number if the file is unblocked.
>16	FDRPAS	Primary allocation size in ADU.
>18	FDRPAA	Primary allocation starting ADU number (starting disk address).
>1A	FDRSAS	Secondary allocation size in ADU.
>1C	FDRSAA	Offset into this FDR of the secondary allocation table, if any. No secondary allocation table is denoted by 0. Secondary allocations are present only for unbounded files.
>1E	FDRRFA	Record number with the directory of first alias name. Files may be known by alias names. The alias names are noted in the directory in alias descriptor records. These alias descriptor records are chained to the actual FDR and each contains a pointer back to the actual FDR.
>20	FDREOM	The logical record number of the end of medium. The end of medium is the end of the last space allocated to the file.
>24	FDRBKM	The logical block number of the end of medium. A logical block is the same as a physical block.
>28	FDROFM	The offset into the end of medium block of the logical record following the end of medium record.

>2A	FDRFBQ	Block number of the first block in a queue of key indexed file free (unused) blocks. Each block points to the next block in the queue (a block is a physical record of the file). Only used for key indexed files.
>2E	FDRBTR	The block number for the B-tree root block of the primary key. The block following this is the KIF root block for key 2, etc. This field is also the total number of blocks that can be used for logging.
>30	FDRSBB	The block number for the first KIF bucket.
>32	FDRTNB	The total number of buckets in the KIF file.
>34	FDRKDR	Record number of the directory file record containing descriptions of the KIF keys.
>36	FDRUD	Date of the last update to this file. The date is made up of three words. Word 1 contains the binary value of the year. Word 2 contains a value that is two times the number of days (counting from the beginning of the calendar year); the least significant bit is the most significant bit of word 3 of the date. Word 3 contains the number of seconds from the beginning of a day.
>3C	FDRCD	Creation date of the file. The date is made up of three words. Word 1 contains the binary value of the year. Word 2 contains a value that is two times the number of days (counting from the beginning of the calendar year); the least significant bit is the most significant bit of word 3 of the date. Word 3 contains the number of seconds from the beginning of a day.
>42	FDRAPB	The number of ADUs per physical record.
>43	FDRBPA	The number of physical records per ADU.
>44	FDRMRS	The minimum size that a key indexed file logical record can be and still contain all of the keys defined.
>46	FDRSAT	The secondary allocation table, which contains 16 2-word entries. The first word of an entry contains the size, in ADUs, of the secondary allocation. The second word contains the starting ADU of the allocation. The table allows up to 16 files, and is only used if the file was created expandable (unbounded). The entry fields are filled in by file management as the file is expanded.
>86	*	

Files can be given other names, each name being a separate alias. Each alias is hashed to find an entry in the directory just like a file name, and an alias descriptor record (ADR) inserted in that entry. The ADR points to the real file. It also points to the next alias for the file. Figure 4-18 shows the format of an ADR.

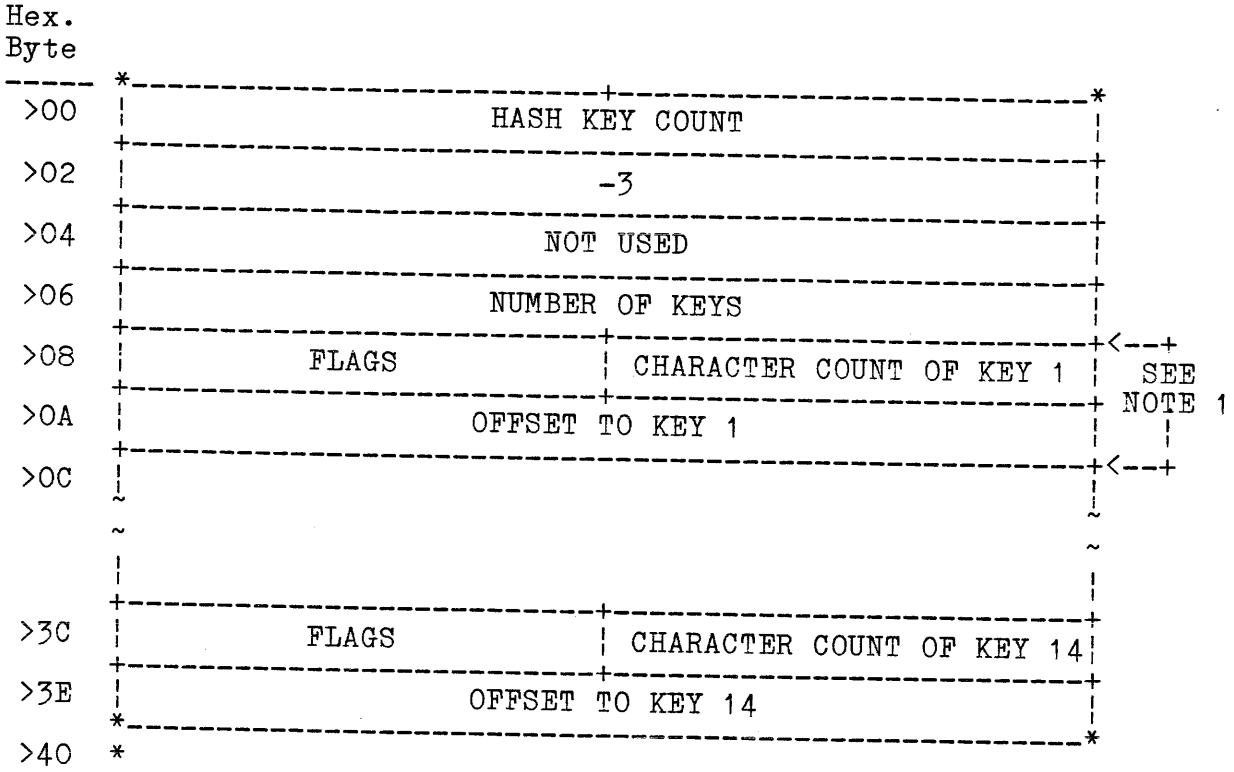


NOTE 1 -- Used to maintain ADR for compatibility.

Figure 4-18 Alias Descriptor Record

Hex. Byte	Field Name	Description
>00	ADRHKC	Hash Key Count. The hash key count is the same as in the file descriptor record.
>02	ADRHKV	Hash Key Value. The hash key value is the same as in the file descriptor record.
>04	ADRFNM	File Name. The file name given in this item is an alias name for the file. In other words, it is a secondary name by which a previously defined file will also be known. The primary name for a file is supplied in the file descriptor record; secondary names are documented in alias descriptor records.
>0C	ADRPSW	Passcode. Space is provided for future implementation of password codes.
>10	ADRFLG	Flags. The flag values are the same as provided in the file descriptor record. Note that the flag for alias is set to a one in this particular record.
>12->1D		These fields are not used.
>1E	ADRRNA	Record Number of Next Alias. This is a pointer chaining forward to another alias descriptor record for the same file, if any exists. A value of zero is provided to indicate the end of the chain; i.e., no more alias descriptor records exist for the file.
>20	ADRRAF	Record Number of Actual File. This is a pointer to the directory file record containing the file descriptor record for this particular file.
>22->85		Unused. These bytes may contain non-zero values, but they are not used.
>86	*	

A key descriptor record is used only for key indexed files. It describes the keys (up to fourteen) used to access records in the file. When a key indexed file is created and its keys are defined, a file descriptor record is hashed into the directory. File utility then performs a linear search for an unused directory record, starting from the file descriptor record. The key descriptor record is placed in the first available directory record. Figure 4-19 shows the format of a key descriptor record.



NOTE 1 -- For the primary key; repeat for each secondary key.

Figure 4-19 Key Descriptor Record

Hex. Byte	Description
>00	Hash Key Count. This is the same as described for the file descriptor record.
>02	Hash Key = -3. This field is similar to that provided with the file descriptor record. The value of -3 is given to indicate that this record is a key descriptor record and, therefore, is unavailable for use as a file descriptor record.
>04	Not used.
>06	The number of unique keys defined for this key indexed file. There are a maximum of 14 keys available for any key indexed file. There must be at least one key, the primary key. Keys 2-14, if any, are secondary keys.

>08 Flags, as follows:

Bit	Meaning When Set
0-3	Must be zero
4	File was created by a system using the sequential placement scheme (primary key only)
5	Of a secondary key is set to 1 if the key value need not always be present
6	Sequential commands are desired on this key (e.g., Read Next)
7	Duplicates are allowed on this key

<09 The key length, in bytes (characters), of the primary key.

<0A The starting byte number for the position of the key within the key indexed file data record. The prior three items (flags, character count of key, and key offset) are repeated for each secondary key.

<40 *

Figure 4-20 shows a dump of the directory file .JB.DIR. The directory contains a sequential file (.JB.DIR.SEQ), an image file (.JB.DIR.IMAG), a program file (.JB.DIR.PROG), and a key indexed file (.JF.DIR.KEY). The directory also contains an alias for the key indexed file. The directory was created to have 11 entries, in addition to record 0 which is the directory overhead record.

4.3.4.3 Image Files.

Image files are nonexpandable, unblocked relative record files that contain memory images of programs. They are not organized in any format; that is, each sector of the image file, starting with the first sector, is completely filled with data. There are no overhead records or words. Image files are designed so that a program image can be read into memory in a single disk access.

```
FILE:.JB.DIR  RECORD:000000
0000  000B 0004 0005 0000 4449 5220 2020 2020    .. .. .. .. DI R
0010  0002 4A42 2020 2020 2020 0000 0000 0000    .. JB          .. .. ..
  SAME
0084  0000                                       ..
FILE:.JB.DIR  RECORD:000001
0086       0002 0001 5345 5120 2020 2020 0000       .. .. SE Q         ..
0094  0000 1A00 0120 0028 0001 016E 0001 0000    .. .. .  .( .. .. .. ..
00A4  0000 0000 0000 0000 0000 0000 0000 0000    .. .. .. .. .. .. .. ..
00B4  0000 0000 0000 0000 07B9 01D4 7E49 07B9    .. .. .. .. .. .. .I ..
00C4  01D4 7E49 0103 0000 0000 0000 0000 0000    .. .I .. .. .. .. .. ..
  SAME
010A  0000                                       ..
FILE:.JB.DIR  RECORD:000002
010C       0000 0001 494D 4147 2020 2020 0000       .. .. IM AG        ..
011A  0000 C420 0120 0120 0004 1065 0001 0000    .. .  .  .  .. .. .. ..
012A  0000 0000 0000 0000 0000 0000 0000 0000    .. .. .. .. .. .. .. ..
013A  0000 0000 0000 0000 07B9 01D4 7E8B 07B9    .. .. .. .. .. .. .. ..
014A  01D4 7E8B 0103 0000 0000 0000 0000 0000    .. .. .. .. .. .. .. ..
  SAME
0190  0000                                       ..
FILE:.JB.DIR  RECORD:000003
0192       0000 0000 0000 0000 0000 0000 0000       .. .. .. .. .. .. ..
  SAME
0216  0000                                       ..
FILE:.JB.DIR  RECORD:000004
0218       0000 0000 0000 0000 0000 0000 0000       .. .. .. .. .. .. ..
  SAME
029C  0000                                       ..
FILE:.JB.DIR  RECORD:000005
029E       0000 0000 0000 0000 0000 0000 0000       .. .. .. .. .. .. ..
  SAME
0322  0000                                       ..
FILE:.JB.DIR  RECORD:000006
0324       0000 0000 0000 0000 0000 0000 0000       .. .. .. .. .. .. ..
  SAME
03A8  0000                                       ..
FILE:.JB.DIR  RECORD:000007
03AA       0001 0007 5052 4F47 2020 2020 0000       .. .. PR OG        ..
03B8  0000 8C20 0120 0120 001D 24A9 0001 0000    .. .  .  .  .. $. .. ..
03C8  0000 0000 004A 0000 004A 0000 0000 0000    .. .. .J .. .J .. .. ..
03D8  0000 0000 0000 0000 07B9 01D4 7E78 07B9    .. .. .. .. .. .. .. ..
03E8  01D4 7E77 0103 0000 0000 0000 0000 0000    .. .. .. .. .. .. .. ..
  SAME
042E  0000                                       ..
FILE:.JB.DIR  RECORD:000008
0430       0000 0000 0000 0000 0000 0000 0000       .. .. .. .. .. .. ..
  SAME
04B4  0000                                       ..
FILE:.JB.DIR  RECORD:000009
04B6       0001 0009 4B45 5946 494C 4520 0000       .. .. KE YF IL E   ..
04C4  0000 1E18 0000 0000 0000 0000 0000 0000    .. .. .. .. .. .. .. ..
04D4  0000 000A 0000 0000 0000 0000 0000 0000    .. .. .. .. .. .. .. ..
  SAME
053A  0000                                       ..
FILE:.JB.DIR  RECORD:00000A
053C       0001 000A 4B45 5920 2020 2020 0000       .. .. KE Y         ..
054A  0000 1E08 0120 0050 001B 248E 0002 0000    .. .. .  .P .. $. .. ..
055A  0009 0000 0000 0000 0000 0006 0000 004E    .. .. .. .. .. .. .. .N
056A  0027 0029 0025 000B 07B9 01D4 7E1D 07B9    .' .) .% .. .. .. .. ..
057A  01D4 7E1C 0103 0009 0000 0000 0000 0000    .. .. .. .. .. .. .. ..
  SAME
05C0  0000                                       ..
FILE:.JB.DIR  RECORD:00000B
05C2       0000 FFFD 0064 0002 0104 0000 0005       .. .. .. .. .. .. ..
05D0  0004 0000 0000 0000 0000 0000 0000 0000    .. .. .. .. .. .. .. ..
  SAME
0646  0000                                       ..

2278136
```

Figure 4-20 Directory File Dump

SECTION 5

SYSTEM FILES

5.1 GENERAL

This section describes the structure and purpose of various files used by DX10. These special files are:

* System program file
* System overlay file
* Crash file
* Roll file

The files used by the System Command Interpreter are discussed in the section on SCI.

5.2 SYSTEM PROGRAM FILE

The system program file has the same structure as a general program file, as described in the section on disk data structures. It is called the system program file because it contains all of the disk resident system tasks and their overlays, as well as many utility programs.

Queue-serving system tasks on the system program file include:

* SLMFOT - System log message formatting and output task
* TM$SBD - Scheduled Bid Task SVC processor
* TMDGN - Termination task
* SVCKIL - Kill TASK SVC processor
* PF$LIN - Install Task, Procedure, and Overlay SVC processor
* PF$LOE - Delete Task, Procedure, and Overlay SVC processor
* FUTIL - File Utility SVC processor
* PF$LMN - Map Name to ID SVC processor

* PF$LAS - Assign Space on Program File SVC processor
* INSTAL - Install, Unload, Initialize Volume SVC processor

5.3 SYSTEM OVERLAY FILE

The system overlay file is an unblocked relative record file used to contain system overlays. Each overlay is placed in the record that corresponds to the overlay number, and is in memory image format. Each record is 800 bytes.

When a system overlay is requested, the record that corresponds to the required overlay number is read directly into one of the system overlay areas by the system overlay loader, thus incurring very little overhead.

Table 5-1 shows what overlays are contained on the system overlay file.

Table 5-1 System Overlay Numbers

Overlay Number/ File Record	Overlay
1	Key indexed file B-Tree split routine
2	Key indexed file record insert routine
3	More key indexed file record insert
4	Key index file open and close routine
5	Sequential placement B-Tree Split routine
6	Sequential placement record insert routine
7	More sequential placement record insert
8	KIF record delete overlay
9	KIF delete B-Tree entry
A	Sequential placement record delete overlay
B	Sequential placement record delete overlay
C	Sequential placement delete B-Tree entry
D-F	Disk manager overlays
10-13	File manager overlays
14	Disk manager overlay
20	Bidder task error recovery routine

5.4 CRASH FILE

The system crash file, .S$CRASH, is an image file created large enough to contain a dump of all memory. When a system crash occurs, the routine SCRASH displays the crash code on the front panel lights and idles the CPU. If a dump is taken, SCRASH writes all of memory to the file .S$CRASH. This file may later be used as input by the crash analysis program, ANALYZ.

5.5 ROLL FILE

The roll file is an expandable image file that is used to contain rolled-out task and procedure images. It is created by the CSF (Create System Files) command.

There is no single directory of roll file entries. Instead, a linked list of the TSBs and PSBs of rolled tasks and procedures is maintained. Each TSB and PSB contains the starting record number and number of roll file records used by that segment. When the roll allocation routine, TMRDAL, searches for a block of free roll file space, it runs down the list until it finds a hole between segments, or space between the last rolled segment and the end of the file, large enough to fill the request for roll space from the task loader. The roll file may be extended, if necessary.

SECTION 6

DATA STRUCTURES

6.1 GENERAL

Memory resident data structures within DX10 consist of many tables, queues, and buffers. Most of the tables and buffers are dynamically allocated from the system table area. Most of the system queues are anchored in the DX10 data base modules, DDATA and DXDAT2. The following paragraphs describe the important data structures used by DX10.

6.2 QUEUES

The general structure of a DX10 queue is described in Section 1. The queues are singly linked, first-in, first-out lists of data structures to be processed. A queue is established by a queue anchor, which is usually a 10-byte block having a format as shown in Figure 6-1.

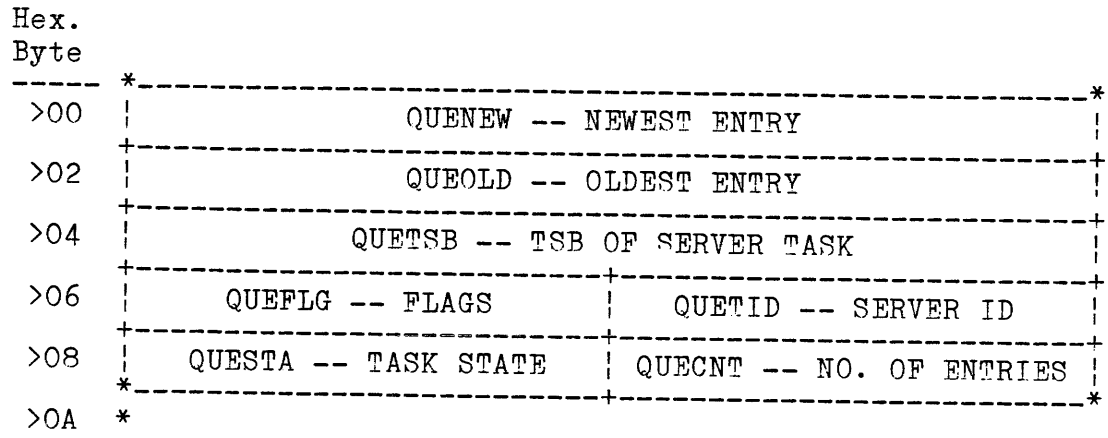


Figure 6-1 Queue Anchor

Hex. Byte	Field Name	Description
>00	QUENEW	The address of the newest (last) entry on the queue. Note that since this is only a one-word address, it is implied that the queued structures are mapped in with the queue anchor.
>02	QUEOLD	The address of the oldest (first) entry on the queue.
>04	QUETSB	The address of the task status block of the queue serving task (see paragraph 6.7 on TSBs). This field is zero if no queue server exists for the queue, or if the queue server is not loaded into memory.
>06	QUEFLG	Flags, which when set mean: Bit — Meaning 0 — Priority ordered queue (TSBs with high priorities are at the front of the queue). 1 — TSB queue (entries are TSBs). 2-7 — Reserved.
>07	QUETID	The installed ID of the queue serving task (zero if no server). Queue servers must be installed on the system program file.
>08	QUESTA	The task state that is to be assigned to all TSBs that are placed in this queue (=>FF if none).
>09	QUECNT	The number of entries currently in the queue.
>0A	*	

Most queue anchors are located in the module named DXDAT2. See Section 7 for further information about DXDAT2.

6.3 PHYSICAL DEVICE TABLE

A Physical Device Table (PDT) is a data structure that represents a physical device to the operating system. In addition to containing information describing the device, the PDT is used as a workspace by the device service routine. Some of the uses of the PDT are discussed in Volume V of the DX10 reference manuals. A physical device table has the format shown in Figure 6-2.

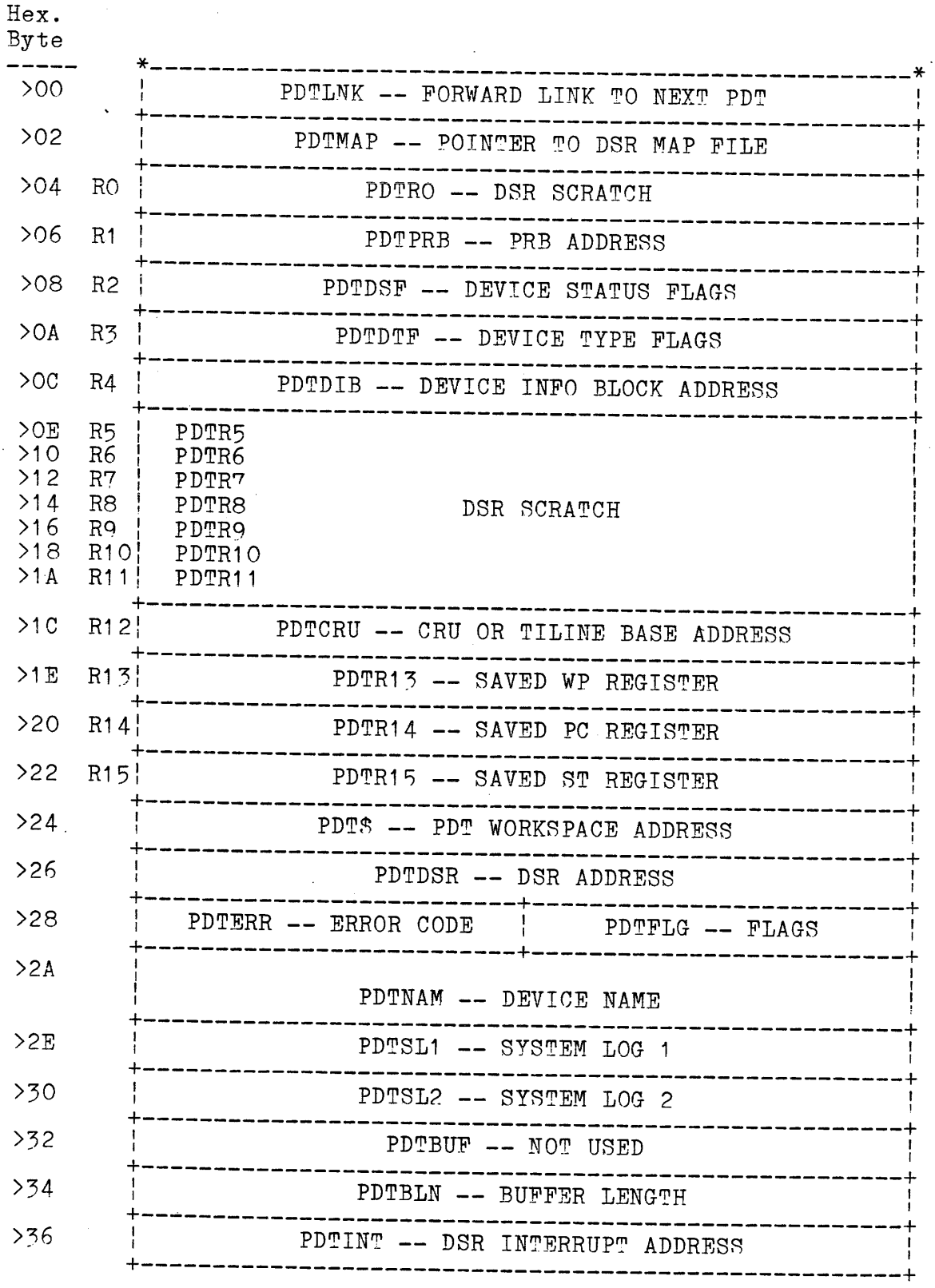


Figure 6-2 Physical Device Table (Part 1 of 2)

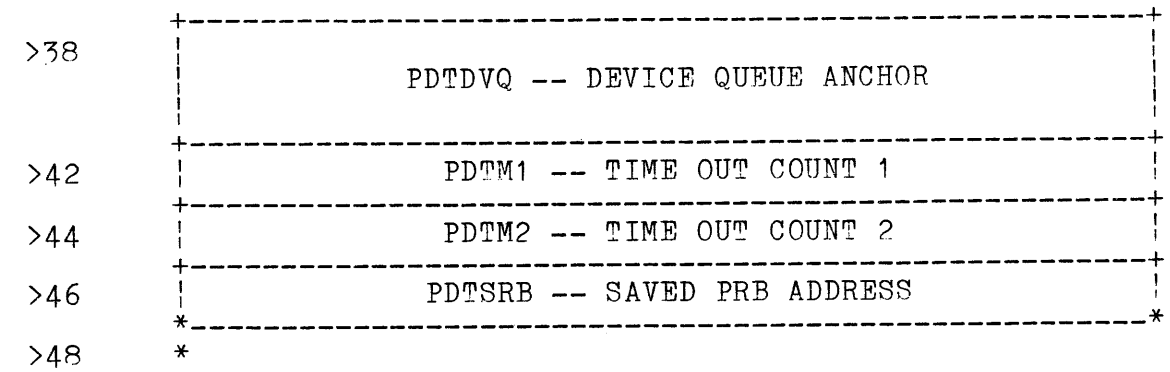


Figure 6-2 Physical Device Table (Part 2 of 2)

Hex. Byte	Field Name	Description
>00	PDTLNK	Address of the next PDT and the PDT expansion block for this PDT. All the PDTs are linked in a single list that is located in the D$DATA module.
>02	PDTMAP	Address of the DSR map file.
>04	PDTRO	This word begins the workspace to be used by the device service routine initial entry processor.
>06	PDTPRB	Address of buffered I/O supervisor call block.
>08	PDTDSF	Device status flags that are set by the system:

Bit	Meaning When Set
0	Device is opened; i.e., LUNOs are assigned to the device.
1	Device is busy.
2	Kill I/O at this device is in progress.
3	Task doing I/O at this device is being killed.
4	Make this device available (unassigned) at the end of this I/O operation.
5	Signals the task scheduler to reenter the DSR. This flag can be used by a DSR to wait for a device, by setting the flag and then returning to the system.
6	End-of-record processing needs to be done for this device; i.e., data transfer is complete.
7	0 = ASCII. 1 = JISCII.

>09 ------ Interrupt mask to be used by the DSR. This field is the interrupt level assigned to the device minus one, and is set at system generation time.

>0A PDTDTF Device type flags that are all set at system generation time except for the system disk flag that is set by the system loader.

Bit	Meaning When Set
0	File oriented device (if the flag is zero, the device is record oriented).
1	Device uses the TILINE data bus.
2	The time-out logic should be enabled for this device.
3	Device may only be used by privileged tasks.
4	This is a terminal (keyboard device) with a keyboard status block attached to the PDT.
5	This is a communications device.
6	This is the system disk.
7	A PDT extension exists.
8-11	Not used.
12-15	Device type code, as follows: 0 -- Dummy 1 -- Teleprinter 2 -- Line Printer 3 -- Cassette 4 -- Card Reader 5 -- Video Display Terminal 6 -- Disk and Diskette 7 -- Communications 8 -- Magnetic Tape and AMPL E -- AMPL Emulator F -- AMPL Trace Module

>0C PDTDIB Pointer to the word after the PDT itself. This may be the address of the keyboard status block (KSB), disk PDT extension (DPD), tape PDT extension (TPD) or line printer PDT extension (LPD) depending upon the type of device.

>0E PDTR5 thru PDTR11 Scratch registers to be used by the DSR.

>1C PDTCRU The CRU or TILINE address of the device.

>1E	PDTR13 PDTR14 PDTR15	These three words contain the saved context (WP, PC ST) to which the DSR returns control via a RTWP.
>24	PDT$	Pointer to the beginning of the PDT workspace; i.e., byte 4 of the PDT.
>26	PDTDSR	A pointer to the beginning of the DSR.
>28	PDTERR	Error code returned by the DSR.
>29	PDTFLG	Device flags as follows:

Bit	Meaning When Set
8	Use PRB in log message.
9	Receive mode for JISCII.
10	Transmit mode for JISCII.
11-12	Device state: online = 00, offline = 01, diagnostic = 10.
14	Operation failed bit.

>2A	PDTNAM	The 4-character device name.
>2E	PDTSL1, PDTSL2	For CRU devices, these words contain the controller image after an error. For TILINE devices, these words contain a pointer to the controller image after an error.
>32	PDTBUF	Not used.
>34	PDTBLN	Maximum length of a data buffer which may be transferred by the device in an I/O operation (e.g., 80 for a card reader). This is only necessary for CRU devices.
>36	PDTINT	The interrupt entry address of the DSR and reenter-me-address.
>38	PDTDVQ	The anchor for the queue of I/O requests for this device. The anchor has the same format as the queue anchor described in paragraph 6.2.
>42	PDTM1	The number of system time units in the time-out count for the device.
>44	PDTM2	The number of time units remaining in the time-out count before the system assumes that a device error has occurred. When the DSR starts an I/O operation, it should move the time-out count in bytes >42->43 to this word. This word is then used by the

		scheduler as the time-out counter. Each time a system time unit has elapsed, the scheduler decrements the time-out count and sets the time-out enable flag (bytes >0A->0B). If the counter goes to zero before a device interrupt occurs (and the DSR resets the counter or the flag), the system assumes that a device error has occurred and reports it.
>46	PDTSRB	The address of the queued supervisor call block, plus two (offset to PRB; see the DX10 Operating System Systems Programming Guide).
>48	*	

All PDTs must be defined during system generation. The PDTs are concatenated and inserted into the D$DATA module by GEN990.

Under DX10, PDTs for disks and terminals each have an extension that is used mainly by the interrupt processing routine of the DSR. The following paragraphs describe those extensions.

6.3.1 PDT EXPANSION BLOCK.

Every PDT has a 10-byte expansion block. The end of this block is pointed to by the link to the next PDT (PDTLNK). In the case of the last PDT (DSO1) where the link is zero, PDTLST in ROOT points to the expansion block. The format of the PDT expansion block is shown in Figure 6-3.

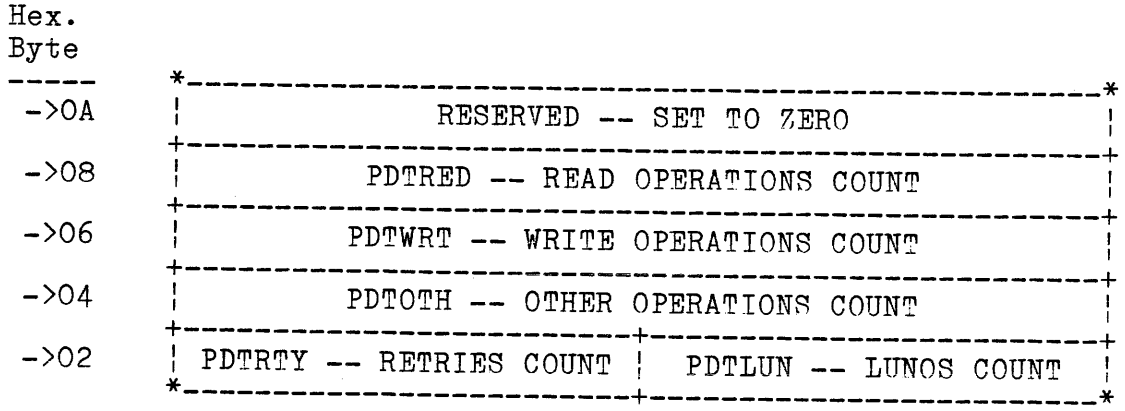


Figure 6-3 Physical Device Table Expansion Block

Hex. Byte	Field Name	Description
->0A	------	Reserved. Initialized to zero.
->08	PDTRED	The number of read operations that have been performed.
->06	PDTWRT	The number of write operations that have been performed.
->04	PDTOTH	The number of other operations that have been performed.
->02	PDTRTY	The number of retries.
->01	PDTLUN	The number of LUNOs assigned.

6.3.2 DISK PDT EXTENSION (DPD).

The extension that is appended to disk PDTs is 96 bytes long. The format is shown in Figure 6-4.

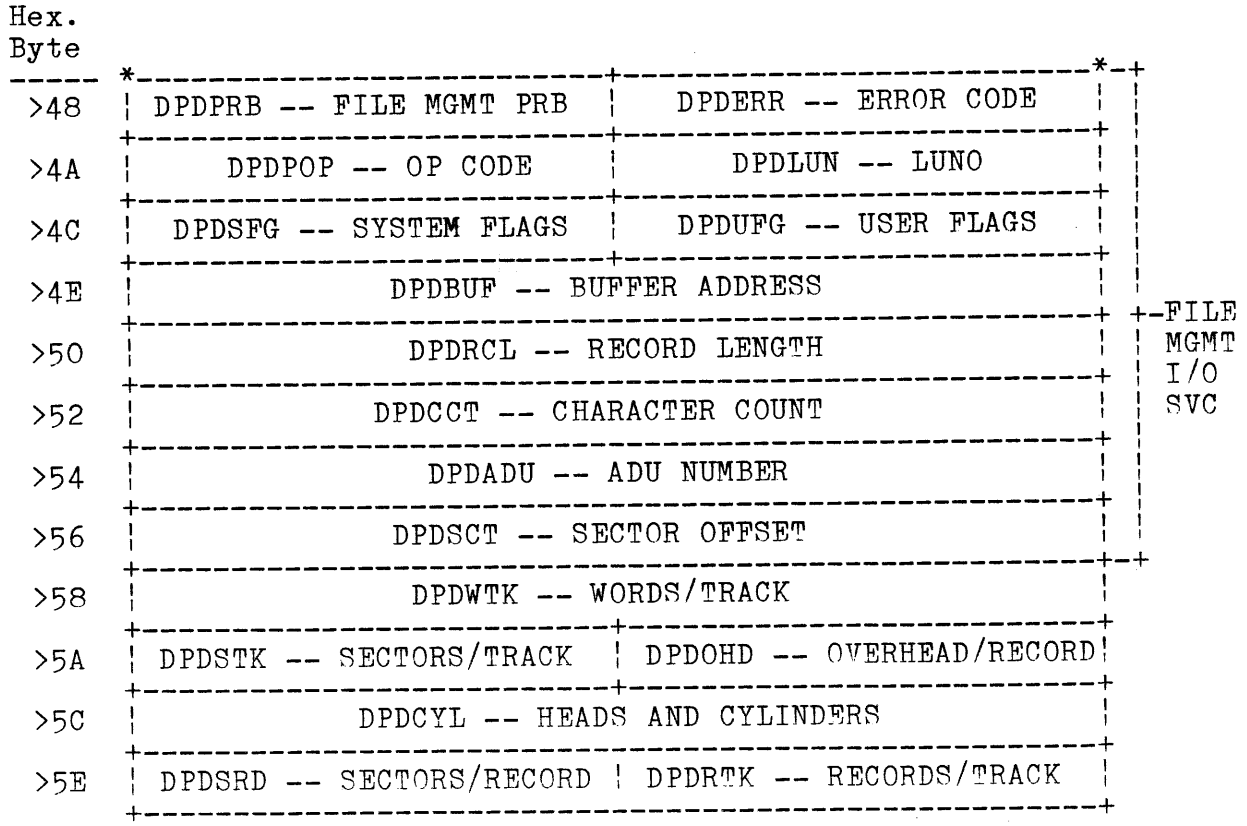


Figure 6-4 Disk PDT Extension (Part 1 of 2)

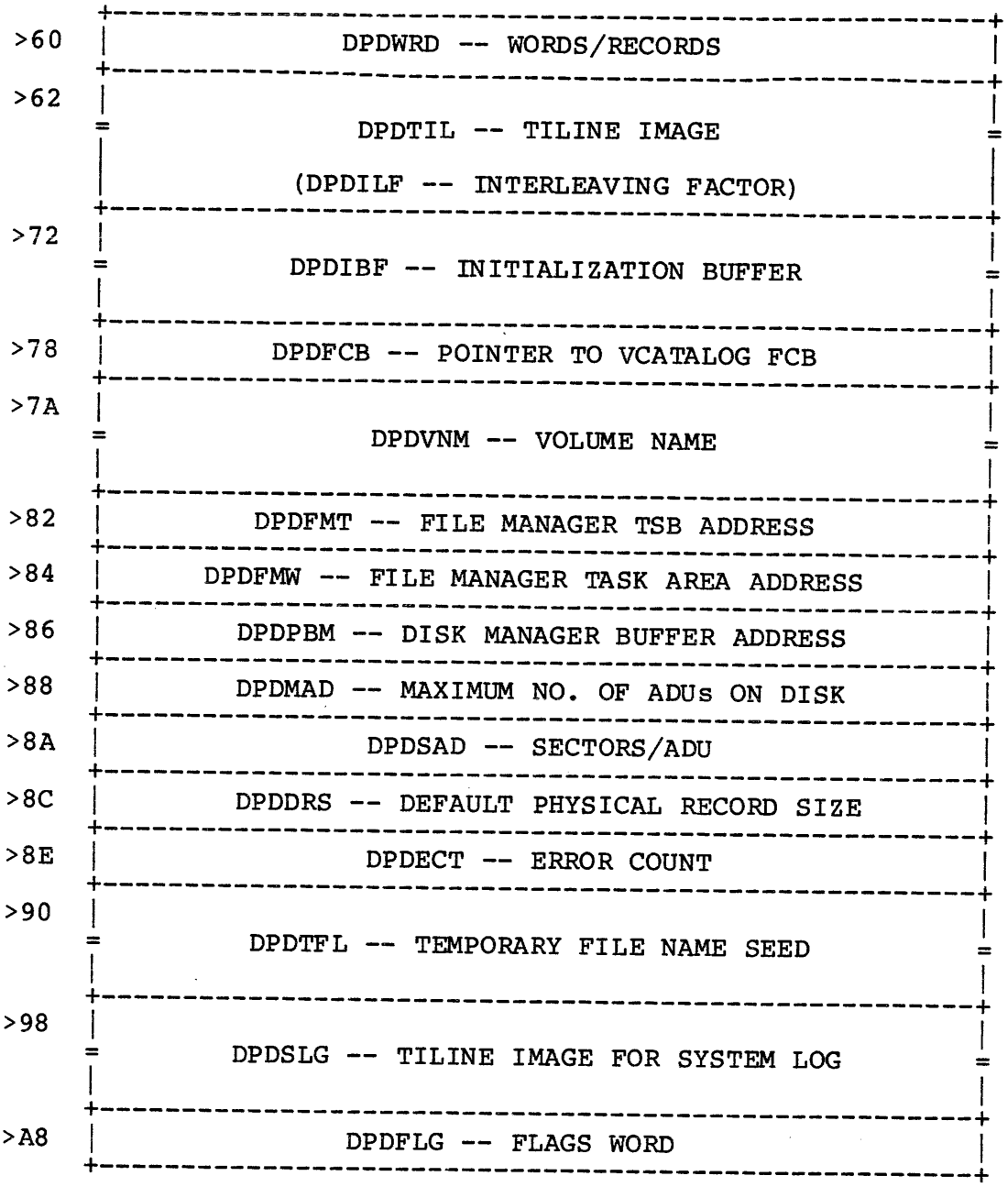


Figure 6-4 Disk PDT Extension (Part 2 of 2)

Hex. Byte	Field Name	Description
>48	DPDPRB	These 16 bytes are a copy of the file I/O supervisor call block.
>58	DPDWTK	Number of words per track on the disk.

>5A	DPDSTK	Number of sectors per track.
>5B	DPDOVH	Number of overhead bytes per physical record (equal one sector).
>5C	DPDCYL	Number of heads and cylinders as follows: Bit Meaning --- ------- 0-4 Number of heads on the disk. 5-15 Number of cylinders.
>5E	DPDSRD	Number of sectors per physical record (=1).
>5F	DPDRTK	Number of physical records per track.
>60	DPDWRD	Number of words per record.
>62	DPDTIL	An image of the eight TILINE controller registers for the disk.
>62	DPDILF	Interleaving factor, integer number, saves first word of TILINE controller registers image.
>72	DPDIBF	The buffer used to hold information returned by the Store Registers direct disk I/O call.
>78	DPDFCB	A pointer to the file control block for the disk volume directory (see paragraph 6.4).
>7A	DPDVNM	The 8-character name of the volume that is currently installed on the disk unit.
>82	DPDFMT	The address of the task status block (see paragraph 6.7) for the file management task for this disk controller.
>84	DPDFMW	The address of the file management task work area (see the section on D$DATA).
>86	DPDPBM	The address of the disk manager buffer for this disk.
>88	DPDMAD	The maximum number of ADUs on this disk.
>8A	DPDSAD	The number of sectors per ADU.
>8C	DPDDRS	The default physical record size for this disk, as defined to GEN990.
>8E	DPDECT	A count of the number of controller errors returned. This count is reinitialized when the system is booted.

>90	DPDTFL	The temporary file name last used.
>98	DPDSLG	A copy of the TILINE image used by the system log.
>A8	DPDFLG	Flags as follows:

Bit	Meaning
0	DPFRTY--No retry desired.
1	DPFRAW--Disk read after write.
2	DPFBRW--Bit map read after write.

6.3.3 TELEPRINTER DEVICE PDT EXTENSION (DIB).

The Device Information Block (DIB) is a data structure appended to the PDT that contains information about the current status of the device as well as information about how it was configured at system generation time. Figure 6-5 shows the format of the TPD extension.

Hex. Byte	
>00	ACU CRU ADDRESS
>02	ISR TYPE (COMM-1, TTY-5)
>03	LINE CONTROL TYPE (ALWAYS 0)
>04	READ ASCII TIMEOUT
>06	WRITE TIMEOUT
>08	READ DIRECT TIMEOUT 1
>10	READ DIRECT TIMEOUT 2
>12	SYSGEN ACCESS FLAGS
>13	STATE FLAGS
>14	LINE FLAGS
>15	TEMPORARY ACCESS FLAGS
>16	SPEED (ENCODED)
>17	END OF RECORD CHARACTER
>18	END OF FILE CHARACTER
>19	LINE TURN AROUND CHARACTER

Figure 6-5 Teleprinter Device PDT Extension (Part 1 of 2)

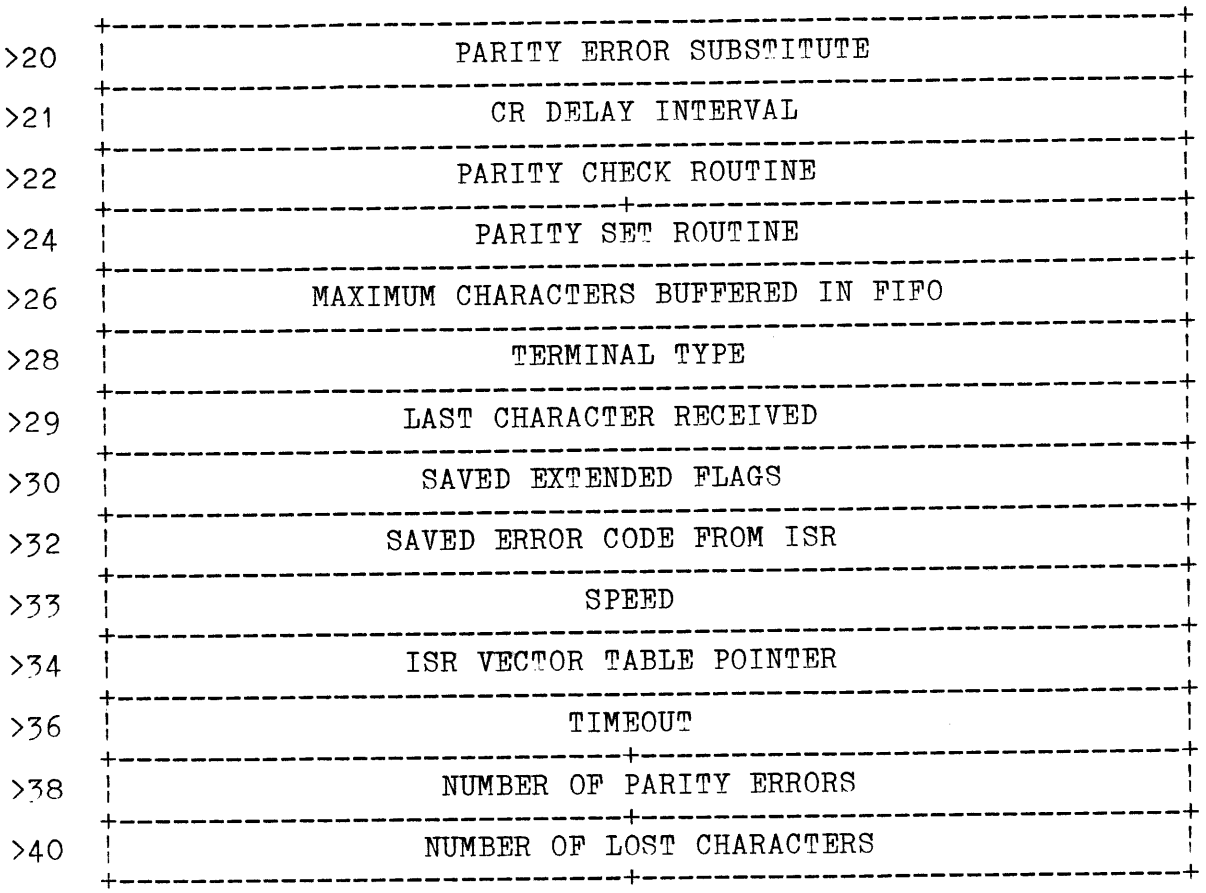


Figure 6-5 Teleprinter Device Extension to PDT (Page 2 of 2)

Hex. Byte	Description
>00	The CRU address of the ACU.
>02	The Interrupt service routine type, which is either communications (1) or teletype mode (5)
>03	The line control type, which is always 0.
>04	The Read ASCII timeout.
>06	The Write timeout.
>08	The Read Direct timeout 1.
>10	The Read Direct timeout 2.
>12	The sysgen access flags.
>13	The state flags as follows:

Bit	Meaning
0	Online
1	Connect in progress
2	Open
3	DLE received
4	Half-duplex line belongs to remote terminal
5	Resend flag
6-7	Unused

>14 Line flags as follows:

Bit	Meaning
0	Half-duplex modem
1	Switched line
2	Refuse call
3	Auto-disconnect enable
4	DLE/EOT for disconnect sequence
5	SCF ready/busy monitor
6	File transfer exclusive access
7	Half-duplex LTA enable

15 The temporary access flags as follows:

Bit	Meaning
0	No echo
1	Unused
2	Transmit parity enabled
3-4	Transmit parity type 00 = Even 01 = Odd 10 = Mark 11 = Space
5	Receive parity enabled
6-7	Receive parity type

>16 The speed (encoded).

>17 The end of record character.

>18 The end of file character.

>19 The line turnaround character.

>20 The parity error substitute character.

>21 The carriage return delay interval.

>22 The parity check routine pointer.

>24 The parity set routine pointer.

>26	The maximum characters buffered in FIFO.
>28	The terminal type.
>29	The last character received.
>30	The saved extended flags.
>32	The saved error code from interrupt service routine.
>33	The sysgened speed.
>34	The interrupt vector table pointer.
>36	The sysgened timeout.
>38	The number of parity errors.
>40	The number of lost characters.

6.3.4 KEYBOARD STATUS BLOCK (KSB).

Each keyboard type device supported by DX10 has a KSB appended to the PDT for the device. The KSB is generally used by the keyboard interrupt decoder of the device service routine. Special keyboard devices that are supported by user-written DSRs need not have a KSB unless the System Command Interpreter (SCI) is to be bid at the terminal, in which case the KSB must be present. Figure 6-6 shows the format of a KSB.

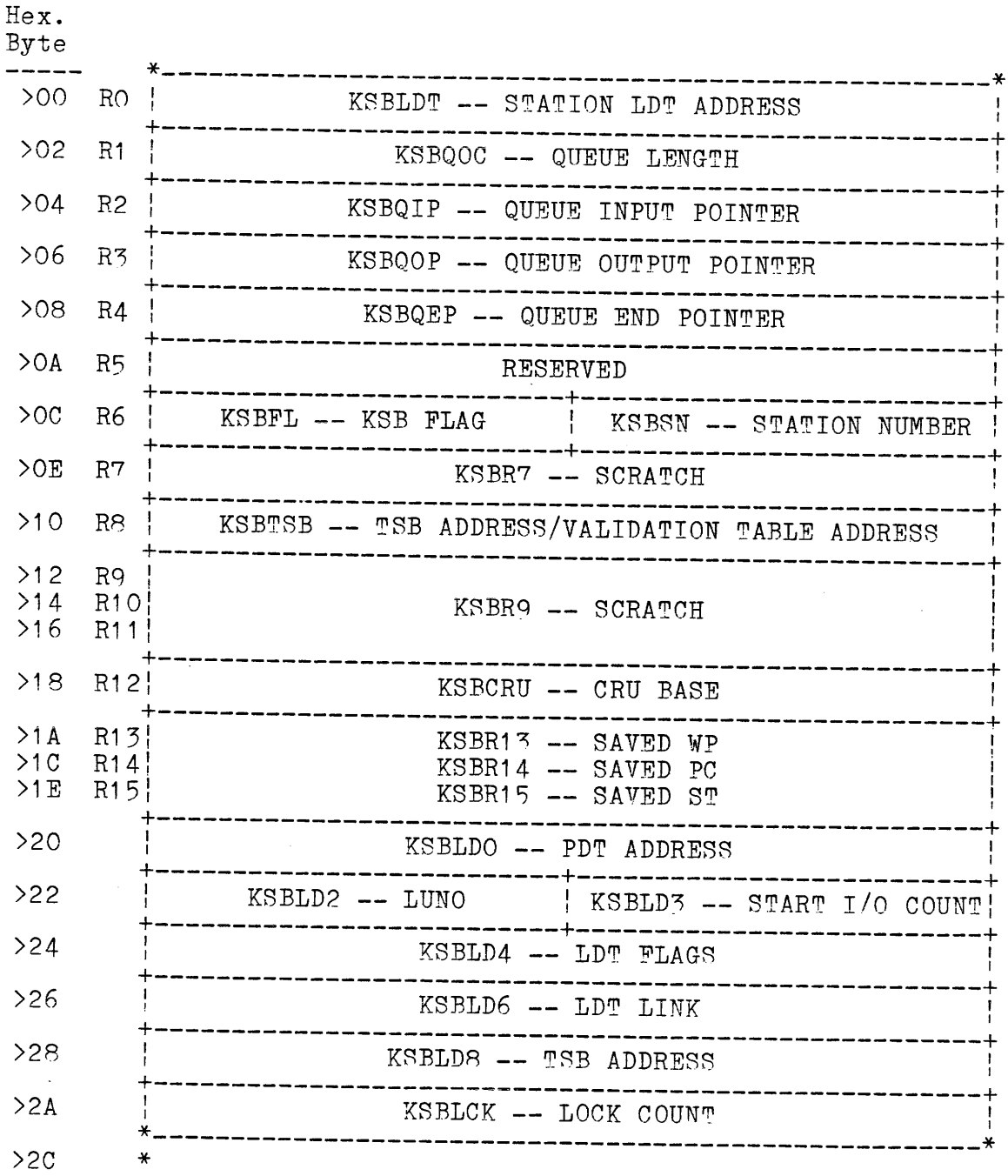


Figure 6-6 Keyboard Status Block

Hex. Byte	Field Name	Description
>00	KSBLDT	The offset into the KSB of the logical device table (LDT) anchor for station (terminal) local LUNOs.
>02	KSBQOC	The number of characters currently in the input character queue.
>04	KSBQIP	A pointer to the next byte of the character queue that is available to receive an input character.
>06	KSBQOP	A pointer to the oldest character in the input character queue, i.e., the next character to be picked up by the DSR.
>08	KSBQEP	A pointer to the word after the character queue. That word contains the length of the queue.
>0A	RESERVED	
>0C	KSBFL	Flags as follows:

Bit	Meaning When Set
0	Character mode (no mapping).
1	Enable the command interpreter bid logic.
2	Keyboard is in record mode (always set).
3	Bid the command interpreter.
4	The command interpreter is active.
5	Halt I/O.
6	Abort I/O.

>0D	KSBSN	Station (terminal) ID.
>0E	KSBR7	Scratch register for use by the DSR.
>10	KSBTSB	The address of the TSB of the task currently using the terminal if the terminal is in character mode. If a validation table is being used, this field contains the validation table address.
>12	KSBR9, KSBR10, KSBR11	Scratch registers for use by the DSR.

Hex. Byte	Field Name	Description
>18	KSBCRU	The CRU address of the terminal. For VDTs, this address is >10 more than in the PDT.
>1A	KSBR13	The saved context (WP, PC, ST) to which the DSR keyboard interrupt handling routine returns control via a RTWP instruction.
>20	KSBLDO	These 10 bytes form a logical device table (see paragraph 6.5 on LDTs) that serves as an anchor for the terminal local LDT list, as described in Section 1. Flag bit 0 in byte >24 is set to mark this LDT as an anchor. This LDT assigns terminal local LUNO 0 to the terminal itself.
>2A	KSBLCK	The lock out count, which is a count of the)the number of Read With Event Characters SVCs issued for this terminal.
>2C	*	

6.3.4.1 Video Display Terminal Extension (VDT).

The VDT is an extension to the KSB used by the 911 and 913 terminals. The offsets are expressed from the beginning of the Physical Device Table (PDT) to which the KSB has been appended. Figure 6-7 shows the format of the VDT extension.

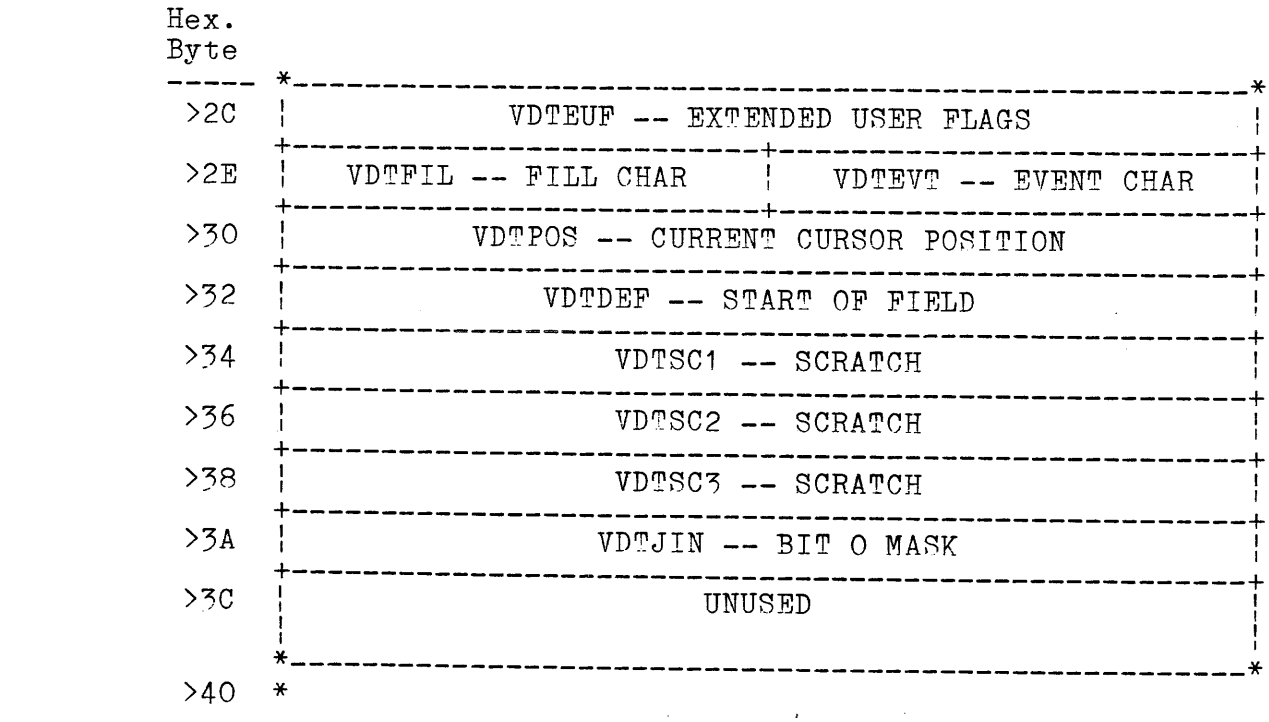


Figure 6-7 Video Display Terminal Extension to KSB

Hex. Byte	Field Name	Description
>2C	VDTEUF	The extended user flags to be used during the current operation.
>2E	VDTFIL	The fill character to be used during the current operation.
>2F	VDTEVT	The event character to be returned to the user call block.
>30	VDTPOS	The current position of the cursor.
>32	VDTDEF	The beginning of the field.
>34	VDTSC1	Scratch field for use by the extension.
>36	VDTSC2	Scratch field for use by the extension.
>38	VDTSC3	Scratch field for use by the extension.
>3A	VDTJIN	A mask that is used to set bit 0 of a character to the appropriate value.
>3C->3F	------	Not used.
>40	*	

6.3.4.1A Electronic Video Terminal Extension (VDT940).

The VDT940 is an extension to the KSB used by the 940 terminals. The offsets are expressed from the beginning of the Physical Device Table (PDT) to which the KSB has been appended. Figure 6-7A shows the format of the VDT940 extension.

NOTE

The meaning and position of the flags are for TI internal use only and may be moved, changed or deleted at any time.

The VDT940 has the following format:

Hex. Byte		
>2C	VDTEUF - EXTENDED USER FLAGS	
>2E	VDTFIL - FILL CHARACTER	VDTEVT - EVENT CHARACTER
>30	VDTPOS - CURSOR POSITION	
>32	VDTDEF - FIELD DEFINITION	
>34	VDTRED - READ DIRECT WORD	
>36	VDTSC1 - FLAG WORD 1	
>38	VDTSC2 - FLAG WORD 2	
>3A	VDTSC3 - TEMP LINK SAVE LOCATION	
>3C	RESERVED	GENSPD - TERMINAL SPEED
>3E	RESERVED	
>40	VDTATT - ATTRIBUTE SENT	VDTATB - ATTRIBUTE RECEIVE
>42	VDTSC4 - TEMP LINK SAVE LOCATION	
>44	VDTCNT - COUNT FOR VDTRED	
>46	VDTPTR - POINTER TO PRINTER PDT	
>48	VDTSC5 - LINK REGISTER FOR CHANGING CHARACTER SETS	
>4A	VDTMFL - FLAG WORD FOR MODE FLAGS	
>4C	VDTEDL - FLAG WORD FOR EVENT KEY FLAGS	
>4E	VDTSIZ - NO. OF CHAR SCREEN MEMORY	
>50	VDTFIS - SAVE R0 FOR FIFO	
>52	VDTSC6 - TEMPORARY STORAGE	

Figure 6-7A Electronic Video Terminal Extension to KSB (Part 1 of 2)

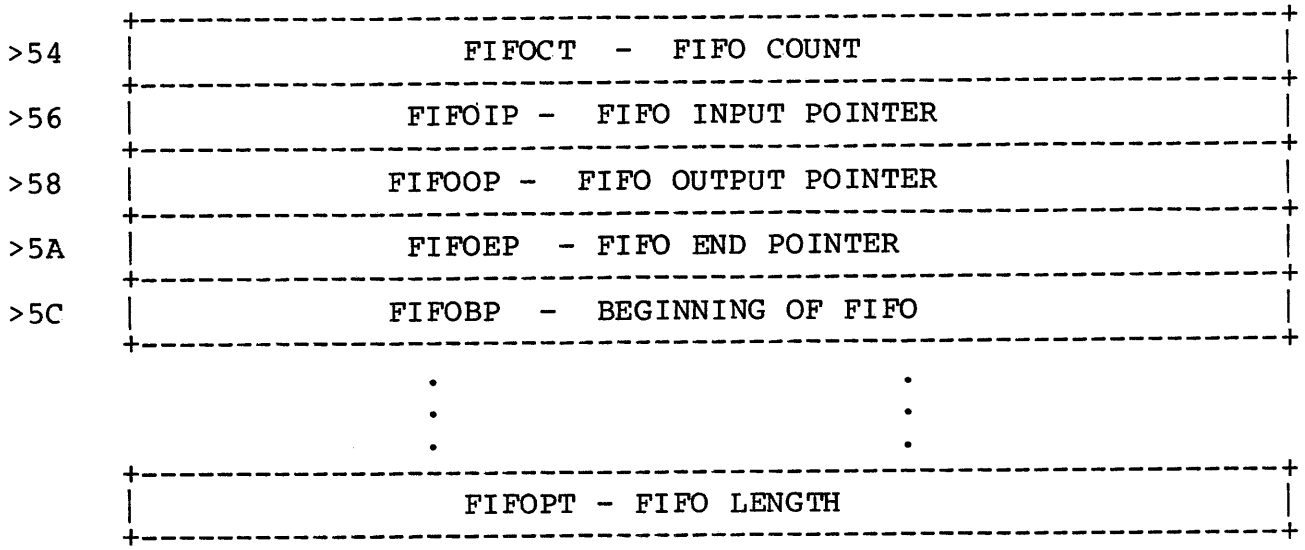


Figure 6-7A Electronic Video Terminal Extension to KSB (Part 2 of 2)

Hex. Byte	Field Name	Description
>2C	VDTEUF	The extended user flags to be used during current operation.
>2E	VDTFIL	The fill character to be used during the current operation.
>2F	VDTEVT	The event character to be returned to the user block.
>30	VDTPOS	The current position of the cursor.
>32	VDTDEF	The beginning of the field.
>34	VDTRED	The buffer or address of the buffer for the read to address based on the flag in VDTSC1 (see byte >36).

Hex. Byte	Field Name	Description
>36	VDTSC1	Flag word 1 as follows:

Bit	Meaning
0	Found beginning of Read Information
1	Next start of header was requested by the DSR
2	Read information goes into address in VDTRED (set to 1); otherwise stored in VDTRED
3	Found first ESC in response
4	Found a right parenthesis in string
5	A second ESC was found in string
6	An aid character* was found
7	A change character set was found
8	An attribute character was found
9	The cursor position was found
10	The requested read was finished
11	Indicates terminal is in insert mode
13	Indicates terminal is connected to computer
14	Indicates modem phone has rung
15	Instructs DSR to set re-enter me flag

Hex. Byte	Field Name	Description
>38	VDTSC2	Flag word 2 as follows:

Bit	Meaning
0	Timed out device
1	Time out for response
2	Printer has control of line
3	Printer wants control of line
4	EVT has control of channel
5	EVT wants control of channel
6	Extended character is in SC1
7	Extended character is in SC2
8	Alternate character set is in terminal
9	Alternate character set is on input
10	Alternate character set is on read to address
11	Graphics set is in terminal
12	Graphics set is on input

* Aid characters are the SEND key, and the 24 function keys.

Hex. Byte	Field Name	Description
>38	VDTSC2	Flag word 2 as follows: Bit — Meaning 13 — Graphics is set on read to address 14 — Terminal busy flag 15 — A mode 3 table check is in progress
>3A	VDTSC3	Link register save location
>3C	Reserved	
>3D	GENSPD	Terminal definition as follows: Bit — Meaning 0 — Set if switched 3-7 — Speed of terminal: Setting — Speed 11110 — 110 baud 10101 — 300 baud 10001 — 600 baud 11111 — 1200 baud 11011 — 2400 baud 10111 — 4800 baud 10011 — 9600 baud
>3E		Reserved
>40	VDTATT	Attribute sent to terminal as follows: Bit — Meaning 2 — Double wide characters 3 — Nondisplay 4 — Blink display 5 — Underline display character 6 — Reverse image display character 7 — High intensity display character
>41	VDTATB	Attribute received from terminal bit definition the same as VDTATT
>42	VDTSC4	Link register save location for outputting characters
>44	VDTCNT	Counter for VDTRED

Hex. Byte	Field Name	Description
>46	VDTPTR	Pointer to PDT of attached printer, if present
>48	VDTSC5	Link register for changing character sets
>4A	VDTMFL	Flag word for mode flags

Bit	Meaning
0	Pass through flag
1*	Terminate on receipt of ETX
2*	Terminate on receipt of ESC right parenthesis
3	Extended event characters
4	Extended display characters
5**	Allow ESC and SOH through write ASCII
6**	Do not set attributes
7**	132-column mode 3 flag
8	Modified data to caller
9	Extended character validation
10-15	Reserved

>4C VDTEDL Flag word for event key flags as follows:

Bit**	Meaning
0	Erase field
1	Right field
2	Cursor left out of Field
3	Tab
4	Reserved
5	Skip
6	Home
7	Return
8	Erase input
9	Reserved
10	Delete character
11	Insert character
12	Cursor right out of field
13	Enter
14	Left field
15	Reserved

*Flag applies only if bit 0 of VDTMFL is on.
**Flag applies only if bit 3 or bit 4 of VDTMFL is on.

Hex. Byte	Field Name	Description
>4E	VDTSIZ	Number of characters of screen memory
>50	VDTFIS	Save R0 for FIFO
>52	VDTSC6	Temporary storage
>54	FIFOCT	FIFO count
>56	FIFOIP	FIFO input pointer
>58	FIFOOP	FIFO output pointer
>5A	FIFOEP	FIFO end pointer
>5C	FIFOBP	Beginning of FIFO
.	.	.
.	.	.
.	.	.
	FIFOPT	FIFO length (beginning address of FIFO and length of FIFO set at system generation time)

6.3.4.2 KSR Extension (KSR).

The KSR is an extension to the KSB used by keyboard-type devices such as the 733. The offsets are expressed from the beginning of the Physical Device Table (PDT) to which the KSB has been appended. Figure 6-8 shows the format of the KSR extension.

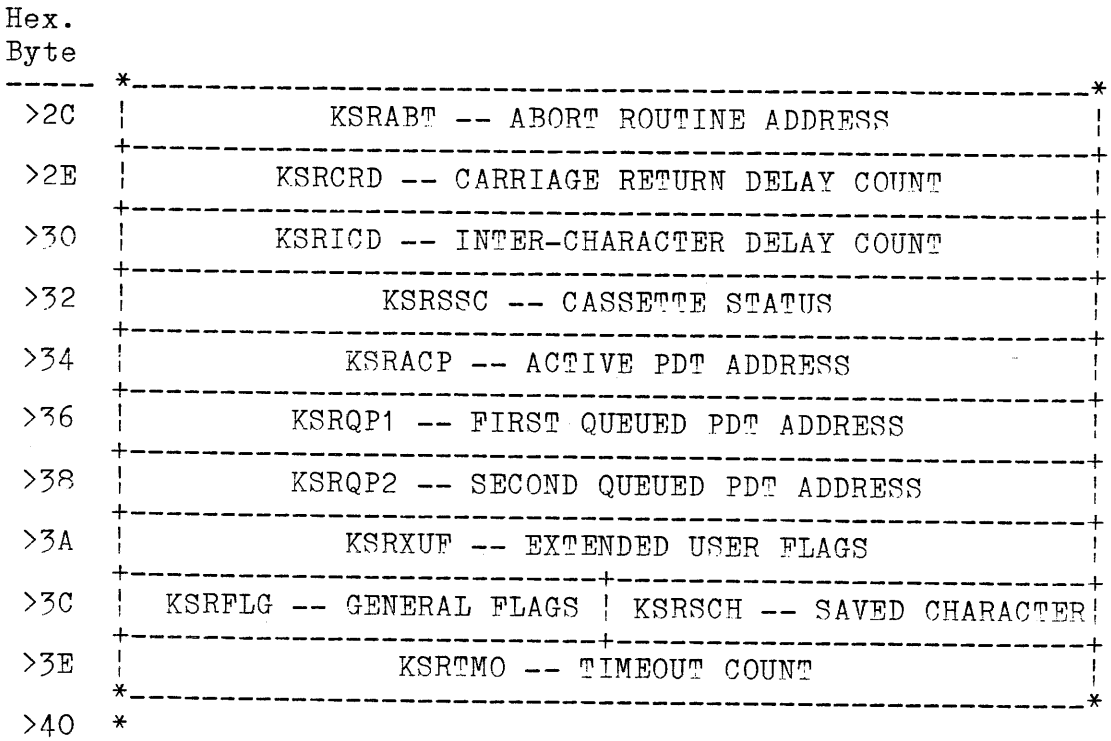


Figure 6-8 KSR Extension to KSB

Hex. Byte	Field Name	Description
>2C	KSRABT	The address of the abort routine.
>2E	KSRCRD	The carriage return delay count.
>30	KSRICD	The inter-character delay count.
>32	KSRSSC	The status of the cassettes.
>34	KSRACP	The address of the active PDT for the 733.
>36	KSRQP1	The address of the first queued PDT.
>38	KSRQP2	The address of the second queued PDT.
>3A	KSRXUF	Extended user flags.

>3C	KSRFLG	General flags as follows:

Bit	Meaning When Set
0	Hang up condition.
1	Time out switch.
2	SCI is active during hang up.
3	A data carrier drop was detected.
4	Shift in/out for JISCII.
5	Direct character input requested.

>3D	KSRSCH	Character saved for JISCII output.
>3E	KSRTMO	Timeout count for hang condition.
>40	*	

6.3.4.3 820 Extension (T82).

T82 is an extension of the KSB for the 820 terminal. The offsets are expressed from the beginning of the Physical Device Table (PDT) to which the KSB has been appended. Some fields must be compatible with the KSR because they are used outside of the Device Service Routine (DSR). Figure 6-9 shows the format of the 820 extension.

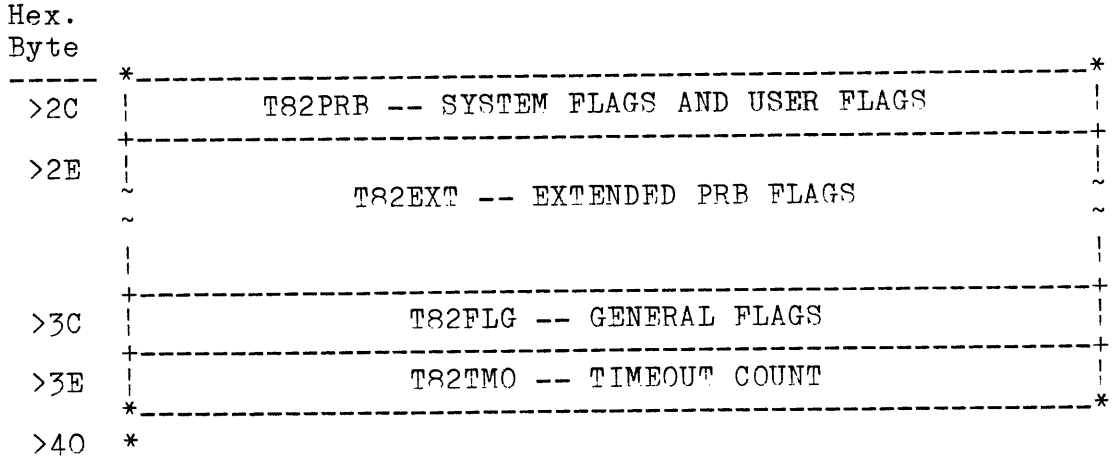


Figure 6-9 820 Extension to KSB

Hex. Byte	Field Name	Description
>2C	T82PRB	System and user flags to be used during the current operation.
>2E	T82EXT	Flags from the extended Physical Record Block (PRB); 0 if not extended.
>3C	T82FLG	General flags as follows:

Bit	Meaning When Set
0	Hang up condition.
1	Time out switch.
2	SCI is active during hang up.
3	A data carrier drop was detected.
4	Shift in/out for JISCII.
5	Direct character input requested.

>3E	T82TMO	The timeout count.
>40	*	

6.3.4.4 Character Queue.

The character queue follows the KSB and KSB extension for terminal devices. Currently, the character queue starts at >64 from the beginning of the KSB. However, all references to the queue should be through the KSB pointers. The length of the queue is set at SYSGEN time. The word following the queue buffer is the length of the queue.

6.3.5 LINE PRINTER EXTENSION (LPD).

The line printer extension to the PDT is used for both fast (2230/2260) and slow (e.g., 810) line printers. Figure 6-10 shows the format.

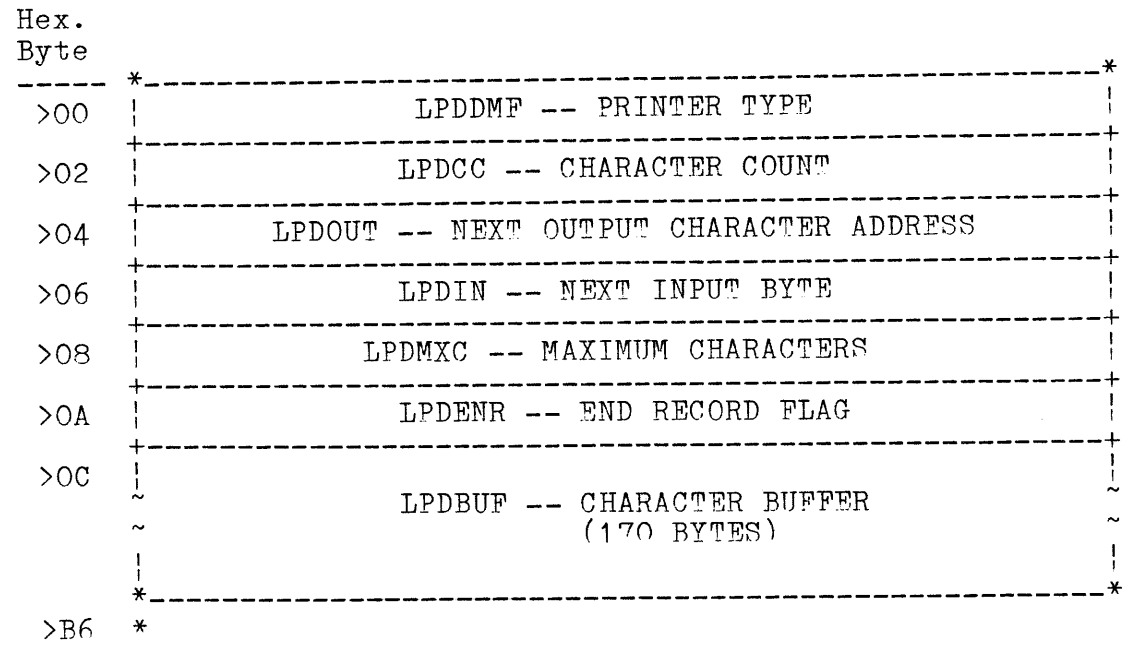


Figure 6-10 Line Printer Extension

Hex. Byte	Field Name	Description
>00	LPDDMF	Describes the type of line printer. A zero denotes a fast printer and a 2 denotes a slow printer.
>02	LPDCC	The number of characters currently in the buffer.
>04	LPDOUT	A pointer to the next character to be output (unless LPDCC = 0).
>06	LPDIN	A pointer to the next free byte in which a character can be stored (unless LPDCC = LPDMAX).
>08	LPDMXC	The maximum number of characters that may be stored in the buffer.
>0A	LPDENR	Used as a flag to cause end record processing.
>0C	LPDBUF	The 170-byte character buffer.
>B6	*	

6.3.6 TAPE EXTENSION (TPD).

The tape extension to the PDT is similar to the disk extension. Currently, the disk and tape are the only TILINE devices. Certain fields in these extensions that are used outside the Device Service Routine (DSR) must be in the same location. Therefore, the tapee extension must be the same size as the disk extension. Figure 6-11 shows the format of the TPD.

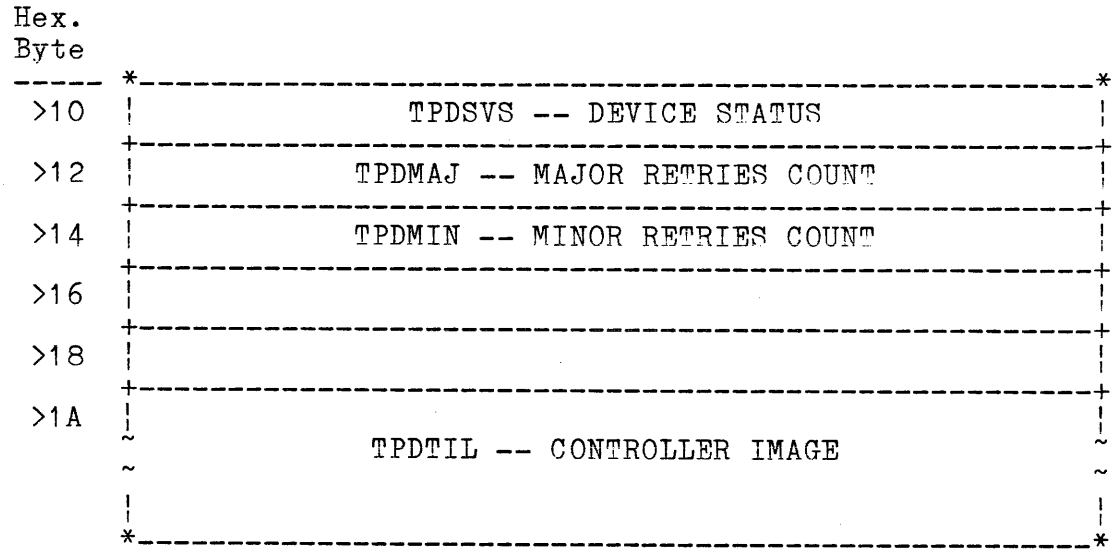


Figure 6-11 Tape Extension

Hex. Byte	Field Name	Description
>10	TPDSVS	The device status for a device characteristics call.
>12	TPDMAJ	A count of the number of major retries left.
>14	TPDMIN	A count of the number of minor retries left.
>1A	TPDTIL	The controller image created to start an operation.

6.3.7 FLOPPY DISKETTE EXTENSION (FPD).

The floppy diskette PDT extension is used for CRU-type floppy diskettes (FD800). TILINE floppy diskettes (FD1000) use the normal disk PDT extension. Figure 6-12 shows the format of an FPD.

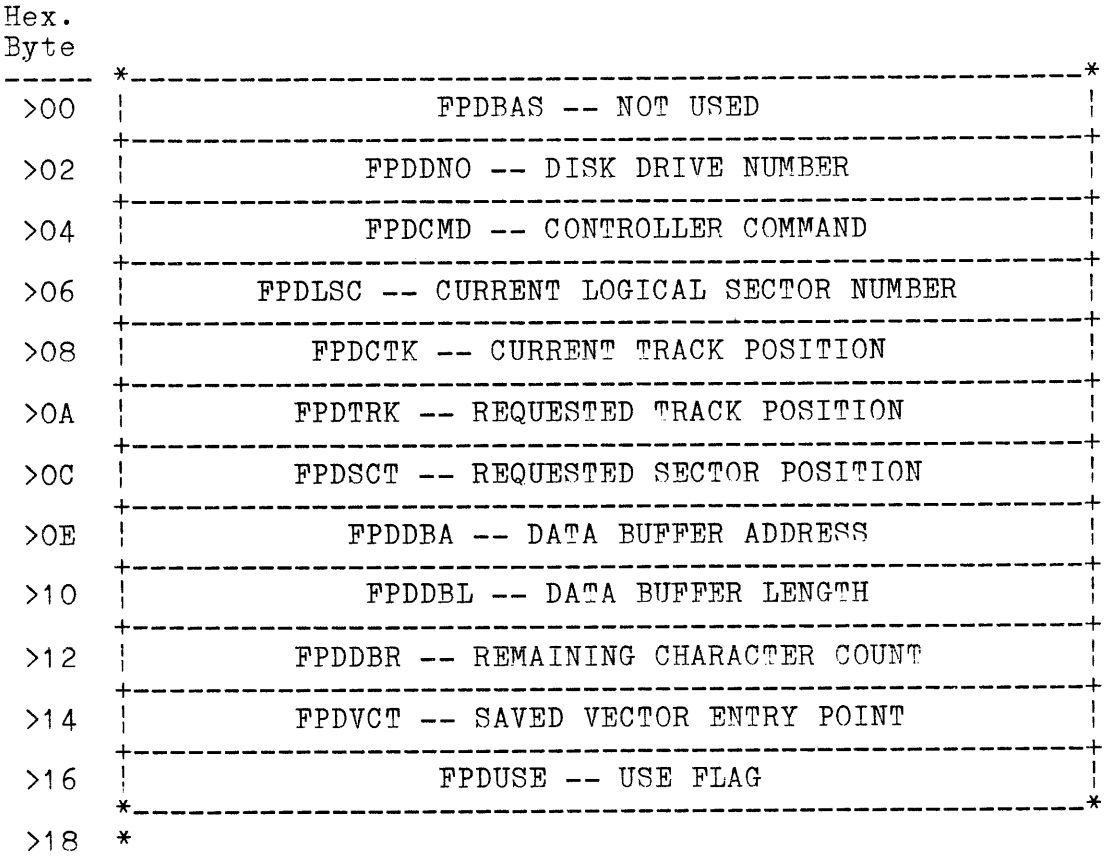


Figure 6-12 Floppy Diskette PDT Extension

Hex. Byte	Field Name	Description
>00	FPDBAS	Not used.
>02	FPDDNO	The disk drive number is contained in bits 4 and 5 of this word.
>04	FPDCMD	The command that is to be issued to the disk controller.
>06	FPDLSC	The current logical sector number.
>08	FPDCTK	The current track position.
>0A	FPDTRK	The requested track position.
>0C	FPDSCT	The requested sector position within the track.
>0E	FPDDBA	The data buffer address.
>10	FPDDBL	The data buffer length.
>12	FPDDBR	The remaining character count for multi-sector transfers.
>14	FPDVCT	The saved vector entry point within the device service routine (DSR).
>16	FPDUSE	Used as a flag to indicate that this PDT has control of the disk controller.
>18	*	

6.4 FILE CONTROL BLOCK (FCB)

Within DX10, files are represented by, or accessed through, file control blocks. As described in Section 1, whenever a file is accessed, a tree structure of FCBs is built into memory. This structure resembles the directory/file hierarchy on the disk. FCBs are built in the system table area by the File Utility Assign LUNO processor, and are basically a memory resident copy of the file descriptor record (FDR) of the file. Figure 6-1? shows the format of a file control block.

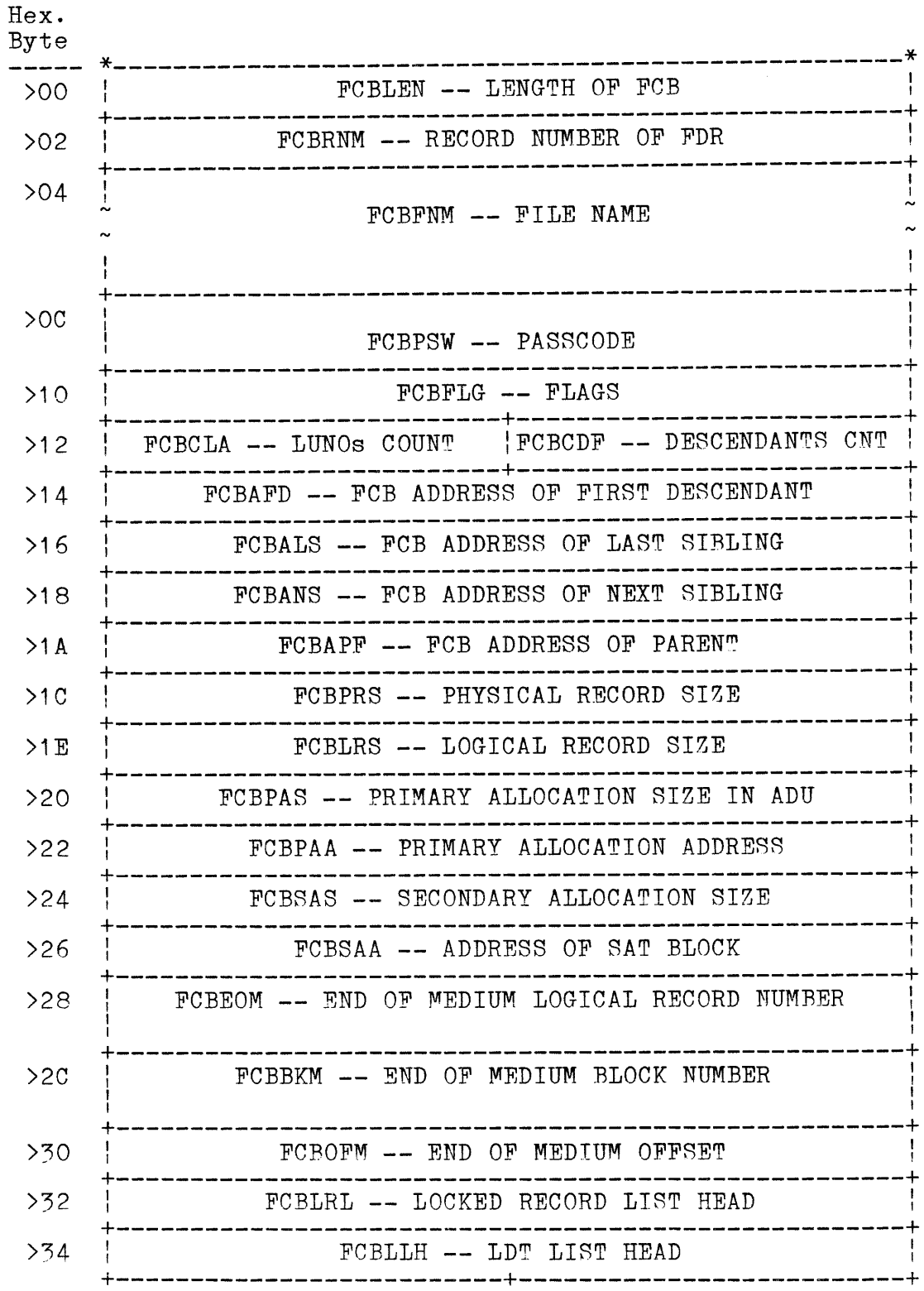


Figure 6-13 File Control Block (FCB) -- Part 1 of 2

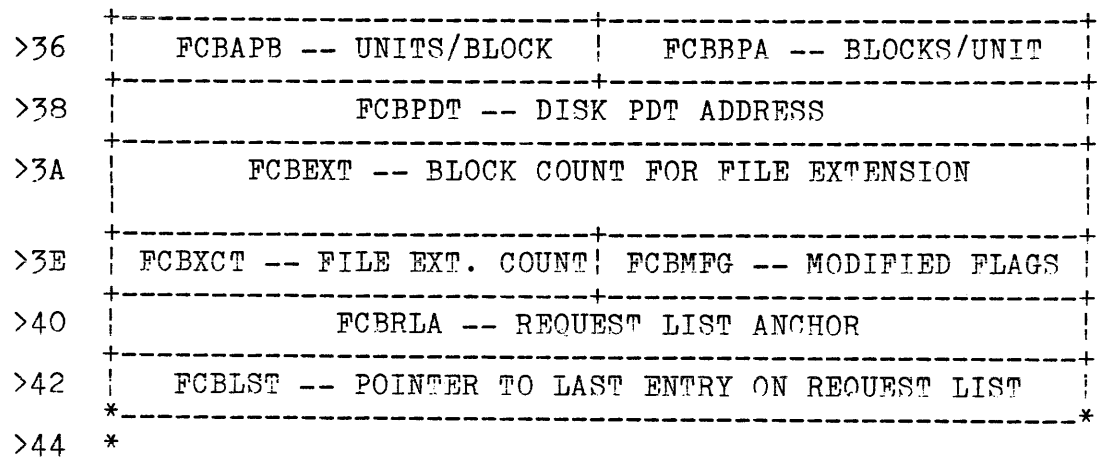


Figure 6-13 File Control Block (FCB) -- Part 2 of 2

Hex. Byte	Field Name	Description
>00	FCBLEN	Length in bytes of the FCB and any extensions (described later in this section). If there are no extensions, the value of this field is zero.
>02	FCBRNM	The number of the directory file record that contains the file descriptor record for this file.
>04	FCBFNM	File name (eight characters)
>0C	FCBPSW	Passcode, reserved for future extension.
>10	FCBFLG	Flags, which are the same as in a file descriptor record except for bits 12-13. The flags have the following meanings: Bit Meaning --- ------- 0-1 File usage flags: 00 No special usage 01 Directory 10 Program file 11 Image file 2-3 Data format: 00 Binary 01 Blank Suppressed 10 Reserved for ASCII & print forms control 11 Reserved

4 Allocation type:
0 Bounded
1 Unbounded

5-6 File type:
00 Reserved for device
01 Sequential
10 Relative record
11 Key indexed

7 Write protection flag:
0 Not write protected
1 Write protected

8 Delete protection:
0 Not delete protected
1 Delete protected (file cannot be deleted)

9 Temporary file flag:
0 Permanent file
1 Temporary file

10 Blocked file flag:
0 Blocked
1 Unblocked

11 Alias flag:
0 Not an alias
1 An alias file name

12-13 Most restrictive access applied to all users of the file:
00 Exclusive write
01 Exclusive all
10 Shared
11 Read only

14-15 Reserved

>12 FCBCLA Number of LUNOs assigned to the file.

>13 FCBCDF Number of descendant FCBs in memory. Only a directory file may have descendants. Descendants are all files that are cataloged under this directory and any sub-directories.

>14 FCBAFD Address of the FCB of the first descendant (i.e., the first file cataloged under this directory that was accessed, and is still being accessed).

>16 FCBALS Sibling pointers. All FCBs of files cataloged under the same directory are laterally linked by these pointers, as described in Section 1.

>1A	FCBAPF	Address of the FCB of the directory under which this file is cataloged.
>1C	FCBPRS	Size in bytes of a physical record of this file.
>1E	FCBLRS	Size in bytes of a logical record.
>20	FCBPAS	Size in allocable disk units of the primary file allocation.
>22	FCBPAA	Starting ADU number of the primary allocation.
>24	FCBSAS	Size in ADUs of the secondary allocation.
>26	FCBSAA	Address of te in-memory copy of the secondary allocation table (SAT) for the file. The SAT is an exact copy of the last 64 bytes of the file descriptor record (FDR) and is located in the system table area.
>28	FCBEOM	The number of the logical record immediately following the last allocated logical record (end of medium).
>2C	FCBBKM	The physical record in which the file allocation ends (end of medium).
>30	FCBOFM	The sector offset into the physical record that marks the end of medium.
>32	FCBLRL	The head of a singly linked list of record lock tables, which are also located in the system table area, each of which points to a locked record of the file.
>34	FCBLLH	Head of a linked list of all logical device tables that represent LUNOs assigned to this file.
>36	FCBAPB	The number of ADUs per physical record.
>37	FCBPBA	The number of physical records per ADU.
>38	FCBPDT	The address of the PDT for the disk on which this file is written.
>3A	FCBEXT	Block count for file extension.
>3E	FCBXCT	Number of secondary allocations.

>3F FCBMFG Flags as follows:

Bit	Meaning When Set:
0	End of medium for this file has changed
1	Data has been written into the file
2	FCB is busy

>40 FCBRLA Pointer to the next buffered I/O request for this file (I/O requests for the same file are queued from the FCB until processed by file management).

>42 FCBLST Pointer to the last buffered I/O request on the list for this file.

>44 *

6.4.1 KIF EXTENSION TO THE FCB.

When an FCB represents a key indexed file, an extension to the FCB is used to contain additional information. Figure 6-14 shows the format of a KIF extension.

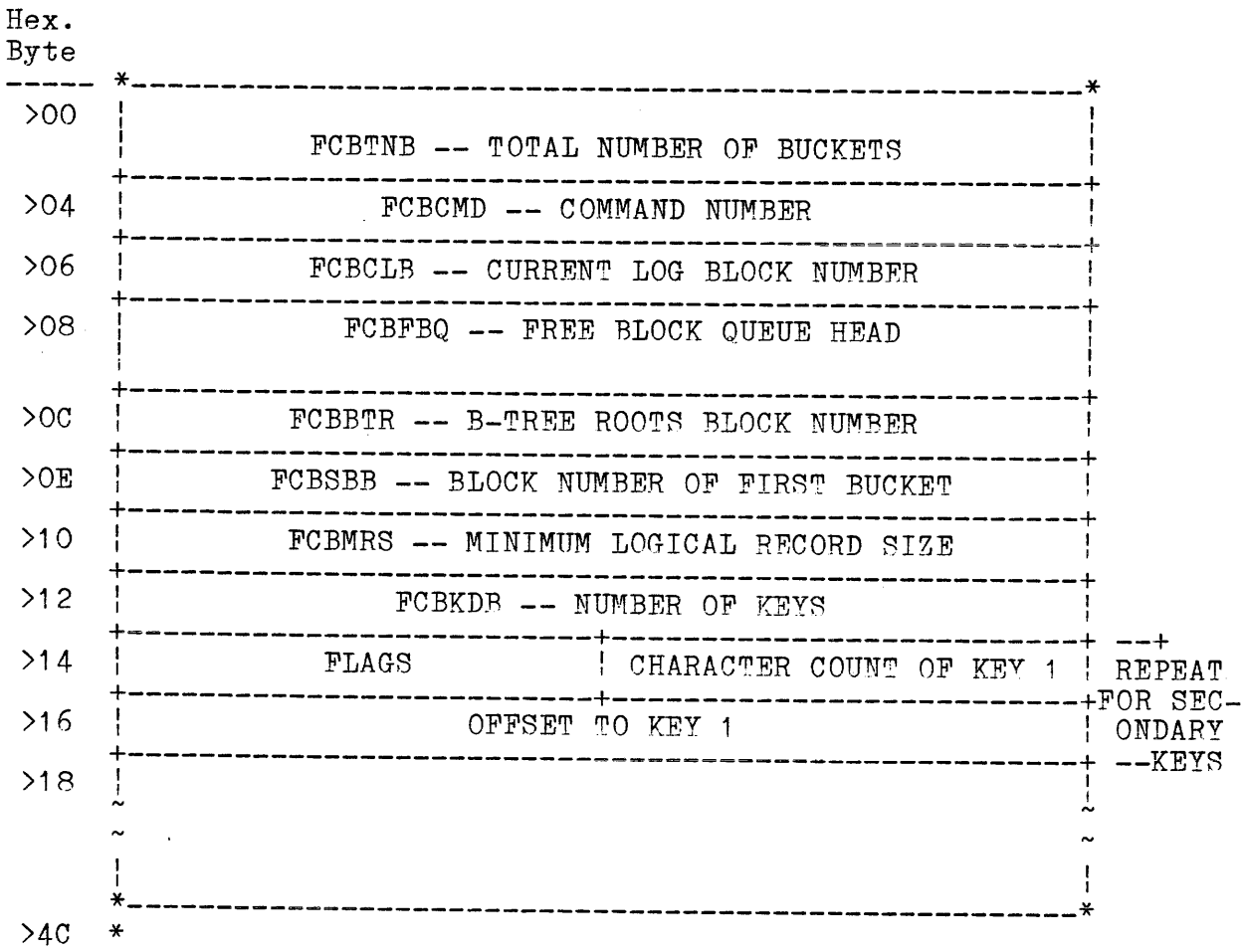


Figure 6-14 FCB Extension for Key Indexed Files

Hex. Byte	Field Name	Description
>00	FCBTNB	Total number of buckets allocated in the file.
>04	FCBCMD	The opcode of the current command (used for logging).
>06	FCBCLB	The physical record number of the currently used log block (see the discussion on key indexed files in Section 4).
>08	FCBFBQ	The physical record number (block number) of the first record in a linked list of available records.
>0C	FCBBTR	The number of the first physical record containing a B-tree root.
>0E	FCBSBB	The number of the first physical record containing the first bucket.
>10	FCBMRS	Minimum logical record size, in bytes, needed to contain all defined keys.
>12	FCBKDB	These fields are in-memory duplicates of bytes 6-63 (>06->3F) of the disk resident key descriptor record.
>4C	*	

6.4.2 QUEUE EXTENSION TO THE FCB.

When the file represented by an FCB is a directory, a queue anchor of the form shown in Figure 6-1 is appended to the FCB. The queue anchored is a queue of TSBs of tasks waiting for access to the directory file. The queue is implemented to prevent both file management and file utility from updating the same directory record concurrently (e.g., if one task were writing to a file and another task were renaming the file at the same time).

Hex. Byte	Field Name	Description
>00	FCBQUE	Address of the TSB of the newest task waiting for access.
>02	------	TSB address of the oldest task waiting for access.
>04	FCBLOK	TSB address of the task currently accessing the directory.

>06	FCBQFL	Flags. Must have a value of >40.
>07	------	Task Id of server. Must be zero.
>08	FCBQST	Task state. Must be >1E.
>09	------	Count of items on queue.

6.4.3 RECORD LOCK TABLE (RLT).

An RLT is a 10-byte block of system table area that points to a file record that is locked. All locked records of a file are represented by a linked list of RLTs, ordered by an ascending disk address, which is headed by a word in the file control block. Each time a record is locked, file management builds a new RLT and links it on the list. Figure 6-15 shows the format of a record lock table.

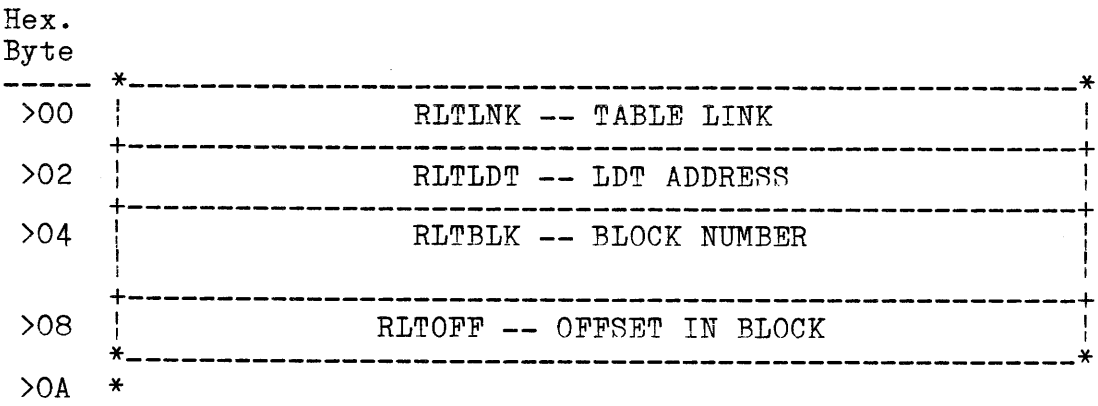


Figure 6-15 Record Lock Table (RLT)

Hex. Byte	Field Name	Description
>00	RLTLNK	Link to the next RLT (0 = end of list).
>02	RLTLDT	Address of the logical device table that represents the LUNO assigned by the task that locked this record.
>04	RLTBLK	Number of the physical record of the file in which the locked logical record is written.
>08->09	RLTOFF	The logical record within the above addressed physical record that is locked.

6.4.4 PROGRAM FILE EXTENSION TO THE FCB.

When an FCB represents a program file, an extension to the FCB is used to contain additional information. Figure 6-16 shows the format of the program file extension.

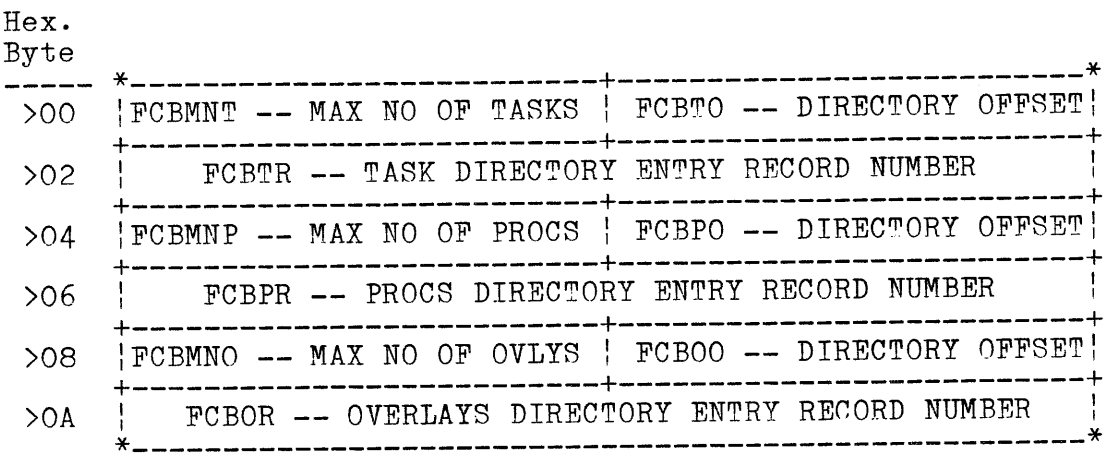
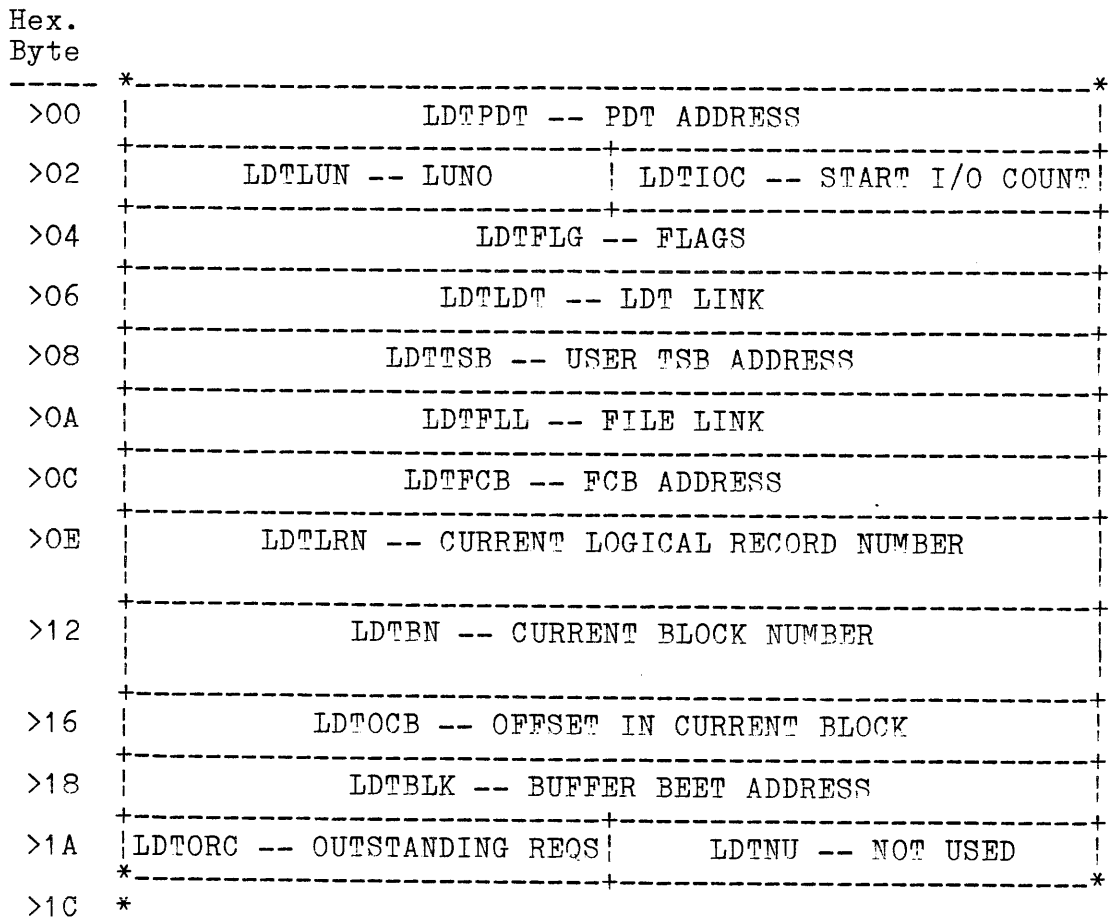


Figure 6-16 FCB Extension for Program Files

Hex. Byte	Field Name	Description
>00	FCBMNT	Maximum number of task entries in the file.
>01	FCBTO	Task directory entry offset.
>02	FCBTR	Task directory entry record number.
>04	FCBMNP	Maximum number of procedure entries in the file.
>05	FCBPO	Procedure directory entry offset.
>06	FCBPR	Procedure directory entry record number.
>08	FCBMNO	Maximum number of overlay entries in the file.
>09	FCBOO	Overlay directory entry offset.
>0A	FCBOR	Overlay directory entry record number.
>0C	*	

6.5 LOGICAL DEVICE TABLE (LDT)

Logical device tables are built in the system table area by the file utility assign LUNO processor. Whenever a LUNO is assigned to a file or device, an LDT is built to represent the logical unit to the system. Figure 6-17 shows the format of an LDT.



Note: Bytes >00->09 are the same for file and device LDTs. Bytes >10->1B are used only for file LDTs.

Figure 6-17 Logical Device Table (LDT)

Hex. Byte	Field Name	Description
>00	LDTPDT	Address of the PDT to which the LUNO is assigned. For LUNOs assigned to files, this is the PDT for the disk unit on which the file is written.
>02	LDTLUN	The LUNO which this LDT represents.
>03	LDTIOC	The number of initiate I/O operations currently being performed at this LUNO.
>04	LDTFLG	Flags as follows: Bit — Meaning 0 — This LDT is used to anchor a list of LDTs (e.g., in a KSB). 1-2 — Access privileges: 00 Exclusive write 01 Exclusive all 10 Shared 11 Read only 3 — The file to which the LUNO is assigned was created by the Assign LUNO SVC. 4-5 — Scope of LDT anchor (i.e., what kind of LUNO): 00 Task local 01 Station local 10 Global 11 Undefined 6 — Deferred write error. 7 — File is forced write. 8 — LUNO is a system LUNO (cannot be released). 9 — LUNO assigned to a file. 10 — LUNO is busy. 11 — Event mode is locked in record mode. 12 — Initiate I/O is being performed. 13 — Abort I/O is being performed. 14 — Unblocked access. 15 — Print flag.
>06	LDTLDT	Link to the next LDT in the LDT inverted tree structure described in Section 1.
>08	LDTTSB	TSB address of the task that opened the LUNO.

```
*********************************************************************
* The remaining fields (bytes >0A through >1B) are only defined for *
* LDTs that represent LUNOs assigned to files.  When a LUNO is      *
* released, or the task that assigned the LUNO terminates, the LDT  *
* is released by the file utility Release LUNO routine.             *
*********************************************************************
```

>0A	LDTFLL	Link to the next LDT representing a LUNO assigned to the same file. All LDTs assigned to the same file are linked together, and anchored in the file control block.
>0C	LDTFCB	Address of the file control block for the file to which the LUNO is assigned.
>0E	LDTLRN	The number of the logical record that is currently being accessed.
>12	LDTBN	The number of the physical record that is currently being accessed.
>16	LDTOCB	The offset into the physical record to the current logical record.
>18	LDTBLK	The beet address of the buffer that contains the last physical record transferred (read or written) through this LUNO (zero if the buffer has been released).
>1A	LDTORC	Number of outstanding requests for I/O to this LUNO.
>1B	LDTNU	Not used.
>1C	*	

6.6 BUFFERED CALL BLOCK

Whenever a supervisor call that is processed by a queue serving routine is issued, the call block is buffered in the system table area and queued for the SVC processing routine. The first four words of the buffer contain the following system overhead.

Hex. Byte	Field Name	Description
>00	------	Link to the next entry in the queue.
>02	------	Address of the TSB of the task issuing the SVC.
>04	------	Address of the call block within the calling task.
>06	------	Address of the LDT to which an I/O operation is directed (used only for I/O SVCs).

The remainder of the buffer contains the call block and any extension blocks or data buffers.

6.7 TASK STATUS BLOCK (TSB)

Tasks are represented within DX10 by a task status block (TSB). TSBs are built in the system table area by the bidding routines TMBID0 and TM$BID. A TSB is released by the termination task, TMDGN, when a task terminates, unless the task is a queue server. TSBs of inactive queue servers are not released unless more system table area is needed. Figure 6-18 shows the format of a task status block.

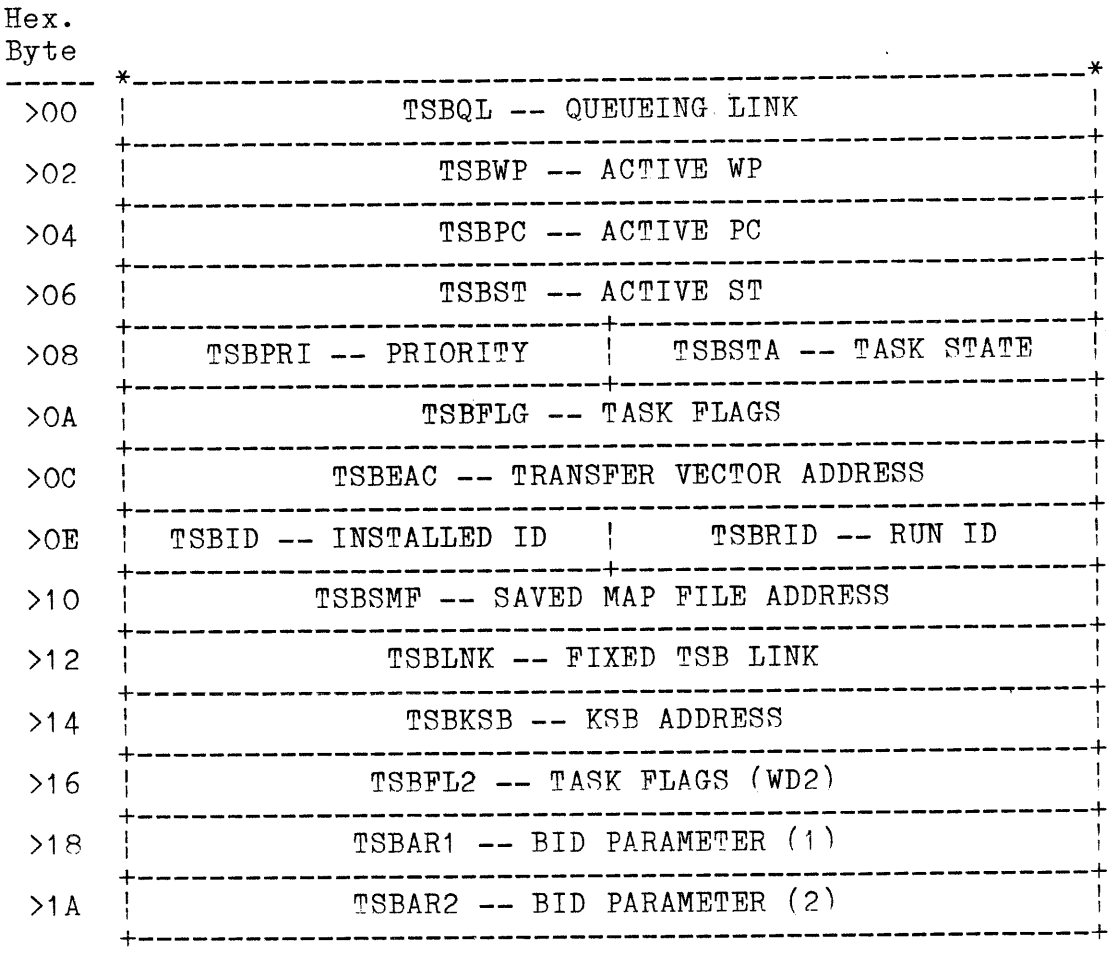


Figure 6-18 Task Status Block (TSB) -- Part 1 of 3

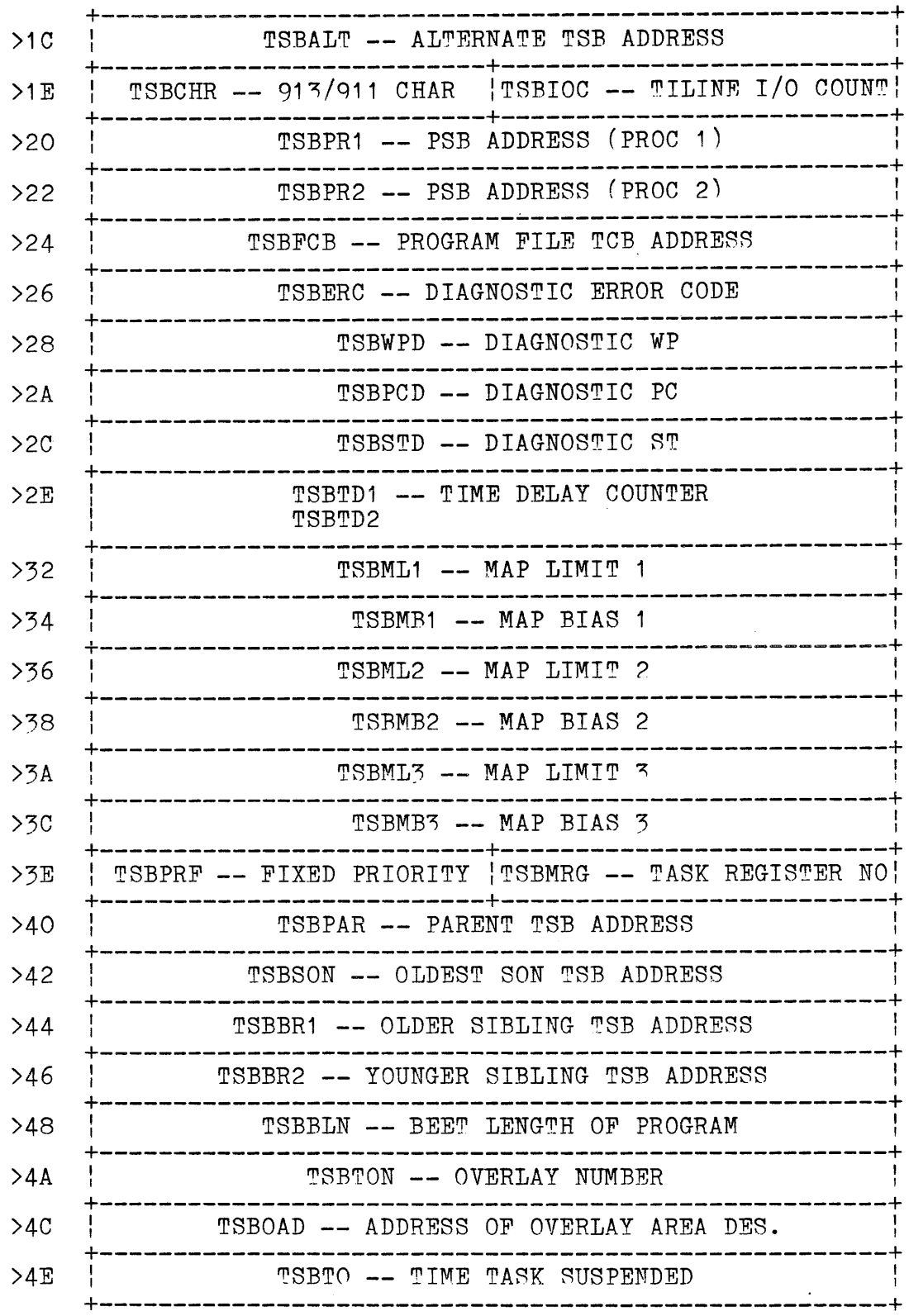


Figure 6-18 Task Status Block (TSB) -- Part 2 of 3

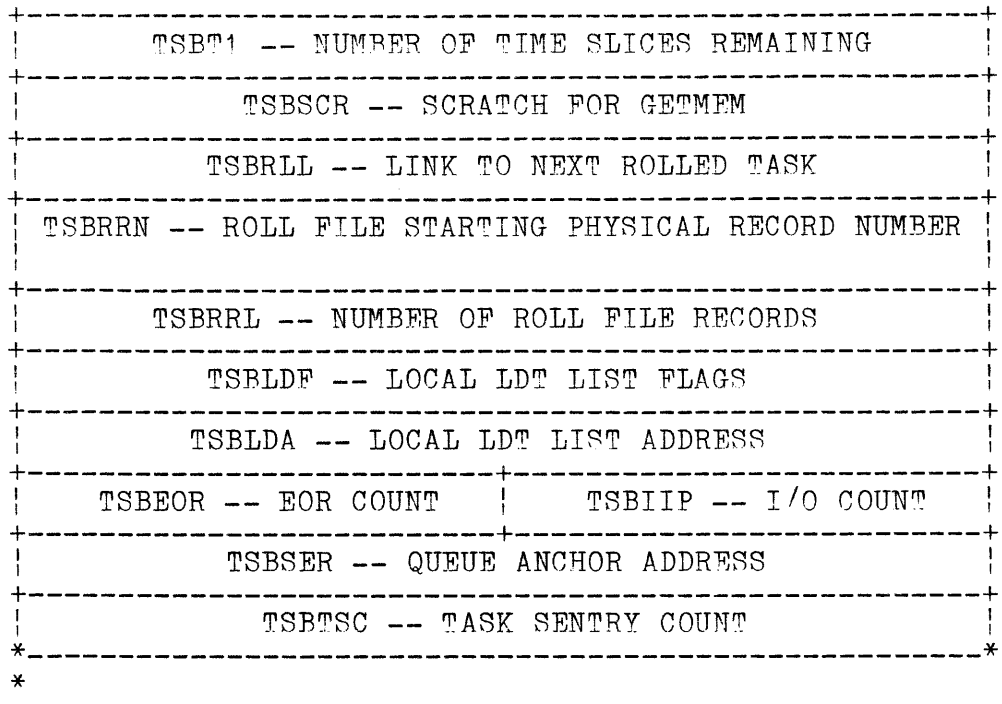


Figure 6-18 Task Status Block (TSB) -- Part 3 of 3

Hex. Byte	Field Name	Description
>00	TSBQL	Link to the next TSB on the queue, when this TSB is queued.
>02	TSBWP, TSBPC, TSBST	The saved context (workspace pointer, program counter and status register values) for the task. When the task is scheduled to execute, these saved values are used to begin execution.
>08	TSBPRI	Task priority (0, 1, 2, 3, or >81, >82, >83, ..., >FF) where >81 is real-time priority 1 and >FF is real-time priority 127.
>09	TSBSTA	Task state as shown in table 6-1.

Table 6-1 Task State Codes

Task State	Significance
00	Active task, priority level 0
01	Active task, priority level 1
02	Active task, priority level 2
03	Active task, priority level 3
04	Terminated task
05	Task in time delay
06	Suspended task
07	Currently executing task
08	Reserved
09	Task awaiting completion of I/O
0A	Task awaiting assignment of device for I/O
0B	Task awaiting disk file utility services
0C	Reserved
0D	Task awaiting file management services
0E	Task awaiting overlay loader services
0F	Task awaiting initial load
10	Reserved
11	Task awaiting disk management services
12	Task awaiting tape management services
13	Waiting on system overlay loader services
14	Waiting on task driven SVC processor
15	Task waiting on GETMEM request
16	Not used
17	Suspended for co-routine activation
18	Task waiting on termination task services
19	Task awaiting completion of any I/O
1A	Waiting on MM$FND door
1B	Task eligible for rollout when requested I/O is complete
1C	Task activated while roll in progress
1D	Suspended for initiate I/O threshold
1E	Suspended for locked directory
1F	Suspended for task management directory buffer
24	Task suspended for queue input
FF	Dummy task state

Hex. Byte	Field Name	Description
>0A	TSBFLG	First word of task flags. The flags are as follows:

Bit	Meaning When Set
0	System task (Hardware and software privilege)
1	Privileged task (Software)
2	Memory resident task
3	Take end action on error
4	Roll out candidate
5	Rolled out
6	Abort/terminate task
7	Activate call outstanding
8	Reactivate bidding task at termination
9	Serially reusable task
10	Task quieting in progress
11	Initial bid
12	Leave task alone; do not abort
13	Task is under control of alternate TSB
14	SCI flag for scanning TSB chain
15	Task is replicated image

>0C	TSBEAC	Transfer vector address.
>0E	TSBIID	Installed task ID.
>0F	TSBRID	Runtime ID assigned by system.
>10	TSBSMF	Address in the TSB of the saved map file register values (bytes >32->3D).
>12	TSBLNK	Link to the next TSB in the fixed list of TSBs. All TSBs in the system table area are linked onto this list when they are created. The list may be searched to find a task with a given runtime ID by the routine named TMTSCH (e.g., to kill the task).
>14	TSBKSB	Address of the KSB of the terminal with which this task is associated (i.e., the task was bid from the terminal).

>16 TSBFL2 Task flags as follows:

Bit	Meaning When Set
0	Task to be suspended next time it executes
1	Task is being controlled
2	SVC traps to be taken when specified
3	SVC switch: when 0, SVC traps are taken
4	Execution stopped by scheduler
5	Execution stopped by trapped SVC
6	Execution stopped by XOP 15,15 (breakpoint)
7	Dynamic priority management
8	Roll in progress
9	Task activated
10	Initiate followed by execute I/O
11	Extend time slice
12	End action available for task a(
13-15	Not used

>18 TSBAR1 The two parameters that may be passed to the task by the Bid SVC, and accessed by the task using the Get Bid Parameters SVC.

>1C TSBALT TSB address of the alternate task. The alternate task is executed in place of this task.

>1E TSBCHR 913/911 character.

>1F TSBIOC Number of outstanding TILINE I/O operations.

>20 TSBPR1 Address of the procedure status block (see paragraph 6.7.1) for attached procedure 1 (zero if none).

>22 TSBPR2 PSB address for procedure 2 (zero if none).

>24 TSBFCB Address of the FCB that represents the program file on which this task is installed.

>26 TSBERC Error code that describes the error that caused the task to terminate (used by termination task).

>28 TSBWPD, TSBPCD, TSBSTD The context (WP, PC, and ST registers) of the task at the time an error forced the task to terminate or a(take end action (used by the termination task).) These These values are returned on a Get End Action Status SVC.

>2E TSBTD1 Number of system time units remaining before this task will be reactivated from its time delayed state (32 bits.)

>32 TSBML1 The map register values to be used when this task executes.

>3E	TSBPRF	Map flags. Fixed priority of task.
>3F	TSBMRG	The offset into the saved map file that marks the limit register that maps the task segment (i.e., 0, 4, or 8).
>40	TSBPAR	TSB family tree pointers as described in Section 1.
>48	TSBBLN	Length of the entire progm (task and procedures) in beets (32-byte blocks).
>4A	TSBTON	The number of the system overlay in which this task was last executing (used for system tasks only).
>4C	TSBOAD	The address of the overlay area in which the above overlay was loaded (the overlay MUST be reloaded in the same place).
>4E	TSBTO	Number of time slices this task has been suspended.
>50	TSBT1	Number of time slices still allotted to this task as the minimum number of time slices it must receive before it can be forcibly rolled-out by an equal priority task.
>52	TSBSCR	Scratch used by the Get Memory SVC processor and the system overlay loader.
>54	TSBRLL	Link to the TSB or PSB that represents the next rolled segment. The TSB or PSB of each rolled task or procedure is linked onto a list of rolled segments. The list is kept in order by increasing roll file record number; i.e., segments that were written at the beginning of the roll file are at the beginning of the list. This linked list serves as a directory into the roll file, so that the various rolled segments can be retrieved for roll-in. Further roll information is kept in TSBs or PSBs.
>56	TSBRRN	Number of the physical record in the roll file that begins the rolled image of the task segment. At initial bid time, this is the program file record number.
>5A	TSBRRL	Number of roll file records occupied by the rolled task image. During initial bid, this is the length of the task in bytes.
>5C	TSBLDF	Task local LDT list flags: bit 0 is the LDT anchor.
>5E	TSBLDA	Pointer to the first task local LDT, or the station local LDT list anchor (if there are no task local LDTs).
>60	TSBEOR	Number of I/O end-of-records that need to be

		processed for this task. If this field is non-zero, the device driver task (DDT) is given the next time slice that would otherwise have been awarded to this task.
>61	TSBIIP	The number of I/O operations outstanding for this task.
>62	TSBSER	The address of the anchor for the queue served by this task (used only for queue servers).
>64	TSBTSC	The task sentry count.
>66	*	

6.8 PROCEDURE STATUS BLOCK (PSB)

Each procedure being accessed by a task within DX10 is represented by a procedure status block, just as tasks are represented by TSBs. When a task is bid, the bidder task, TM$BID, checks to see if the task has any attached procedures. If so, the bidder task scans the fixed list of PSBs anchored in the D$DATA module to see if the procedures are already in memory. If not, TM$BID builds a PSB for the procedures.

A PSB is built in the system table area and linked on the fixed list of PSBs. Figure 6-19 shows the format of a PSB.

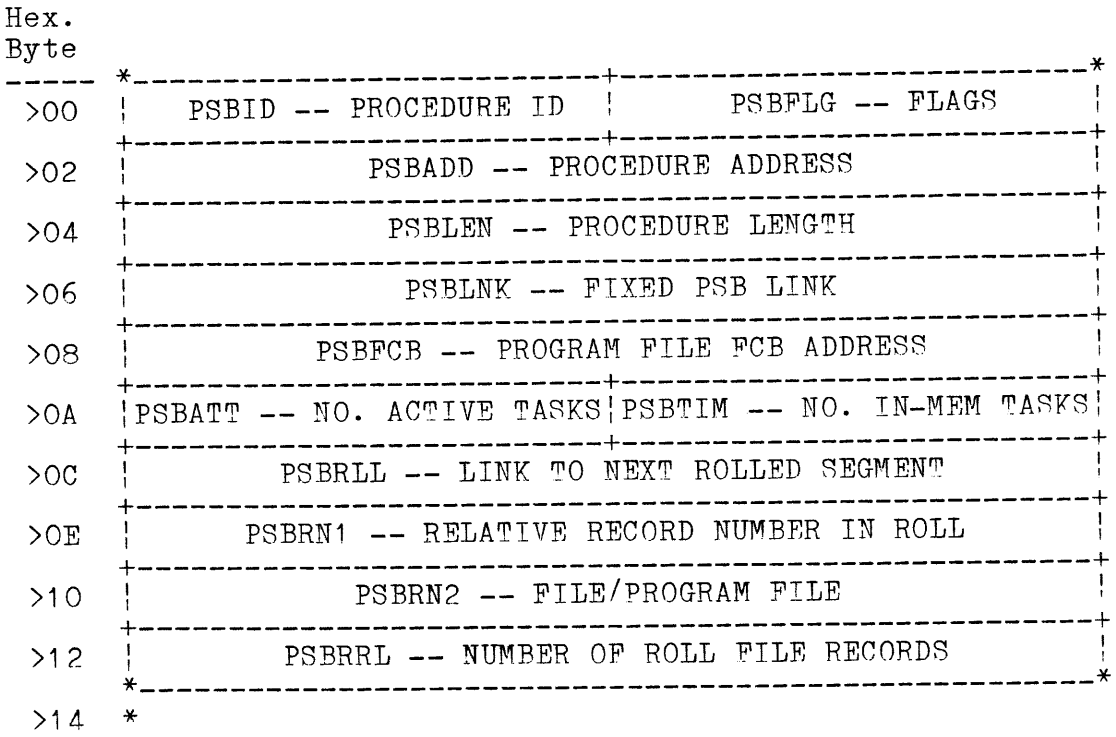


Figure 6-19 Procedure Status Block (PSB)

Hex. Byte	Field Name	Description
>00	PSBID	ID assigned to the procedure when it was installed on the program file.
>01	PSBFLG	Flags as follows:

Bit	Meaning When Set
0	Memory resident procedure
1	This is the initial bid of the procedure
2	Procedure is rolled out
3	Procedure roll is in progress
4	Writable control storage XOP
5	PROC is write protected
6	PROC is execute protected

>02	PSBADD	Address of the starting beet (32-byte block of memory) of the procedure.
>04	PSBLEN	Length of the procedure in beets.
>06	PSBLNK	Link to the next PSB in the fixed list of PSBs (zero if at end of list).

>08	PSBFCB	Address of the FCB for the program file on which the procedure is installed.
>0A	PSBATT	Number of active tasks that share this procedure.
>0B	PSBTIM	Number of active tasks with memory (not rolled) that share this procedure.
>0C	PSBRLL	Link to the next rolled segment (same as described for TSBs).
>0E	PSBRN1	Relative record number in roll.
>10	PSBRN2	File/program file.
>12	PSBRRL	Number of roll file records occupied by the rolled procedure. During the initial bid, this is the length of the procedure in bytes.
>14	*	

A PSB may be released to the system table area by the memory management routine named RELPSB. The PSB may only be released if the procedure has zero attached active tasks, in which case both the procedure memory and the PSB are released.

6.9 TIME ORDERED LIST (TOL)

As described in Section 1, all allocated blocks of memory (excluding the system table area) are linked on a doubly-linked, circular, time ordered list. This is done in order to support the least recently used algorithm used by DX10 memory management to select blocks of memory for rollout.

Blocks of memory that may be on the TOL are: task memory, procedure memory, and file management blocking (I/O) buffers (maximum of 30 buffers on TOL). Whenever a block of memory is accessed (i.e., executed if it is a task; read or written if it is a buffer), it is removed from its current position on the TOL, and relinked at the beginning of the list when the access is ended. An exception to this rule is procedure memory, which is not removed from the TOL when procedures are used. Procedure blocks, therefore, tend to go to the end of the list.

The overhead involved in maintaining the TOL consists of a TOL header, located in the D$DATA module, and an overhead beet (32-byte block) at the beginning of each allocated segment of memory. The overhead beet is created by either the task loader (for tasks and procedures) or buffer management (for blocking buffers). Figure 6-20 shows the format of a TOL beet. Note that an overhead beet is also used to maintain the linked list of free memory blocks (see paragraph 6.11).

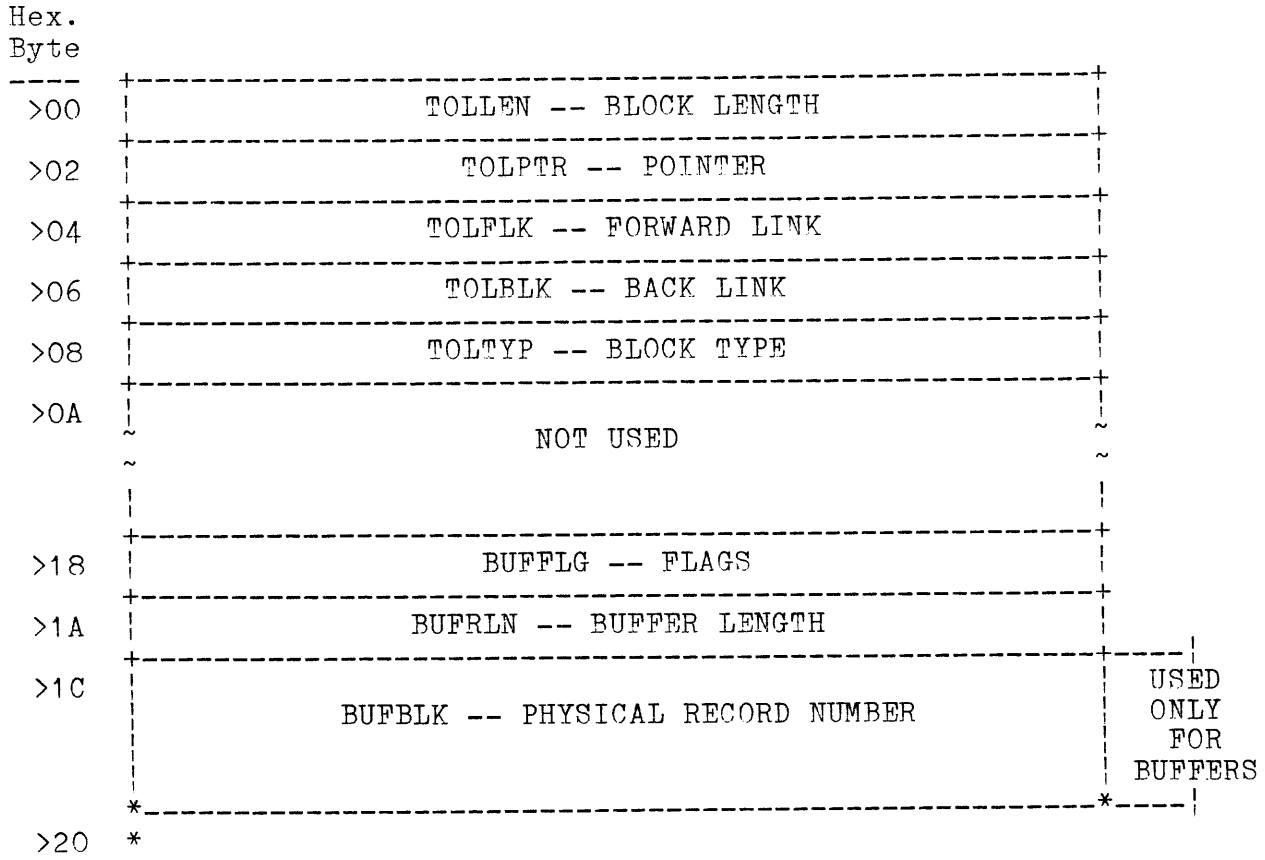


Figure 6-20 TOL Overhead Beet

Hex. Byte	Field Name	Description
>00	TOLLEN	Length, in beets, of the attached block of memory including this overhead beet.
>02	TOLPTR	Pointer, which varies depending on how this block is being used: Task -- Pointer to TSB Procedure -- Pointer to PSB Buffer -- Pointer to LDT Free block -- Pointer to next free block
>04	TOLFLK	Forward link to next oldest block.
>06	TOLBLK	Back link to next youngest block.

>08	TOLTYP	Block type as follows: 1 = task 2 = procedure 3 = free block 0 = blocking buffer -1 = list header
>0A	------	Not used.
>18	BUFFLG	Flags as follows:

Bit	Meaning When Set
0	Buffer is busy
1	Write this block
2	This is the memory resident buffer
3	Release this buffer immediately

>1A	BUFRLN	Length of buffer (excluding overhead beet) in bytes.
>1C	BUFBLK	Number of the file physical record that is buffered in this buffer.
>20	*	

6.10 SYSTEM LOG PARAMETER BLOCKS (SLPB)

Whenever a message is to be written to the system log, the message information is queued to the system log message queue in the form of a 12-byte system log parameter block (SLPB) plus extensions. The SLPB is created by different routines depending upon the source of the log message as follows:

Source	Creation Routine
Device Errors	SYSLQ, called by DDTEND, the end record and statistic messages processor.
Task Errors	SLPRQC, called by TM$DEN, the diagnostic task.
User Messages	SLSVC, the system log SVC processor.
Log Messages	SLMFOT, the system log formatter.
Memory Errors	Non-correctable errors in TM$INT; correctable errors in TM$SHD.

The SLPBs are queued for the system log formatting and output task, SLMFOT, which formats each SLPB and writes the message to the logging device and/or files. Figure 6-21 shows the format of the SLPB.

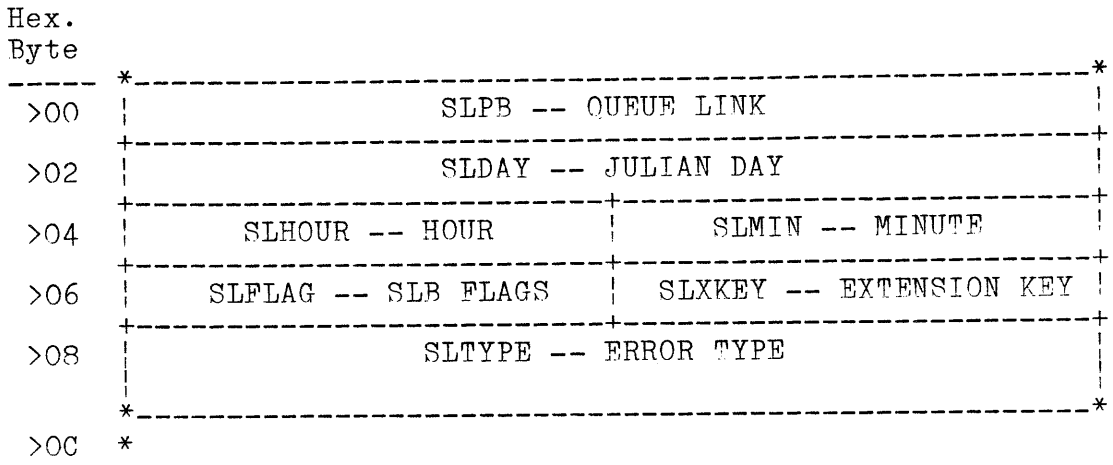


Figure 6-21 System Log Parameter Block

Hex. Byte	Field Name	Description
>00	SLPB	Link to the next block on the queue.
>02	SLDAY	Julian day.
>04	SLHOUR	Hour.
>05	SLMIN	Minute.
>06	SLFLAG	Flags as follows: Bit — Meaning When Set 0 — Subsequent messages have been lost 1-7 — Not used
>07	SLXKEY	Extension key as follows: 0 — Device extension with controller image 1 — User call extension 2 — Memory error extension 3 — Statistics extension 4 — Interrupt extension 6 — Task extension 8 — Cache memory extension 9 — Device extension with PRB
>08	SLTYPE	Error type (task, DSR, etc.)
>0C	*	

Depending upon the source of the message, various extension blocks are appended to the SLPB. The type of extension block to be appended is indicated by the extension key in the SLPB. The format of each of the extension blocks is shown in the following paragraphs.

6.10.1 DEVICE EXTENSION WITH CONTROLLER IMAGE (SLXKEY = 0).

Figure 6-22 shows the format of the SLPB Device Extension with Controller Image.

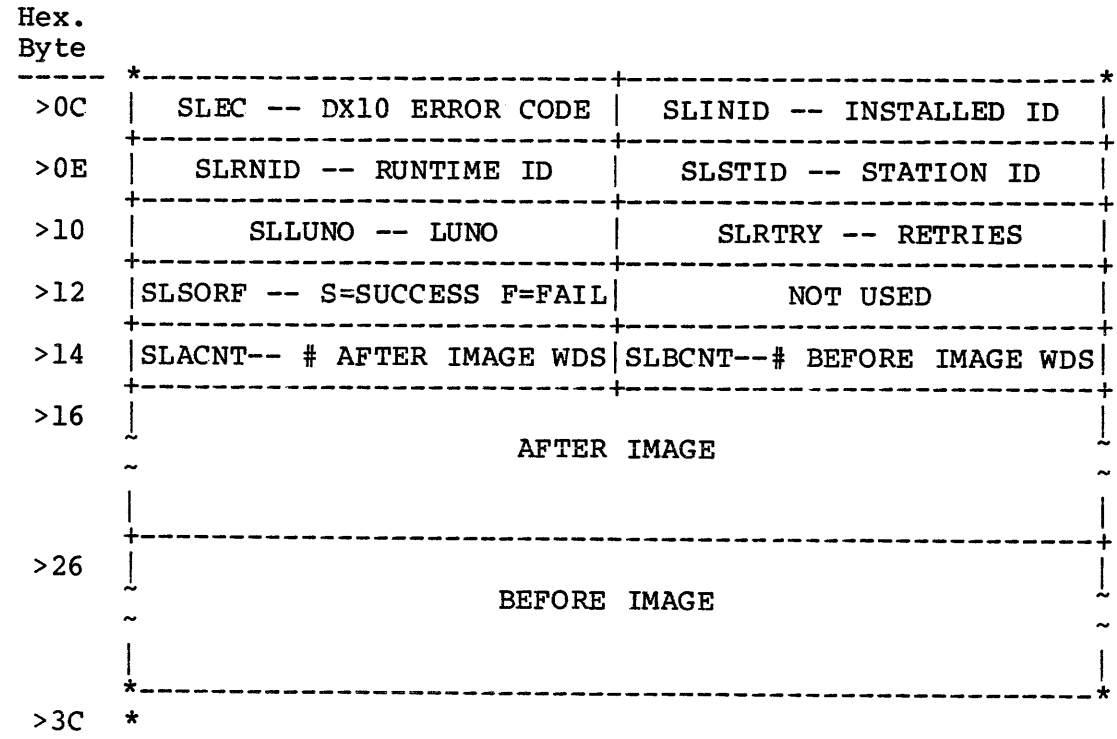


Figure 6-22 SLPB Device Extension With Controller

6.10.2 USER CALL EXTENSION TO SLPB (SLXKEY = 1).

Figure 6-23 shows the format of the User Call Extension to the SLPB.

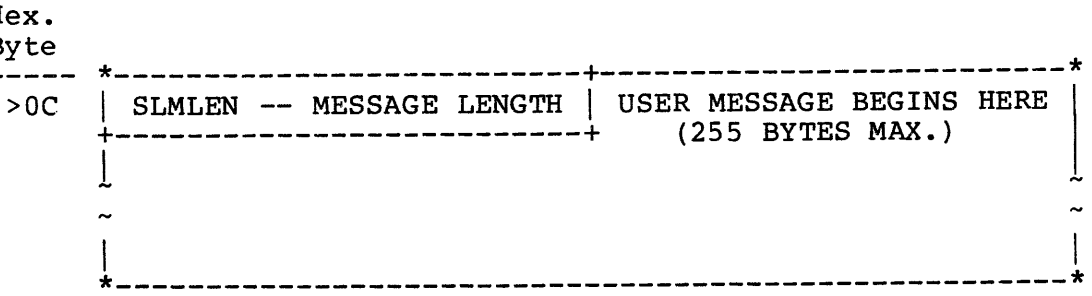


Figure 6-23 User Call Extension to SLPB

6.10.3 MEMORY ERROR EXTENSION TO SLPB (SLXKEY = 2).

The Memory Error Extension applies to 16K RAMs only. Figure 6-24 shows the format of the Memory Error Extension to the SLPB.

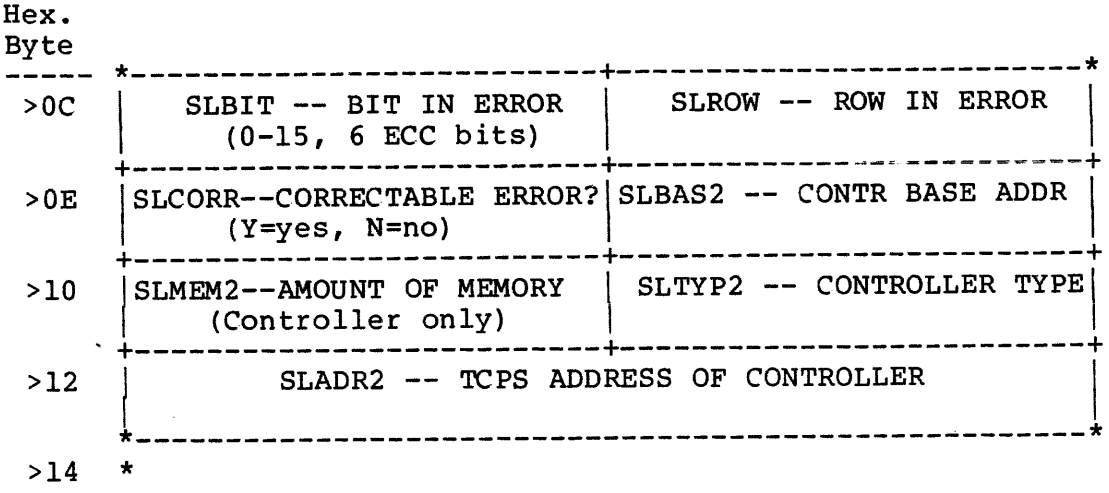


Figure 6-24 Memory Error Extension to SLPB

6.10.4 STATISTICS EXTENSION TO SLPB (SLXKEY = 3).

Figure 6-25 shows the format of the Statistics Extension to the SLPB.

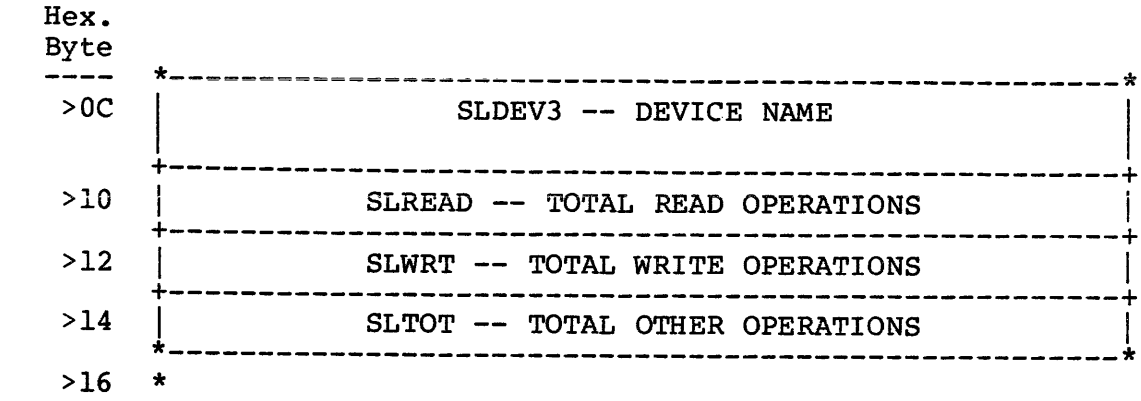


Figure 6-25 Statistics Extension to the SLPB

6.10.5 INTERRUPT EXTENSION TO SLPB (SLXKEY = 4).

Figure 6-26 shows the format of the Interrupt Extension to the SLPB.

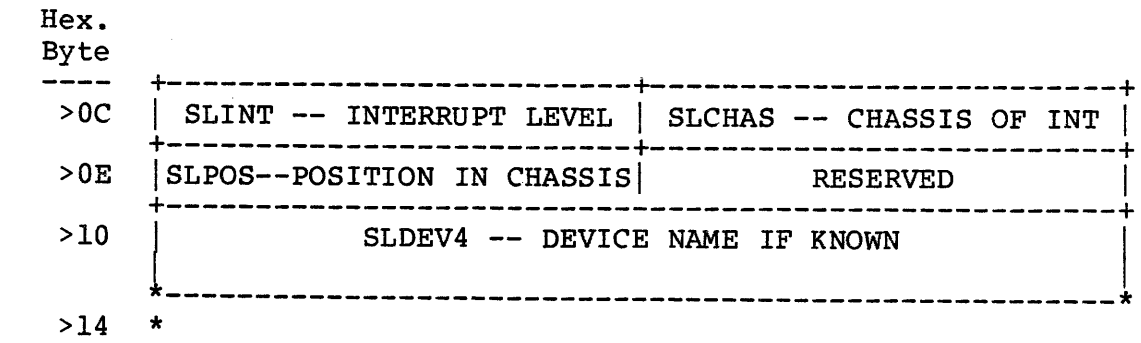


Figure 6-26 Interrupt Extension to the SLPB

6.10.6 TASK EXTENSION TO SLPB (SLXKEY = 6).

Figure 6-27 shows the format of the Task Extension to the SLPB.

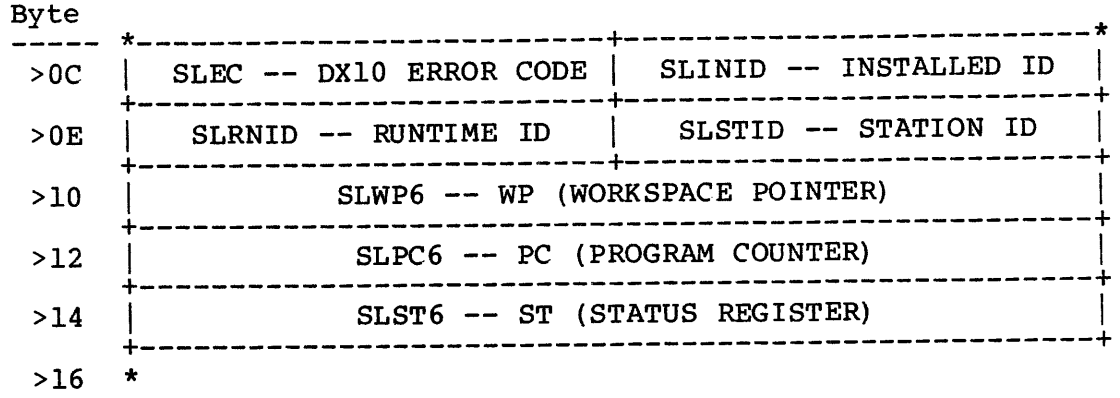


Figure 6-27 Task Extension to the SLPB

6.10.7 CACHE MEMORY EXTENSION TO SLPB (SLXKEY = 8).

Figure 6-28 shows the format of the Cache Memory Extension to the SLPB.

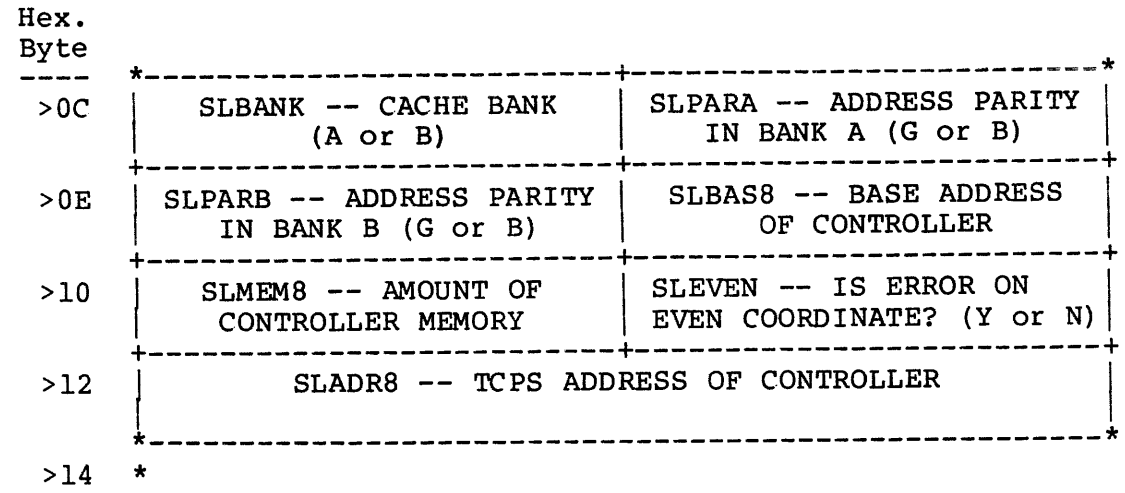


Figure 6-28 Cache Memory Extension to the SLPB

6.10.8 SLPB DEVICE EXTENSION WITH PRB (SLXKEY = 9).

Figure 6-28 shows the format of the SLPB Device Extension with PRB.

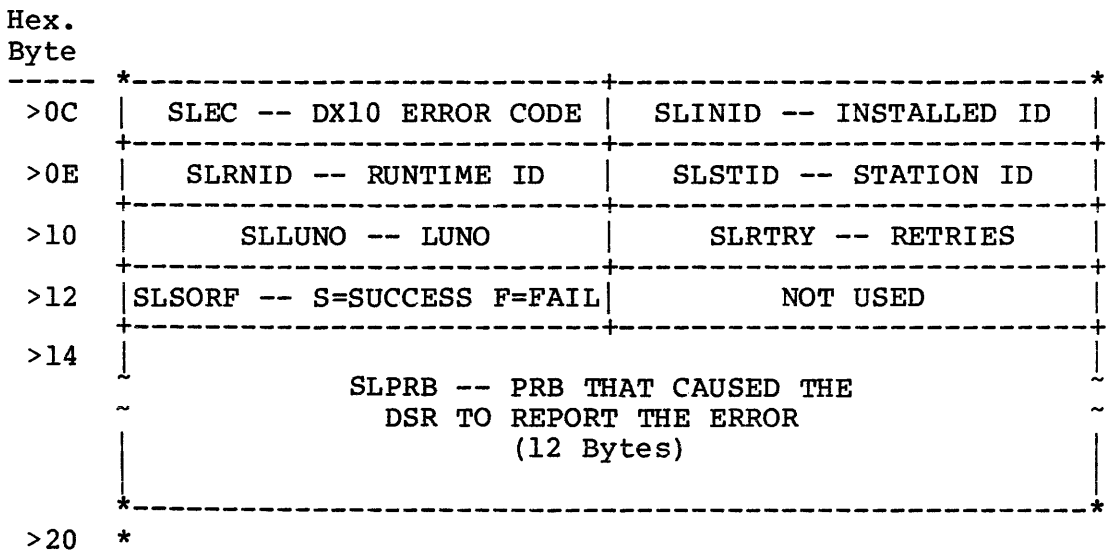


Figure 6-28 SLPB Device Extension with PRB

6.11 SYSTEM OVERLAY TABLE (OVT)

The system overlay table (OVT) is a vector table that contains the addresses of many system routine entry points, data structures, lists, and queue anchors. It is used by disk resident system tasks that are not linked with the DX10 memory resident code, but which must refer to and/or use information contained therein. The address of the vector table may be obtained via a special SVC, Get System Pointer Table Address. By using the table address as a base register value, a system task can refer to any of the addresses within the table by name. A template of the overlay table, showing the labels defined, follows.

```
*****************************************************************
*          OVERLAY TABLE          (OVT)                         *
*****************************************************************
0000                DORG 0
0000 0000  TSKLST   DATA 0          START OF TSB'S
0002 0000  UPS      DATA 0          ADDR SYSTEM TIME UNITS/SEC
0004 0000  PSBLST   DATA 0          START OF PSB'S
0006 0000  FIDMAP   DATA 0          ADDR FIXED TASK ID BIT MAP
0008 0000  PF$FCB   DATA 0          ADDR SYSTEM PROGRAM FILE FCB PT
000A 0000  ETSK     DATA 0          CURRENT EXECUTING TASK
000C 0000  BPT      DATA 0          BREAKPOINT TABLE
000E 0000  TSKSCH   DATA 0          TASK SEARCH UTILITY
0010 0000  SYSPF    DATA 0          ADDR OF PATHNAME OF SYSTEM PROG
0012       MM$RLM   BSS  4          TSB CLEAN UP
0016 FFFF  F$FLG    DATA -1         FUTIL/UNLOAD LOCKOUT
0018 0000  FUTPDT   DATA 0          FUTIL PDT CURRENTLY IN USE
001A 0000  PDTLST   DATA 0          START OF PDT'S
001C 0000  LDTLST   DATA 0          START OF LDT'S
001E       TM$QRM   BSS  4          REM. SPEC. ENT. FROM SPEC. QUEU
0022       TMBIDO   BSS  4          BID TASK ROUTINE
0026       TMTREE   BSS  4          BUILD TREE LINKAGE
002A 0000  RSTRID   DATA 0          USER RESTART ID
002C 0000  RSTRSW   DATA 0          CRT 'HELP' KEY DISABLE SWITCH
002E 0000  SLDATA   DATA 0          SYSTEM LOG DATA
0030       MM$FND   BSS  4          ALLOCATE USER MEMORY ROUTINE
0034       BM$MPB   BSS  4          MAP BUFFER INTO ADDRESS SPACE
0038 0000  TSKSTR   DATA 0          START OF NON-LINKED TSB'S
003A 0000  SYSTAB   DATA 0          START OF SYSTEM TABLE
003C 0000  TABSIZ   DATA 0          SIZE OF SYSTEM TABLE
003E 0000  AQPTRS   DATA 0          PTR TO ACTIVE QUEUES
0040 0000  TDL      DATA 0          ADDR OF TIME DELAY LIST ANCHOR
0042 0000  KBTAB    DATA 0          PTR TO 913 STATUS BLOCKS
0044       SLPBQC   BSS  4          SYSTEM LOG QUEUEING ROUTINE
0048 0000  SCIBMX   DATA 0          SCI BACKGROUND LIMIT
004A 0000  SCIFMX   DATA 0          SCI FOREGROUND LIMIT
004C 0000  MAPSHD   DATA 0          SCHEDULER MAP FILE POINTER
004E 0000  YEAR     DATA 0          BLOCK OF CURRENT DATE AND TIME
0050 0000  UAHEAD   DATA 0          ADDRESS OF MEM MGR HEADER
0052 0000  SAHEAD   DATA 0          ADDRESS OF SYS AREA HEADER
0054 0000  KTSKWP   DATA 0          SUBROUTINE TO KILL I/O - WP
0056 0000  KTSKPC   DATA 0          SUBROUTINE TO KILL I/O - PC
0058 0000  CURMAP   DATA 0          CURRENT MAP FILE POINTER
005A 0000  OADPTR   DATA 0          SYSTEM OVERLAY AREA
005C 0000  OVLYQ    DATA 0          LOAD OVERLAY QUEUE
005E       SO$LTO   BSS  4          LINK TO OVERLAY
0062       SO$BTO   BSS  4          BRANCH TO OVERLAY
0066       SO$RFO   BSS  4          RETURN FROM OVERLAY
006A 0000  ENDADD   DATA 0          LOAD ADR FOR FIRST USER TASK
006C 0000  ENDLIM   DATA 0          LIMIT REG FOR ENDADD
006E 0000  MEMSIZ   DATA 0          SIZE OF MEMORY IN BEETS
0070 0000  BASADJ   DATA 0          ADJUSTMENT VALUE FOR BIAS REG
0072 0000  TMTOL    DATA 0          START OF TIME ORDER LIST
0074 0000           DATA 0
0076       TM$DOR   BSS  4          SERIAL ACCESS DOOR LOCKING
007A       TM$OPN   BSS  4          SERIAL ACCESS DOOR UNLOCKING
```

```
007E        BM$FLS  BSS   4     BUFFER MGMT FLUSH ROUTINE
0082 0000   TM$EXT  DATA  0     INFINITE EXTEND TIME SLICE
0084        PUSH1   BSS   4     SAVE REG R1
0088        PUSH2   BSS   4     SAVE REGISTERS R1-R2
008C        PUSH3   BSS   4     SAVE REGISTERS R1-R3
0090        PUSH4   BSS   4     SAVE REGISTERS R1-R4
0094        PUSH5   BSS   4     SAVE REGISTERS R1-R5
0098        PUSH6   BSS   4     SAVE REGISTERS R1-R6
009C        PUSH7   BSS   4     SAVE REGISTERS R1-R7
00A0        PUSH8   BSS   4     SAVE REGISTERS R1-R8
00A4        PUSH9   BSS   4     SAVE REGISTERS R1-R9
00A8        POP0    BSS   4     EXIT ROUTINE
00AC        POP1    BSS   4     RESTORE R1
00B0        POP2    BSS   4     RESTORE REGISTERS R1-R2
00B4        POP3    BSS   4     RESTORE REGISTERS R1-R3
00B8        POP4    BSS   4     RESTORE REGISTERS R1-R4
00BC        POP5    BSS   4     RESTORE REGISTERS R1-R5
00C0        POP6    BSS   4     RESTORE REGISTERS R1-R6
00C4        POP7    BSS   4     RESTORE REGISTERS R1-R7
00C8        POP8    BSS   4     RESTORE REGISTERS R1-R8
00CC        POP9    BSS   4     RESTORE REGISTERS R1-R9
00D0        MM$RUA  BSS   4     RETURN USER AREA MEMORY
00D4        MM$GSA  BSS   4     GET SYSTEM AREA
00D8        MM$RSA  BSS   4     RETURN SYSTEM AREA
00DC        MM$GUA  BSS   4     GET USER AREA
00E0        SCRASH  BSS   4     SYSTEM CRASH ROUTINE
00E4        TMQUE   BSS   4     GENERAL QUEUEING ROUTINE
00E8        TMDQUE  BSS   4     GENERAL DEQUEUEING ROUTINE
00EC        TMAQUE  BSS   4     QUEUE ON ACTIVE QUEUE
00F0 0000   QHEAD   DATA  0     ADDR OF SVC QUEUES
00F2 0000   STUNIT  DATA  0     CLOCK TICKS / SYSTEM TIME UNIT
00F4 0000   FLG12   DATA  0     MACHINE FLAG 0=/10,-1=/12 (VAL)
00F6 0000   FLGWCS  DATA  0     WCS FLAG 0=NO; 1=YES - (VALUE)
00F8        RETRID  BSS   4     RETURN RUN TIME ID
            *
00FC        FMOPEN  BSS   4     FMT FILE OPEN PROCESSOR
0100        FMCLOS  BSS   4     FMT FILE CLOSE PROCESSOR
0104        FMWRIT  BSS   4     FMT FILE WRITE PROCESSOR
0108        WRTSEQ  BSS   4     SEQUENTIAL FILE WRITE
010C        CKWRIT  BSS   4     CHECK WRITE ACCESS
0110        CKLOCK  BSS   4     CHECK IF RECORD LOCKED
0114        MAPREC  BSS   4     TRANSLATE BLOCK # TO ADU #
0118        UPDFDR  BSS   4     UPDATE FILE DESCRIPTOR RECORD
011C        BM$MAP  BSS   4     MAP IN TASK BUFFER
0120        BM$RD   BSS   4     RETRIVE FILE BLOCK
0124        BM$REL  BSS   4     RELEASE FILE BLOCK
0128 0000   UAHADD  DATA  0     BEET ADDRESS OF UAHEAD
012A 0000   ENDDXL  DATA  0     LIMIT REG VALUE FOR DX-10
012C 0000   MEMSW   DATA  0     USER MEMORY SIZE SWITCH
012E 0000   DIOPDT  DATA  0     ADDRESS OF DISC PDT
0130 0000   TLDTSB  DATA  0     TSB FOR TASK LOADER
0132 0000           DATA  0     * * *  RESERVED   * * *
0134 0000   BIDTSB  DATA  0     TSB FOR BID TASK
0136 0000   TOLBET  DATA  0     FIRST BEET TIME ORDERED
0138 0000   OF$FCB  DATA  0     FCB ADDRESS OF OVERLAY
```

```
   013A 0000  RF$FCB  DATA 0          ROLL FILE FCB
   013C 0000  OF$LDT  DATA 0          OVERLAY FILE LDT
   013E 0000  PF1LDT  DATA 0          LDT FOR LUNO D
   0140 0000  PF2LDT  DATA 0          LDT FOR LUNO B
   0142 0000  PF3LDT  DATA 0          LDT FOR LUNO F
   0144 0000  RF$LDT  DATA 0          ROLL FILE LDT
   0146 0000  SCR$HD  DATA 0          HEAD ADR CRASH FILE
   0148 0000  SCR$TK  DATA 0          CYLINDER ADR CRASH
   014A 0000  SCR$SC  DATA 0          SECTOR ADR CRASH FILE
   014C 0000  SCR$DA  DATA 0          TILINE ADR CRASH FILE
   014E 0000  TOLLNK  DATA 0          TOL LINKAGE ROUTINE
   0150 0000  RIDMAP  DATA 0          ADR RUN TIME ID BIT MAP
   0152 0000  CMEMSZ  DATA 0          MEMORY TO BE CRASH DUMPED
   0154 0000  SCNPDT  DATA 0          DSR POWER UP ROUTINES
   0156 0000  SYSPFN  DATA 0          NAME OF SYSTEM PROGRAM FILE
   0158 0000  SCR$SL  DATA 0          CRASH FILE UNIT SELECT
   015A 0000  XY      DATA 0          TRAP INITIALIZATION TABLE
   015C 0000  UNLPDT  DATA 0          UNLOAD PDT CURRENTLY IN USE
   015E 0000  SCHDWS  DATA 0          SCHEDULER WORKSPACE
   0160 0000  SLCSUS  DATA 0          SCHEDULER ENTRY POINT
   0162 0000  BM$SIZ  DATA 0          LENGTH OF MEM RESIDENT BUFFER
   0164 0000  MMUMAX  DATA 0          MAXIMUM SIZE FREE USER AREA
   0166       FM$RDM  BSS  4          READ MULTIPLE
   016A       BM$MP2  BSS  4          CHECK MEMORY PROTECTION
   016E       TM$INC  BSS  4          INCREMENT TM$EXT BLWP VECTOR
   0172       TM$DEC  BSS  4          DECREMENT TM$EXT BLWP VECTOR
   0176       TM$CLR  BSS  4          CLEAR TM$EXT BLWP VECTOR
   017A 0000  S$$PAT  DATA 0          BEGINNING OF PATCH AREA
   017C 0000  CTRYCD  DATA 0          COUNTRY CODE
   017E 0000  TM$TO   DATA 0          POINTER TO TM$TO
   0180 0000  MM$PAK  DATA 0          MEMORY PACK REQUEST FLAG (NOT A
POIN
        0182  OVTSIZ  EQU  $
   0000               RORG
```

6.12 MEMORY MANAGEMENT LISTS

In addition to the time ordered list (TOL), which is a list of all allocated blocks of user memory (as opposed to system table area), two free memory lists are maintained by memory management.

One is a list of free blocks of system table area and is headed by the SAHEAD anchor in the D$DATA module. The list is singly linked and ordered by increasing address of the free block. Each free block contains the following overhead in the first four bytes:

Bytes	Description
0-1	Size of block in bytes
2-3	Link to the next block

When a block of system table area is allocated, the size of the block is stored in the word immediately preceding the first word of the allocated block (i.e., negative offset).

The other list contains free user memory blocks, and is headed by the UAHEAD anchor in the D$DATA module. The list is also singly linked and ordered by increasing address. Each block contains an overhead beet as described in paragraph 6.8 on the Time Ordered List (TOL).

6.13 STRUCTURE OF THE SEQUENTIAL FILE CREATED BY BACKUP DIRECTORY

By using the Backup Directory command, a directory can be backed up to a sequential file. The following are two examples of the structures of the sequential files created by backing up the directory of Figure 6-30 while using the control file of Figure 6-31.

Figure 6-32 shows the expanded structure of the backup of a program file. In Release 3.3, the block option was introduced. The block option causes information to be blocked in physical records of 9600 bytes (this may be altered in the Backup Directory PROC). The records are packed by preceding eachlogical record by a character count. An EOF (end of file) is represented by a zero count. The first record (the label) is not packed. The backup directory is still terminated by two physical EOFs when blocking is selected. Also, the tape label is extended from 7 words to 15 words.

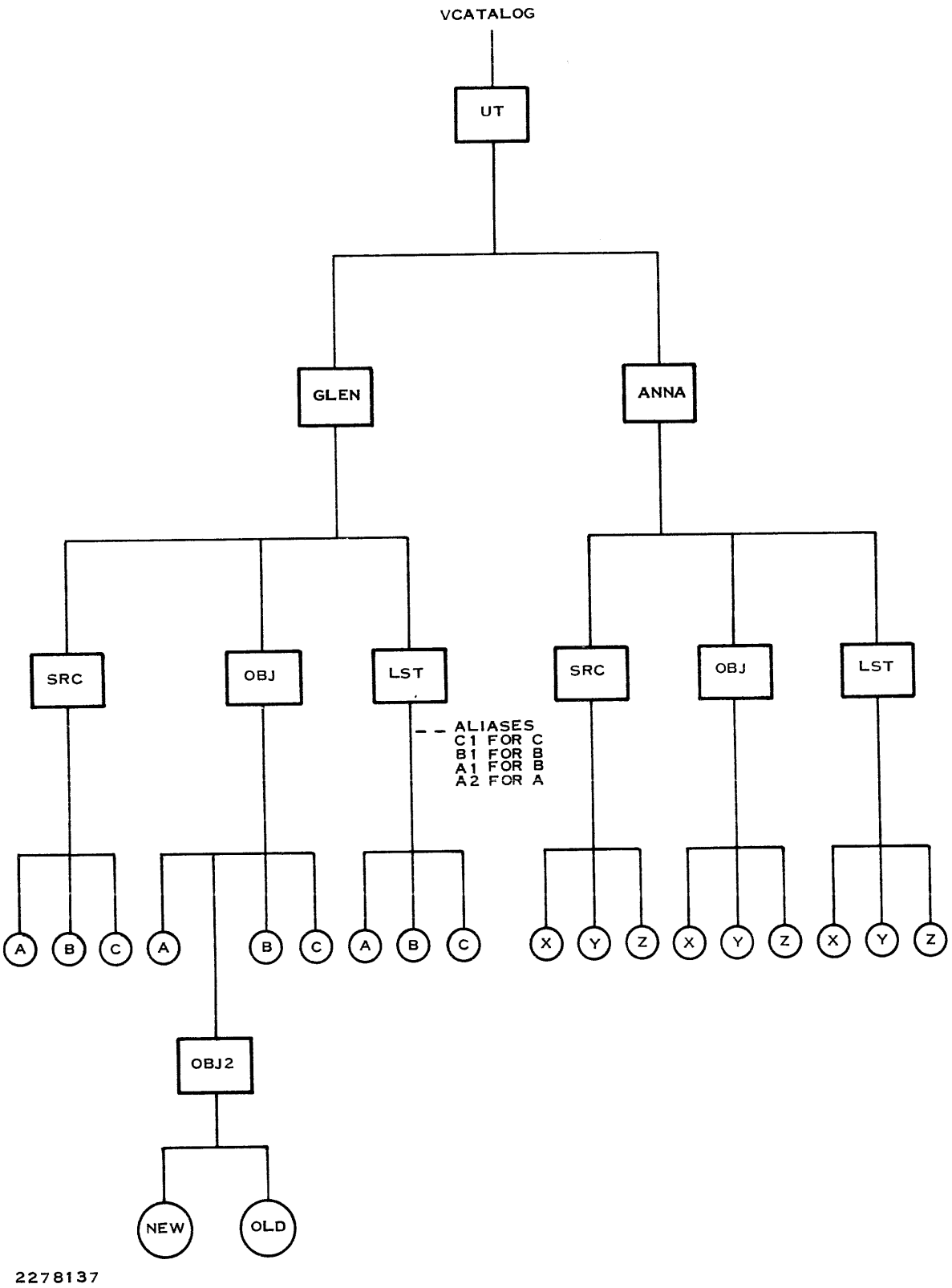


Figure 6-30 Directory To Be Backed Up

```
MOVE        .UT.GLEN.SRC,.SEQFILE
EXCLUDE     B
MOVE        .UT.GLEN.OBJ
EXCLUDE     B
MOVE        .UT.GLEN.LST
EXCLUDE     B
MOVE        .UT.ANNA
END
```

Figure 6-31 Control File

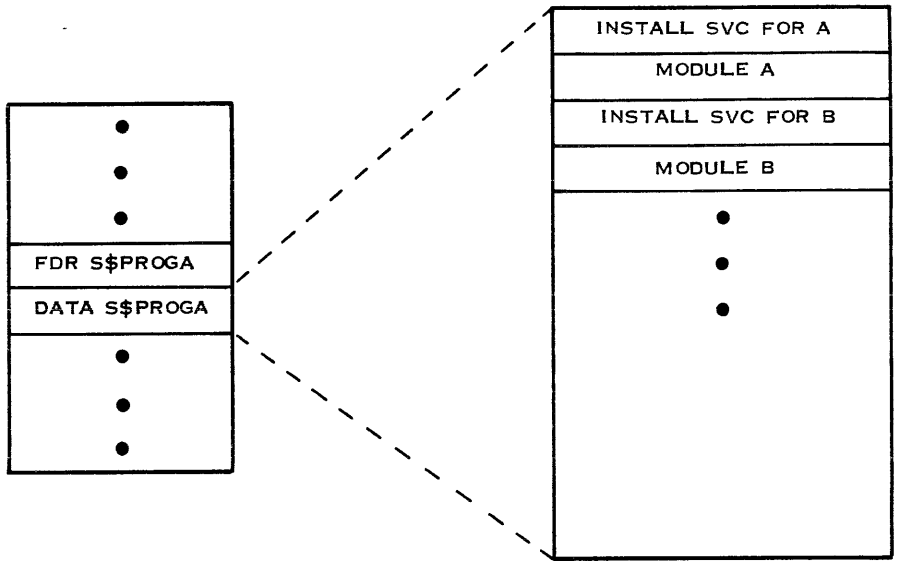


Figure 6-32 Expanded Structure for a Program File

6.13.1 BACKUP DIRECTORY WITH NOMULTI OPTION SELECTED.

Figure 6-32 represents the format of a tape created by the Backup Directory utility with the NOMULTI option specified. The MULTI/NOMULTI option was available on Release 3.2 and earlier. Later releases use the MULTI format exclusively. The only difference between the two formats is a header used by the MULTI format that begins every volume.

The first record written is the directory overhead record (DOR) of the first directory. Following the DOR are the FDRs and data records of the files under the directory. The end of the directory is noted by an EOD marker. The EOD marker is a record consisting of the following four bytes: EODb where b equals a blank space.

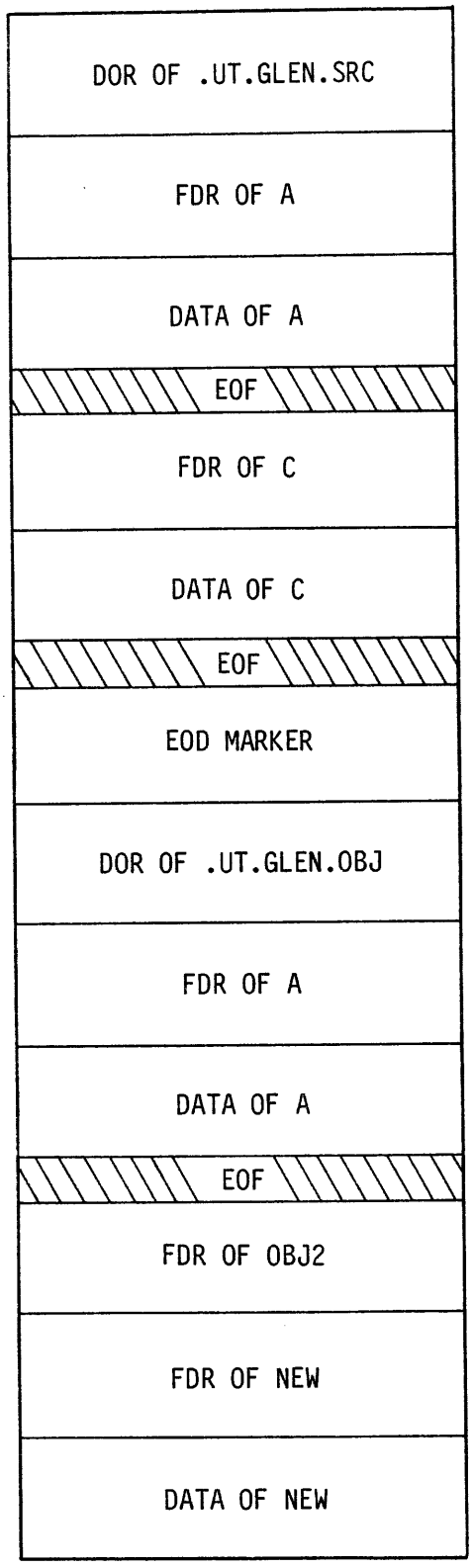
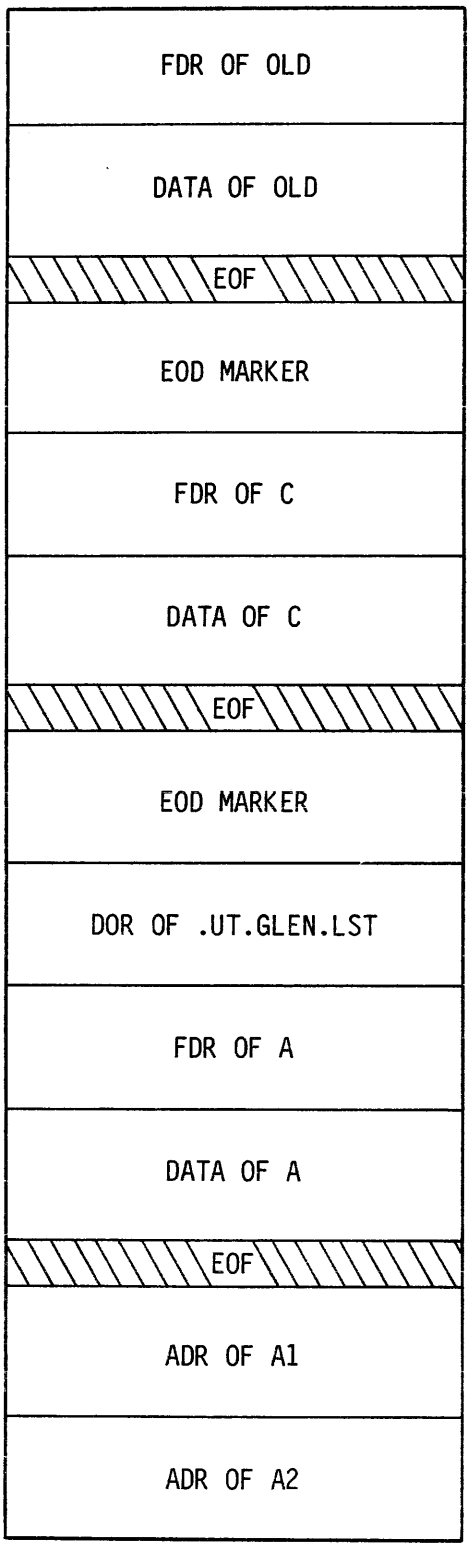
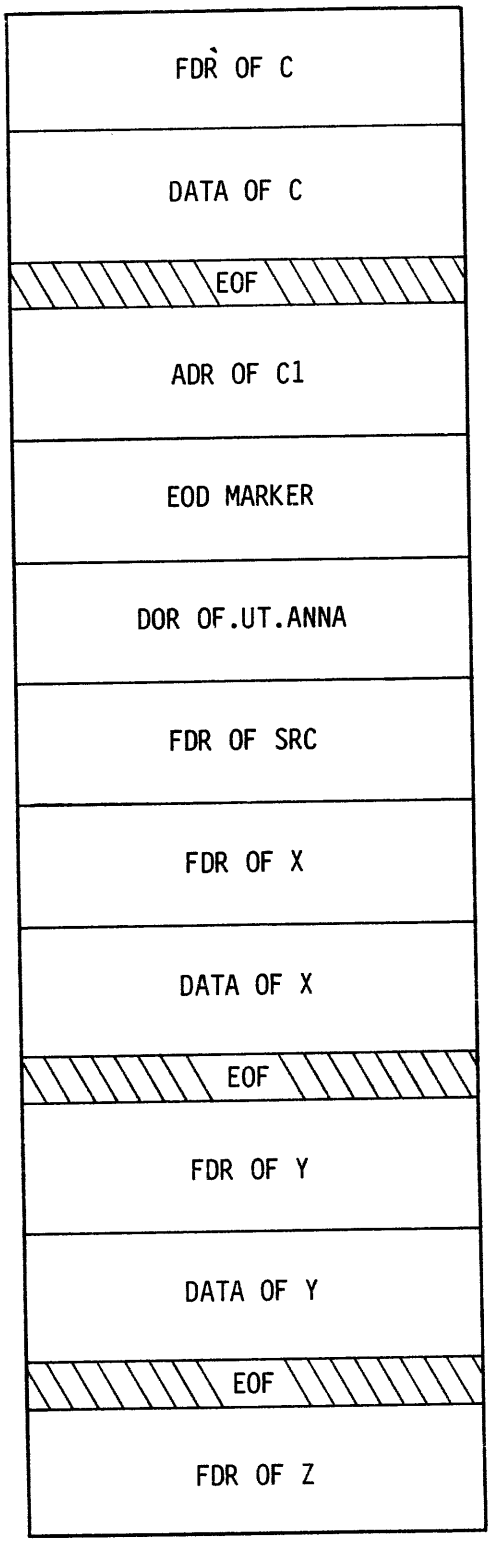


Figure 6-33 Structure of .SEQFILE (Sheet 1 of 5)



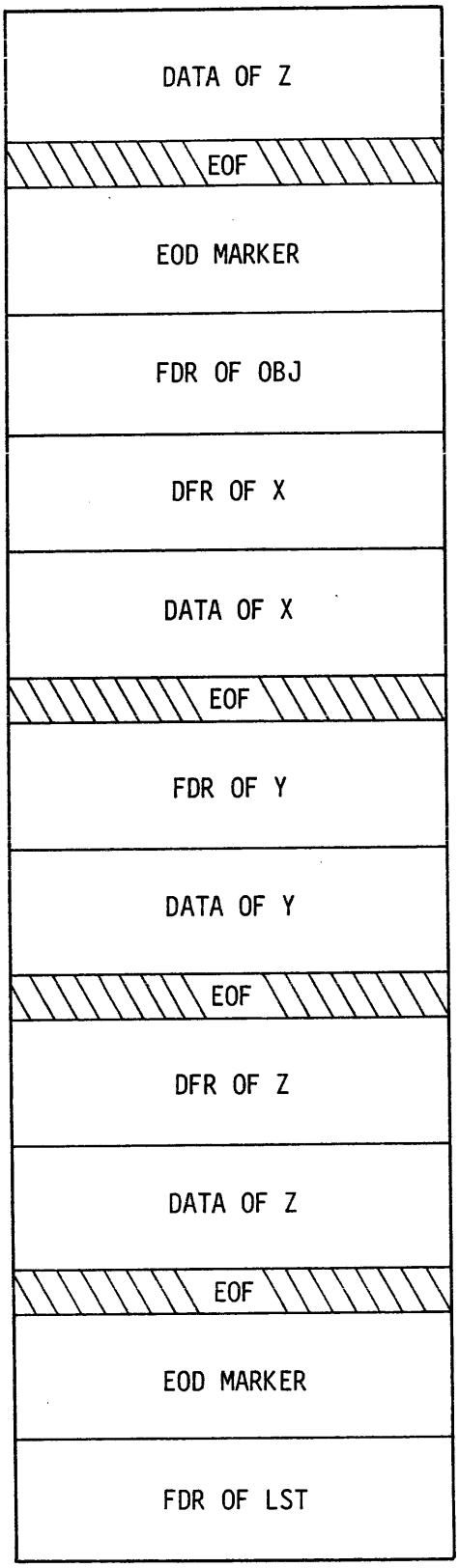
2278139 (2/5)

Figure 6-33 Structure of .SEQFILE (Sheet 2 of 5)



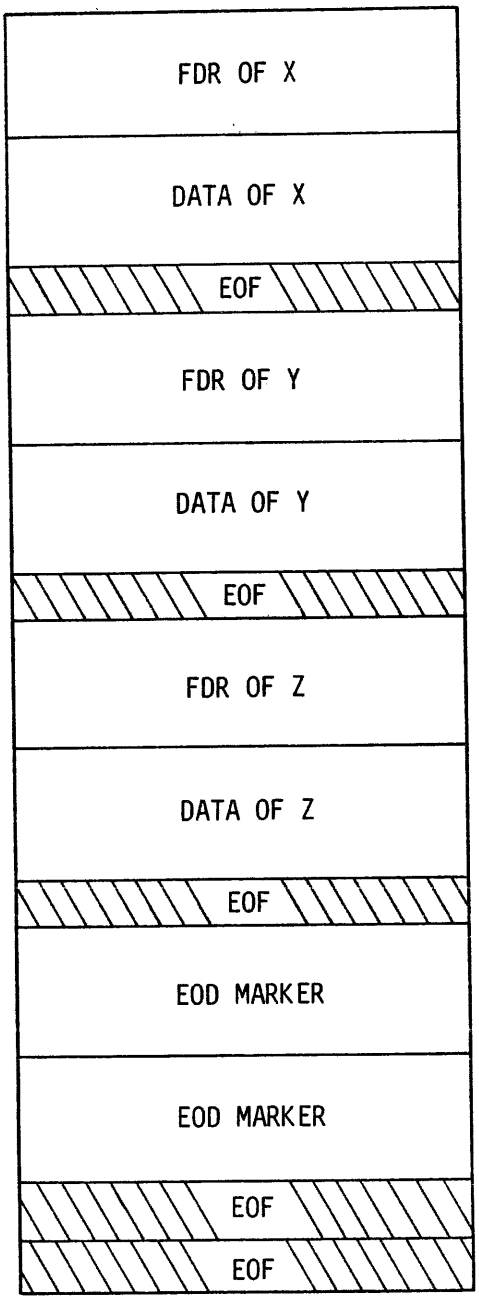
2278139 (3/5)

Figure 6-33 Structure of .SEQFILE (Sheet 3 of 5)



(. 2278139 (4/5)

Figure 6-33 Structure of .SEQFILE (Sheet 4 of 5)



2278139 (5/5)

NOTE

An EOD marker is a record with EODb in the first four bytes, where b equals a blank space.

Figure 6-33 Structure of .SEQFILE (Sheet 5 of 5)

6.13.2 Backup Directory with MULTI Option Specified

The MULTI option is specified when a directory is being backed up to magnetic tape and the directory spans more than one tape volume.

If the MULTI option is specified, the first record written to the sequential file is a header record (see tape 1, Figure 6-34). The header record consists of the following words:

Word	Contents
1-3	The ASCII characters **HDR*
4-7	Date and time as returned from SVC call
8	Tape volume number
9	Blocking factor
10-15	Reserved

After the header record, the file has the same structures as the NOMULTI file.

When the end of tape is encountered by Backup Directory, the record being written to tape is saved to be written to the next tape. After the tape has been changed, a header record is written to the new tape (see Tape 2, Figure 6-34). The header record has the same format as the header record of the first tape, except that the volume has been incremented by 1. The date and time written will be the same as that of the first tape. The record being written when the end of tape was encountered is then written and the backup continues.

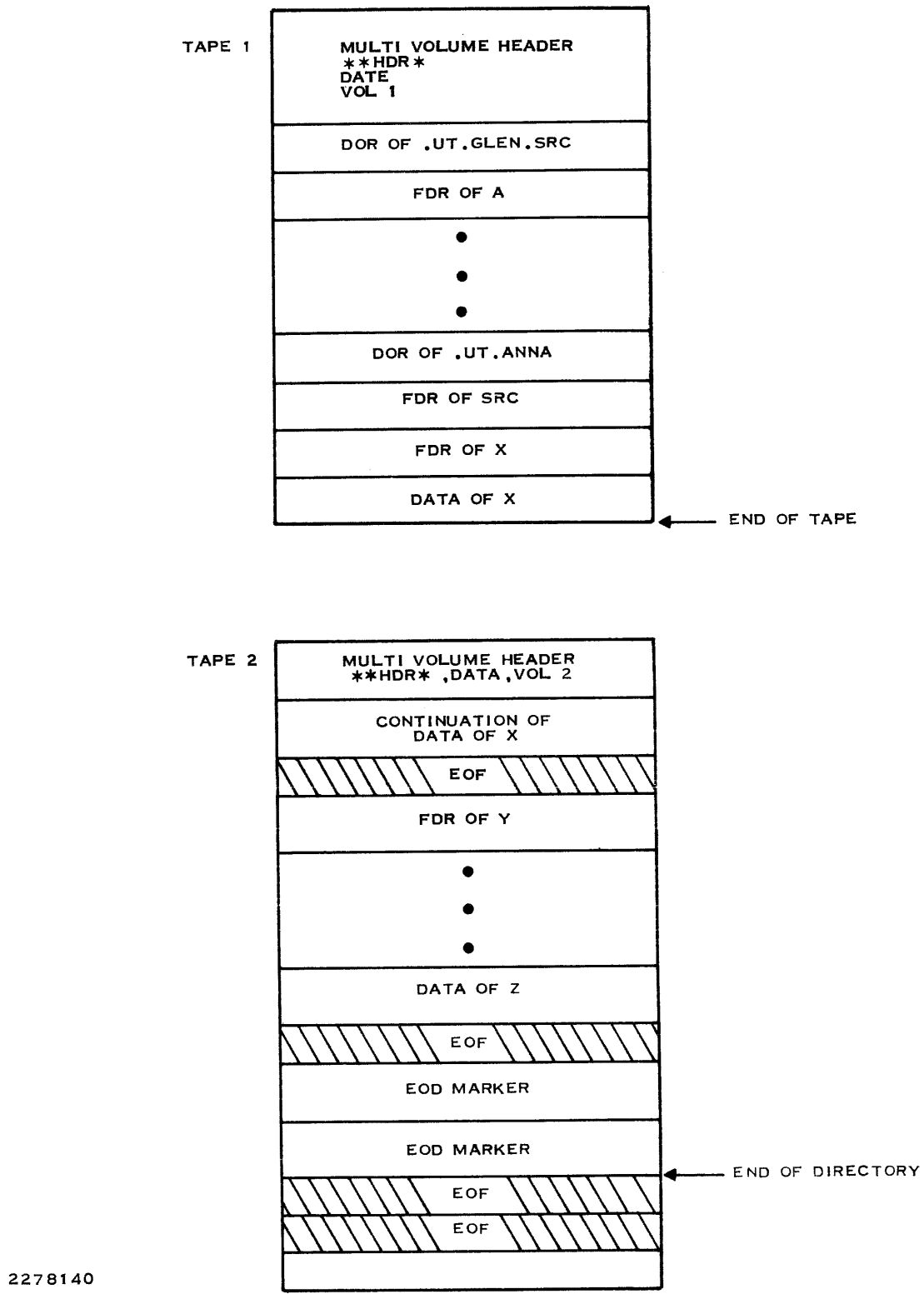


Figure 6-34 Back-up Directory Tape Format

SECTION 7

DX10 DATA BASE MODULES

7.1 GENERAL

Part of the memory resident DX10 kernel consists of the two DX10 data base modules, D$DATA and DXDAT2. The data base is split into two modules in order to separate sysgen dependent data from static data. Constant data is contained in DXDAT2. The D$DATA module is built by GEN990, and contains all of the sysgen dependent data. The following paragraphs describe the contents of the two data modules.

7.2 D$DATA

The D$DATA template is in file .DXMISC.SOURCE.D$DATA on the source disk. The first section of this module contains system constants (e.g., time slice, size of the system table area), list headers, and pointers (e.g., ETSK, which pointsto the task status block of the currently executing task), and a table of user defined SVCs.

The next section of D$DATA contains task status blocks for a file management task (FM$TSK) for each disk drive and the task bidder (TM$BID).

A keyboard status block table follows the TSBs. Each entry in the table points to the KSB for the station whose ID corresponds to the table index (i.e., first table entry points to KSB for ST01, third entry points to KSB for ST03, etc.).

The next section of the D$DATA module contains the PDT for every device defined in the system, including the KSBs for all terminals.

Following the PDTs are the global LDTs. There is a global LUNO (represented by a logical device table) assigned to every disk drive, and one assigned to ST01.

The remainder of the D$DATA module contains system log data, the system table area, the breakpoint table, the system overlay areas, the interrupt and XOP trap initialization table (written to locations >00->7F by the DX10 image loader), and the interrupt decoding module.

7.3 DXDAT2

The DXDAT2 source module is on the file named .DXMISC.SOURCE.DXDAT2 on the source disk. The constant data base module contains system SVC information, queue anchors for most system queues, and TSBs for some link-in system tasks.

The first section of this module is a table of constants which are often used by DX10 routines. The next section is a vector table of SVC processor addresses, and is used by the SVC interpreting code. Following the vector table is a table of the lengths of all supervisor call blocks. This table is used by this SVC intepreter, to find out how much call block needs to be buffered before passing control to the SVC processor.

The next section contains the anchors for most of the DX10 queues. Each anchor is of the form described in the data structures section.

Following the queue anchors is the list of SVC definition blocks which are used by the SVC unbuffering task, SVCCLN, to decide what information needs to be returned (unbuffered) to a task which has issued an SVC processed by a queue server. The list is terminated by a zero word.

The next section of DXDAT2 is a list of return field definitions, which is used by SVCCLN to determine which fields within a buffered supervisor call block need to be unbuffered into the calling task. This list is also ended by a zero word.

The next section contains task status blocks for the following linked-in system tasks:

* Task loader (TM$LDR)
* Overlay loader (TM$OVY)
* Buffer read/write task (TM$RWT)
* Device driver task (DDT)
* Disk manager (DM$TSK)
* SVC clean up task (SVCCLN)
* System overlay loader (SOVLDR).

The remainder of the DXDAT2 module contains the workspace used by the scheduler to update the time and date.

Section 8

COMMON SYSTEM ROUTINES

8.1 STACKING ROUTINES

Routines in DX10 use runtime stacks for passing parameters, storing registers and return information, and loading registers. In order to allow several routines that share a common workspace to use the same stack, R10 is reserved in DX10 routines as a pointer to the top of the stack (next available entry). Several stack handling routines, PUSHn and POPn, are used to store data on, and retrieve data from, a stack.

PUSHn is used to store registers R1-Rn (n </= 9) on the stack. PUSH automatically increments the stack pointer, R10, to point to the top of the stack. To call PUSH, execute a BL to @PUSHn. For example:

```
BL      @PUSH3      STORE R3, R2, R1
   or
BL      @PUSH9      STORE REGISTERS 9-1
```

PUSH stores the registers starting with the highest numbered register (i.e. a call to PUSH3 will store R3 first, then R2, and finally R1), and always clear R0.

Before calling PUSH, a routine must store its return address (usually the value of R11, if the routine was called via a BL instruction) on the stack. The example at the end of this description shows the general DX10 convention for using PUSH.

POPn is used to load registers R1-Rn from the top n words of the stack (n </= 9) and to return from a subroutine. POPn automatically decrements the stack pointer, R10, to point to the new top of the stack. Registers are loaded starting with R1. POP is entered by executing a B instruction to POPn.

POPO is a special entry point into POP, and is always executed at the end of any call to POP. POPO loads R11 with the top word of the stack. This should contain the address of the word following the instruction which branched to the routine now using POP. If R0 is zero (i.e. there are no errors being returned by the routine), control returns to the word following the address in R11. If R0 is non-zero (error condition) but the value of the word addressed in R11 is zero (no error return address), control also returns to the word following the address in R11. If R0 and the word addressed by R11 are both non-zero, control returns to the address contained in the word pointed to by R11.

A conventional call to a subroutine which uses PUSH and POP is:

```
               BL     @SUBR       Call to subroutine
               DATA   ERROR       Error return address
       NORML   EQU    $           Normal return point
```

The following example shows such a subroutine call:

```
       REF    PUSH5,POP5
       COPY   .SYSTEM.TABLES.ORS
*
* OFFSETS FOR REGISTER STACK
*
OR1    EQU    -2
OR2    EQU    -4
OR3    EQU    -6
OR4    EQU    -8
OR5    EQU    -10
OR6    EQU    -12
OR7    EQU    -14
OR8    EQU    -16
OR9    EQU    -18
STACK  BSS    30*2              CREATE A 30-WORD STACK
*       .
WS     BSS    16*2              CREATE A WORKSPACE
*       .
*THIS IS THE MAIN PROGRAM
MAIN   LI     R10,STACK         INITIALIZE STACK POINTER, R10
*       .
*       .
*       .
       BL     @SUBR             CALL SUBROUTINE
       DATA   ERROR             THIS IS THE ERROR RETURN ADDRESS
NORML  EQU    $                 THIS IS THE NORMAL RETURN
*       .
*       .
*       .
       RT                       RETURN TO THE CALLING PROGRAM
ERROR  EQU    $
*       .
*       .
*       .
*THIS IS THE SUBROUTINE
SUBR   MOV    R11,*R10+         STORE THE RETURN ADDRESS ON THE STACK
       BL     @PUSH5            STORE R1-R5, CLEAR R0
*       .
*       .
EXAMPL EQU    $                 NORMAL EXIT
       MOV    @RVAL,@OR2(R10)   STORE A RETURN VALUE IN THE STACKED R2
       B      @POP5             RESTORE R1-R5, RETURN TO MAIN
*
SUBERR EQU    $                 ERROR EXIT
       LI     R0,ERRCOD         PUT ERROR CODE IN R0
       B      @POP5             RESTORE R1-R5, TAKE ERROR RET IN MAIN
       END
```

Elements of a stack may be accessed without using PUSH and POP, by using offsets into the stack indexed by the stack pointer, R10, (as in EXAMPL above). The top word on the stack is at address @-2(R10), the next word is at @-4(R10), and so on down into the stack. Offsets for registers pushed onto a stack are given in .SYSTEM.TABLES.ORS.

8.2 QUEUING ROUTINES

There are six routines used within DX10 to add or delete entries from the various data queues. The routines are memory resident, and are located in the module TM$QUE. The routines are: TMAQUE, TMAQO, TMQUE, TMDQUE, TMTSBQ, and TM$QRM. These routines may be entered by memory resident system tasks by executing a BL to the name of the routine. Input register and stack requirements are given for each routine in the following paragraphs. Disk resident system tasks may access all of the routines except TMAQO via the system overlay table.

8.2.1 TMQUE

This is the general queueing routine. It places the specified data structure (any type) on the specified queue. The address of the structure to be queued is expected to be in R2. The address of the queue anchor should be in R1. TMQUE requires from 6 to 24 words of stack, according to the following conditions:

1. The queue has no dedicated server task--6 words.
2. The queue has a dedicated, memory resident server task--12 words.
3. The queue has dedicated, disk resident server task--24 words.

When the data structure is placed on the queue, TMQUE checks two conditions. If the queue has a dedicated server task, and the task is terminated, TMQUE bids the task (calls TMBIDO). If the queue server is in memory in state >24, it is activated directly. If the queue is a TSB queue (entries are TSBs), then the TSB that has just been queued is given the task state contained in the queue anchor (see paragraph 6.2 on queues).

8.2.2 TMAQUE

This routine puts the specified TSB on the active queue for that task's priority. TMAQUE uses six words of a stack. The routine expects R1 to contain the address of the TSB to be queued. The TSB flags are checked

to determine if the task has been allocated memory. If it does not have memory, TMAQUE calls TMAQU to put the task on the waiting on memory queue, TMWOM. If the task already has memory, TMAQUE checks the TSB priority, and calls TMQUE to put the task on the active queue for that priority.

8.2.3 TMAQO

This routine puts the specified TSB on the active queue at position one for priority one tasks. The task loader and the system overlay loader use TMAQO so that tasks which have just been loaded will get a good chance of executing at least once before being rolled.

TMAQO uses six words of stack. The routine expects R1 to contain the TSB address of the task to be queued.

8.2.4 TMTSBQ

This routine queues the specified TSB on the specified queue. TMTSBQ is actually a second entry point to TMQUE, and is therefore the same routine.

8.2.5 TMDQUE

This is the general dequeuing routine. It is used to remove an entry from the head of the specified queue. The routine uses one word of a stack. The address of the dequeued anchor should be in R1. TMDQUE will return the address of the dequeued data structure in R12, or an error message in R0. The only error code is 1, which means the queue is empty.

8.2.6 TM$QRM

This routine is used to remove any specified entry from the specified queue. It requires one word of a stack. The address of the queue anchor should be in R1, and the address of the structure to be removed from the queue should be in R2. TM$QRM searches through the queue for an entry with the address specified in R2. If no such entry is in the queue, an error code of 1 is returned in R0. Otherwise, the structure is removed from the queue.

SECTION 9

DESCRIPTION OF DX10 ROUTINES

9.1 GENERAL

This section breaks the major DX10 routines into functional categories (e.g., task management, file management) and describes each routine briefly. Several tables are included which show how the routine source may be found on a DX10 source disk.

9.2 SVC PROCESSING

SVC processing includes individual SVC processing routines and several overhead routines that are involved in decoding SVCs and buffering and unbuffering supervisor call blocks. Tables 9-1 and 9-2 show the routines that process SVCs.

Table 9-1 SVC Overhead Routines

Routine	Source Module Pathname	Description
SVCINT	.DXMISC.SOURCE.SVCINT	Interprets XOP 15 by accessing user call block and SVC code. Looks up SVC processor address in SCTAB (DXDAT2 module), transfers control to that address.
SVCBUF	.DXMISC.SOURCE.SVCBUF	Buffers user call blocks for SVCs processed by queue serving tasks into system table area. Calls TMQUE to queue the buffered call block and bid the queue server.
SVCFND	.DXMISC.SOURCE.SVCBUF	Looks up the SVC definition block in the DXDAT2 table, SVCDEF. Called by SVCBUF.
SVUBUF	.DXMISC.SOURCE.SVCBUF	Buffers the user's call block and any expansion blocks (as defined in the definition block retrieved by SVCFND) into system table area.
SVCCLN	.DXMISC.SOURCE.SVCCLN	Unbuffers the buffered call block after a queue serving SVC processor has finished. SVCCLN may have to cause the task to be rolled-in. The task is reactivated after the unbuffering. The buffer is released to the system table area.

Table 9-2 SVC Processors -- Part 1 of 2

SVC Code	SVC Title	XOP/Queue Server	Processor Name	Source Module Pathname
00	I/O	X	DXIOS	.DXMISC.SOURCE.DXIOS
01	Wait for I/O	X	WAITIO	.DXMISC.SOURCE.WAITIO
02	Time delay	X	TDLY	.TSKMGR.SOURCE.TM$FUN
03	Date and time	X	DTTIM	.TSKMGR.SOURCE.TM$FUN
04	End of task	Q	ENDTSK	.TSKMGR.SOURCE.TM$FUN
05	Bid task	Q	TM$SBD	.SYSTSK.SOURCE.TM$SBD
06	Unconditional wait	X	UNCDWT	.TSKMGR.SOURCE.TM$FUN
07	Activate suspended task	X	ACTTSK	.TSKMGR.SOURCE.TM$FUN
09	Do not suspend	X	HOTSK	.TSKMGR.SOURCE.TM$FUN
0A	Convert binary to decimal	X	CBDA	.DXMISC.SOURCE.CNVRSN
0B	Convert decimal to binary	X	CDAB	.DXMISC.SOURCE.CNVRSN
0C	Convert binary to hexadecimal	X	CBHA	.DXMISC.SOURCE.CNVRSN
0D	Convert hexadecimal to binary	X	CHAB	.DXMISC.SOURCE.CNVRSN
0E	Activate time delay task	X	ATDLYT	.TSKMGR.SOURCE.TM$FUN
0F	Abort I/O (LUNO)	X	ABTIOX	.DXMISC.SOURCE.ABTIOX
10	Get common data address	X	GETCOM	.TSKMGR.SOURCE.TM$CMN
11	Change priority	X	CHGPRI	.TSKMGR.SOURCE.TM$FUN
12	Get memory	Q	MM$GTM	.MEMMGR.SOURCE.MM$SVC
13	Release memory	X	MM$RTM	.MEMMGR.SOURCE.MM$SVC
14	Load overlay	Q	TM$OVY	.TSKMGR.SOURCE.TM$OVY
15	Disk file utility	Q	FUTIL	.FUTIL.SOURCE.FU$
16	End of program	Q	ENDPGM	.TSKMGR.SOURCE.TM$FUN
17	Get parameters	X	GETPRM	.TSKMGR.SOURCE.TM$FUN
1B	Return common data	X	RETCOM	.TSKMGR.SOURCE.TM$CMN
1C	Put data	X	PUTDAT	.TSKMGR.SOURCE.TM$IQ
1D	Get data	X	GETDAT	.TSKMGR.SOURCE.TM$IQ
1F	Scheduled bid task	Q	TM$SBD	.SYSTSK.SOURCE.TM$SBD
20	Install disk volume	Q	INSTAL	.SYSTSK.SOURCE.INSTAL
21	System log SVC	Q	SLSVC	.SYSTSK.SOURCE.SYSLGY
22	Disk manager	Q	DM$TSK	.DSCMGR.SOURCE.DM$TSK
24	Suspend awaiting queue input	X	SUSPQI	.TSKMGR.SOURCE.TM$FUN
25	Install task	Q	PF$LIN	(SYSTEM TASK)
26	Install procedure	Q	PF$LIN	
27	Install overlay	Q	PF$LIN	
28	Delete task	Q	PF$LDE	(SYSTEM TASK)
29	Delete procedure	Q	PF$LDE	
2A	Delete overlay	Q	PF$LDE	
2B	Execute task	Q	EXCTSK	.TSKMGR.SOURCE.TM$FUN
2C	Read/write TSB	X	TSMWRT	.TSKMGR.SOURCE.TM$RWT
2D	Read/write task	Q	TM$WRT	.TSKMGR.SOURCE.TM$RWT
2E	Self identification	X	TM$SID	.TSKMGR.SOURCE.TM$FUN

Table 9-2 SVC Processors -- Part 2 of 2

SVC Code	SVC Title	XOP/Queue Server	Processor Name	Source Module Pathname
2F	End action status	X	TM$EAS	.TSKMGR.SOURCE.TM$EAS
30	Get event character	X	GTEVNT	.DXIO.SOURCE.GETEVT
31	Map program name to ID	Q	PF$LMN	(SYSTEM TASK)
32	Get overlay table address	X	GTOVYT	.TSKMGR.SOURCE.TM$FUN
33	Kill task SVC	Q	SVCKIL	.SYSTSK.SOURCE.SVCKIL
34	Unload disk volume	Q	INSTAL	.SYSTSK.SOURCE.INSTAL
35	Poll status of task in terminal task set	X	TM$ST	.TSKMGR.SOURCE.TM$FUN
36	Wait on multiple initiate I/Os	X	WANYIO	.DXMISC.SOURCE.WAITIO
37	Assign space on program file	Q	PF$LAS	(SYSTEM TASK)
38	Initialize disk volume	Q	INVOL	.SYSTSK.SOURCE.INVOL
39	Get event character	X	GTEVTL	.DIO.SOURCE.GETEVT
3B	Initialize date and time	X	SDTIM	.TSKMGR.SOURCE.TM$DTM
3E	Reset end action	X	RSTEAC	.TSKMGR.SOURCE.TM$FUN
3F	Retrieve system data	X		.TSKMGR.SOURCE.TM$FUN

9.3 BID TASK SUPERVISOR CALL - CODE >05

The Bid Task supervisor call is included in DX10 3.4 for compatibility with DX10 2.X releases. Because this SVC may be deleted in later releases, use of the Bid Task SVC is not recommended. Documentation is included in this System Design Document for informative purposes.

The Bid Task SVC can only be used for tasks that have been installed on the single system program file and that were designated as being non-replicatable. The call is tranmitted to an Execute Task (>2B) SVC by the operating system task TMSBD. The task is bid with no associated station, and the calling task is not suspended.

The call block for the Bid Task supervisor call is eight bytes in length and must be aligned on a word boundary. The entries in the supervisor call block are defined as follows:

Byte 0 - Contains the code for the Bid Task supervisor call and must be >05.
Byte 1 - This byte is used by the system to return an error code, if necessary.
Byte 2 - Contains the installed ID on the system program file of the task to be executed.
Byte 3 - Reserved.

Bytes 4-7 - These bytes are used to pass user specified parameters to the called task. The called task must issue a Get Parameters supervisor call to obtain the parameters.

The following example of a Bid Task supervisor call specifies that task >5E be loaded from the system program file. The ASCII representation of the characters HELP are passed to the called task.

```
      EVEN          FORCE ALIGNMENT ON A WORD BOUNDARY
BIDT  BYTE >05      CODE FOR BID TASK
ERRC  BYTE >00      SET BYTE ONE TO ZERO
ID    BYTE >5E      INSTALLED ID OF TASK TO BE EXECUTED
      BYTE >00      RESERVED (SET TO ZERO)
PARMS TEXT 'HELP'   FOUR BYTES PASSED TO CALLED TASK
```

Within the procedure portion of the calling task, the following statement is used to initiate the Bid Task supervisor call:

```
XOP  BIDT,15
```

Error codes returned in Byte 1 of the supervisor call block are as follows:

Error Code	MEANING
>04	Signifies successful completion, for compatibility with previous releases.
>FF	Either: specified task is not on the system program file; the specified task is on the system program file and is replicatale; or an error occurred when the translated Execute Task SVC was performed.
Other	A task state is returned. This is the task state of the called task if it is already in execution. Or if a currently running task has allocated the run ID corresponding to the called tasks installed ID, that tasks state is returned as an error code.

9.4 TASK MANAGER

The main routines involved in task management are the scheduler, task loader, overlay loader, system overlay loader, and the task managing SVCs. The task loader also works closely with memory management routines, which are discussed in the next paragraph.

Table 9-3 shows the major task management routines.

Table 9-3 Task Management Routines -- Part 1 of 4

Routine	Source Module Pathname	Description
TM$SHD	.TSKMGR.SOURCE.TM$SHD	Task scheduler. Updates time and date, and delayed tasks, bids SCI, and selects the next task to execute.
TM$LDR	.TSKMGR.SOURCE.TM$LDR	Task loader. Loads tasks and procedures, rolls out quieted tasks and tasks that have issued Get Memory SVCs.
TM$OVY	.TSKMGR.SOURCE.TM$OVY	Overlay loader. Loads overlays requested by tasks. Requests are queued for the overlay loader, which loads the overlay and calls TMAQUE to put the task on an active queue. TM$OVY calls file management routines to read the overlay from disk.
SOVLDR	.DXMISC.SOURCE.SOVLDR	System overlay loader. Serves the queue SOVYQ. TSBs of tasks that have called system overlays are queued on SOVYQ. SOVLDR loads the correct overlay and reactivates the task, calling TMAQO.
SO$LTO	.DXMISC.SOURCE.SO$CPR	Link to system overlay. This routine is called by a task or overlay to link to a system overlay. If the desired overlay is in memory, the TSB is altered to link in the overlay; otherwise, the TSB is queued for SOVLDR and the task is suspended.
SO$BTO	.DXMISC.SOURCE.SO$CPR	Branch to system overlay. This routine is called by one system overlay to branch to another overlay. If the desired overlay is already in memory the calling task's TSB is modified; otherwise, the TSB is queued for OVLDR and the task is suspended.

Table 9-3 Task Management Routines -- Part 2 of 4

Routine	Source Module Pathname	Description
SO$RFO	.DXMISC.SOURCE.SO$CPR	Relink from system overlay. This routine is called by a system overlay to relink back to the last task or overlay that performed a link to overlay. If the relink is back to a task, or back to an overlay that is in memory, control is transferred immediately. If the relink is back to an overlay no longer in memory, the current task is suspended and the TSB is queued for SOVLDR.
TM$DOR	.TSKMGR.SOURCE.TM$DOR	Enforce access privileges to door of a particular structure. The door is represented as a queue. If the data structure is being accessed by a task, the task wanting access is queued and suspended. Queued tasks will be unsuspended and gain access when the current accessing task exits the door, via TM$OPN.
TM$OPN	.TSKMGR.SOURCE.TM$OPN	Open the door to a restricted access structure. This routine is called by a task to release access to the door. The next task in the queue is unsuspended and the door marked open.
TM$CLK	.TSKMGR.SOURCE.TM$OPN	Clock interrupt handler. Resets the timer interrupt, updates the second counter, clock unit counter and time slice counter.
TRAPRT	.TSKMGR.SOURCE.TM$RTN	Return point from all interrupt processors. Returns control to interrupted task (if its time slice has not expired), XOP, or other interrupt processor.
XOPRT1	.TSKMGR.SOURCE.TM$RTN	First return from XOP processors. Returns control to the executing task if its time slice has not expired. Otherwise, it saves the state of the executing task and returns control the scheduler.
SCHRET	.TSKMGR.SOURCE.TM$RTN	Return to scheduler by force.

Table 9-3 Task Management Routines -- Part 3 of 4

Routine	Source Module Pathname	Description
XOPRT2, XOPRT3	.TSKMGR.SOURCE.TMRTN .TSKMGR.SOURCE.TMRTN	Return points from XOP processors that want the calling task suspended. Task execution is halted and control returns to the scheduler.
TM$DPR	.TSKMGR.SOURCE.TM$DPR	Dynamic task priority routine. This routine is called by the device driver task to adjust the priority of a task installed with dynamic priority, according to the type of I/O it is performing.
TOLLNK	.TSKMGR.SOURCE.TM$TOL	Link a block of memory onto the head of the time ordered list (TOL).
TOLDEL	.TSKMGR.SOURCE.TM$TOL	Delink the specified block of memory from the time ordered list.
TOLTDL	.TSKMGR.SOURCE.TM$TOL	Delink the task segment of the specified task from the TOL.
TOLTLK	.TSKMGR.SOURCE.TM$TOL	Link the task segment of the specified task onto the head of the TOL.
TOLTSG	.TSKMGR.SOURCE.TM$TOL	Calculate the beet address of the start of the task segment of the specified task.
TMALPR	.TSKMGR.SOURCE.TMALPR	Allocate memory for a procedure. The procedure may be either in memory already, on a program file, or on the roll file. Called by the task loader.
TMIMAG	.TSKMGR.SOURCE.TMIMAG	Load program segment from image file. Loads either a task or procedure from a program file or the roll file. Called by TMLDTK and TMLDPR.

Table 9-3 Task Management Routines -- Part 4 of 4

Routine	Source Module Pathname	Description
TMLDPR	.TSKMGR.SOURCE.TMLDPR	Load a procedure from disk, either from a program file or the roll file. Called by task loader.
TMLDTK	.TSKMGR.SOURCE.TMLDTK	Load a task segment from disk, either from a program file or the roll file. Called by task loader.
TMRDAL	.TSKMGR.SOURCE.TMRDAL	Allocate space of the roll file. The allocation information is set up in the TSB or PSB of the segment being rolled. The TSB or PSB is linked onto the roll directory list. Called by task loader and memory management.
TMRDDL	.TSKMGR.SOURCE.TMRDLL	Delink the specified segment from the roll directory list. This causes the space occupied by the rolled segment to become available.
RSET12	.TSKMGR.SOURCE.TM$INT	Clears the Protection, Overflow, and WCS flags in the status register. This must be called whenever entering map 0 from an INT or XOP.

9.5 MEMORY MANAGER

Memory management routines perform such functions as allocating memory, releasing memory, and rolling out tasks, procedures and buffers. A special connection of routines, called buffer management, allocates, deallocates, reads, and writes file I/O buffers. Table 9-4 shows the major memory management routines. Table 9-5 shows the buffer management routines, which are generally called by file management.

Table 9-4 Memory Management Routines -- Part 1 of 2

Routine	Source Module Pathname	Description
MM$GSA	.MEMMGR.SOURCE.MM$MGR	Get system table area. This routine searches the free system area list for a block of the specified length and allocates it, if possible.
MM$RSA	.MEMMGR.SOURCE.MM$MGR	Release system table area. This routine releases the specified block of system area, placing it on the free list. The block is consolidated with neighboring blocks if possible.
MM$GUA	.MEMMGR.SOURCE.MM$MGR	Get user area. This routine searches the free user memory (all memory beyond DX10) list for a block of the specified length and allocates it, if possible.
MM$RUA	.MEMMGR.SOURCE.MM$MGR	Release user area. This routine links the specified block of user memory onto the free list and consolidates it with neighboring blocks, if possible.
MM$GSO	.MEMMGR.SOURCE.MM$MGR	Get system table area, clear to zero. This routine calls MM$GSA, then clears the allocated block (if successful) to zero.

Table 9-4 Memory Management Routines -- Part 2 of 2

Routine	Source Module Pathname	Description
MM$FND	.MEMMGR.SOURCE.MM$FND	Find (user) memory. This routine is called by all routines that need a block of user memory. Entry to the routine is restricted to serial access by the use of TM$DOR and TM$OPN. MM$FND first checks free memory, then scans the TOL for rollable blocks. If a rollable block is found, it is rolled.
MM$SCN	.MEMMGR.SOURCE.MM$SCN	Scan the TOL for a rollable block. This routine is called by MM$FND to get a rollable block of memory if there is not a large enough block of free memory. Rollable blocks may be task, procedure or buffer memory.
MM$ROL	.MEMMGR.SOURCE.MM$ROL	Roll a task or procedure segment to the roll file. This routine rolls a block to disk, assuming that roll file space is already allocated. Calls BM$MAP and FM$WTM.
MM$RLM	.MEMMGR.SOURCE.MM$TSK	Release task memory routine. This routine delinks the task memory of the specified task from the TOL and releases it. If there are attached procedures, their attached task counts are decremented. If the count for a procedure goes to zero, its memory and PSB are released.
RELPSB	.MEMMGR.SOURCE.MM$TSK	Release the specified PSB and procedure memory.
MM$TSB	.MEMMGR.SOURCE.MM$TSK	Release TSB and memory of a task suspended awaiting queue input. This routine searches the TSB list for a suspended queue serving task. If one is found, the task's memory and TSB are released.

Table 9-5 Buffer Management Routines -- Part 1 of 3

Routine	Source Module Pathname	Description
BM$RD	.MEMGR.SOURCE.BM$RD	Find a particular file blocking buffer (= file physical record) and map it into the calling task (usually file management). If the buffer is in memory, delink it from the TOL and map it in. Otherwise, allocate memory and read in the correct file physical record, then map it in.
BM$NEW	.MEMMGR.SOURCE.BM$RD	Same as BM$RD except that it will not read in a file record. BM$NEW is used to avoid reading a sequential file record when preparing to write it.
BM$CLO	.MEMMGR.SOURCE.BM$CLO	Close file. This routine writes all modified blocks of the file for a given LUNO (LDT) that are still in memory. The blocks remain on the TOL until needed by MM$FND.
BM$FLS	.MEMMGR.SOURCE.BM$CLO	Flush blocks. This routine returns all memory occupied by buffers for the specified file. Modified buffers are written to the file before being released. Memory resident buffers are marked empty and left on the TOL.
BM$SCH	.MEMMGR.SOURCE.BM$CLO	Search for a particular buffer on the TOL. THe search can be restricted by the following buffer criteria: ... Modified or unmodified buffer with equal LDT address. ... Modified buffer with equal LDT address. ... Modified buffer with equal FCB address.
BM$UPD	.MEMMGR.SOURCE.BM$CLO	Update file. This routine calls BM$SCH to find a buffer on the TOL according to the desired criteria. If the buffer is modified, it is written to the file.

Table 9-5 Buffer Management Routines -- Part 2 of 3

Routine	Source Module Pathname	Description
BM$MAP	.MEMMGR.SOURCE.BM$MAP	Map the specified number of bytes in the specified task's memory into the calling task.
BM$MPB	.MEMMGR.SOURCE.BM$MAP	Map the specified number of bytes from general memory into the calling task. This routine is given a beet address which begins the area to be mapped.
BM$RDM	.MEMMGR.SOURCE.BM$RDU	Read and update a buffer. This routine calls BM$RD to get the specified file buffer. If the buffer has been modified, it is written to the file.
BM$W	.MEMMGR.SOURCE.BM$W	Write a buffer. This routine writes the specified buffer, which is mapped into the calling task, to the specified physical record of the file (destination address need not be the same as the source file record from which the buffer was read).
BM$MPK	.MEMMGR.SOURCE.BM$MAP	Check a mapped segment for possible Write Protection violation.
BM$IOW	.MEMMGR.SOURCE.BM$W	Set up the I/O call block to write the specified buffer to the specified record in the file. After building the call block, it calls FM$IO to perform the I/O.
BM$IOR	.MEMMGR.SOURCE.BM$W	Set up the I/O call block to read the specified buffer from the specified record on the file. Calls FM$IO to perform the I/O.
BM$LNK	.MEMMGR.SOURCE.BM$W	Link the specified buffer onto the TOL.
BM$DEL	.MEMMGR.SOURCE.BM$W	Delink the specified buffer from the TOL.

Table 9-5 Buffer Management Routines -- Part 3 of 3

Routine	Source Module Pathname	Description
BM$REL	.MEMMGR.SOURCE.BM$REL	Release a buffer to buffer management. This routine unmaps a buffer from the calling task and links it onto the TOL. Modified buffers are written to the file.
BM$RMD	.MEMMGR.SOURCE.BM$REL	This routine is the same as BM$REL except it presets the buffer's modified flag, forcing a write to the file.
BM$TRM	.MEMMGR.SOURCE.BM$REL	Trim a buffer from memory. This routine releases the specified buffer to user memory. If the buffer has been modified, it is first written to the file.
BM$WRN	.MEMMGR.SOURCE.BM$RDU	Write a buffer and rename it. This routine calls BM$W to write the specified buffer, then modifies the buffer overhead to make the buffer correspond to the destination file record.

9.6 DISK MANAGER

The disk manager consists of a memory resident, queue-serving task and several system overlays. The queue server is the main driver. It decodes the buffered SVC opcode and links to the correct processor (which is a system overlay) for that opcode. The disk management opcodes include:

... 0 - deallocate a block

... 1 - allocate all of the requested amount

... 2 - allocate as much of the requested amount as possible

... 3 - allocate as much as possible at the address requested or fail.

Table 9-6 shows the major routines included in disk management.

Table 9-6 Disk Management Routines -- Part 1 of 2

Routine	Source Module Pathname	Description
DM$PC	.DSCMGR.SOURCE.DM$TSK	Queue-serving main driver for disk management.
DMALLC	.DSCMGR.SOURCE.DMALLC	Disk allocation main driver. This routine processes all of the allocation opcodes. It converts the requested number of file blocks (physical records) to a number of ADUs, then attempts to allocate according to the restrictions implied by the opcode.
DMDALC	.DSCMGR.SOURCE.DMDALC	Deallocate disk space. This routine deallocates the specified ADUs by resetting the bits in the correct partial bit maps.
ADJALC	.DSCMGR.SOURCE.ADJALC	Adjust allocation count. This routine computes the number of ADUs in a given block on contiguous free ADUs which are useful to allocate file physical records of a given ADU size.
ALCSCN	.DSCMGR.SOURCE.ALCSCN	Allocation bit map scans. This routine contains two bit map scans. The first scans a partial bit map for a particular allocation placement (allocation must start at particular ADU). The second is a first-fit scan which starts with the first partial bit map and searches for a large enough block of free ADUs.
CHGMAP	.DSCMGR.SOURCE.CHGMAP	Change disk allocation bit map. This routine sets or resets bits in the disk resident bit map to reflect the newly completed allocation or deallocation operation. This routine is the common exit path from DMALLC and DMDALC.

Table 9-6 Disk Management Routines -- Part 2 of 2

Routine	Source Module Pathname	Description
DM$TBL	.DSCMGR.SOURCE.DM$TBL	Disk management table builder. This routine scans the partial bit map currently buffered in memory, and fills in the disk management table (DMT) entries particular to that bit map.
EXTEND	.DSCMGR.SOURCE.EXTEND	Extend an allocation across partial bit map boundaries.
MAPPBM	.DSCMGR.SOURCE.MAPPBM	Map partial bit map. This routine reads/writes the specified partial bit map from/to the disk.
RDPBM	.DSCMGR.SOURCE.RDPBM	Read partial bit map. This routine reads the specified partial bit map from the specified disk to the specified buffer.
WRTPBM	.DSCMGR.SOURCE.WRTPBM	Write partial bit map. This routine writes the specified buffered partial bit map to the correct sector on the specified disk.
SCNBIT	.DSCMGR.SOURCE.SCNBIT	Scan for a bit of the opposite state. This routine scans a buffered partial bit map from the specified bit position until it finds a bit of the opposite state.
SETBIT	.DSCMGR.SOURCE.SETBIT	Set bits to the given state. This routine sets the specified number of bits in a bit map, starting at the specified bit position, to the specified state (0 or 1).
WCHPBM	.DSCMGR.SOURCE.WCHPBM	Calculate a partial bit map number and bit position from the specified ADU number.

9.7 DEVICE I/O PROCESSING

In addition to the DX10 I/O supervisor, there are several other routines) which are used to process device I/O calls. These other routines include the device driver task (DDT), the device service routines (DSRs), and several common routines. Table 9-7 shows the major routines involved in processing device I/O.

Table 9-7 Device I/O Processing Routines

Routine	Source Module Pathname	Description
DXIOS	.DXIO.SOURCE.DXIOS	DX10 I/O supervisor. This routine processes SVC code 0. It preprocesses calls for device I/O, file I/O, and file utility services, buffering the call block and queueing it for the appropriate queue server or DSR. It also processes all I/O to the DUMY device.
DDT	.DXIO.SOURCE.DDT	Device driver. Serves all device queues. Initiates device I/O, starts the DSRs, and does end-of-record processing (i.e. unbuffers data to tasks doing device I/O).
TMOUT	.DXIO.SOURCE.DSRTMX	Device time out check. This routine is called by the scheduler after a system time unit has elapsed. It scans the PDT list to see if a device has a time-out error, or if the re-enter-me flag in the PDT is set. If the re-enter-me flag is set, control is passed to the DSR. If the device has a time-out error the I/O is aborted.
FSTXFR	.DXIO.SOURCE.FSXTXFR	Routine which tests a file I/O request, .DXIO.SOURCE.FSTXFR, to see if a "fast transfer" is possible.

Table 9-8 shows the Device Service Routines included with DX10. Note that the source modules for all DSRs are cataloged under the library .DEVDSR.SOURCE on the disk.

Table 9-8 Device Service Routines

Routine	Description	Source Module Name
CAS733*	DSR for cassette units on a 733ASR.	.CAS733
CRDSR	DSR for 804 card reader.	.CRDSR
DDIOSR	DSR for direct disk I/O.	.DDIOSR
DSR911	DSR for 911 VDT.	.DSR911
DSR913	DSR for 913 VDT.	.DSR913
DSR979	DSR for 979 magnetic tape drive.	.DSR979
DSRKSR*	DSR for 733 KSR and 743 KSR.	.DSRKSR
FPYDSR	DSR for FD800 diskette.	.FPYDSR
LPDSR	DSR for 306, 588, 810, 2230 and 2260 line printers.4	.LPDSR
DSR820	DSR for 820 keyboard device.	.DSR820
DSRTPD	DSR for teleprinter devices	.DSRTPD .COMISR .TTYISR .TPDCOM

* CAS733 and DSRKSR are linked to form DSR733 for the 733 ASR.

Several routines commonly used by DSRs are contained in the source module .DXIO.SOURCE.IOCOMX. The routines are:

... BZYCHK See if device is busy.

... SETWPS Set up interrupt mask and workspace.

... ENDRCD Activate end-of-record routine (part of DDT).

... XFERM Put format (direct disk I/O) data in buffer.

... GTADDR Calculate a 20-bit absolute address from a 16-bit mapped address (used by DSRs for TILINE devices).

... MAPCHK Verify that the specified address is mapped into a user's space.

... BUFCHK Verify that a buffer is mapped into a single base/limit register pair.

... BRCALL Branch table call.

... BRCALT Alternate branch table call.

... JMCALL Jump table call.

... JMCALT Alternate jump table call.

... SCNPDT Scan PDT list, and enter power-up routine for each device.

... PUTEBF Put a character in the event character buffer (for keyboard DSRs).

... PUTCBF Put a character in the character queue (for keyboard DSRs).

... GETC Get a character from the event buffer or character queue (for keyboard DSRs).

... KEYFUN Recognize HOLD, ABORT, or BID keys from a keyboard.

... TILERR Move the TILINE image to the system log buffer in the PDT extension for TILINE devices (used by disk and magnetic tape DSRs).

... ASCCHK Compare an ASCII character to a table of characters and transfer control to the address associated with the matched character.

9.8 FILE UTILITY ROUTINES

File utility SVCs are queued by DXIOS for the file utility task, FUTIL (task >0B on the system program file). FUTIL consists of a main driver, FU$, and several routines to process the different file utility opcodes. It also contains two conversion routines, LC$ and FC$, which convert the still supported librarian and FUR call blocks to DX10 3.0 file utility call blocks.

Table 9-9 shows the major file utility routines which make up the file utility task, FUTIL. Note that all utility source modules are cataloged under the library .FUTIL.SOURCE on the source disk.

Table 9-9 File Utility Routines -- Part 1 of 5

Routine	Source Module Pathname	Description
FU$	.FU$	Main driver and queue server for file utility requests. Decodes file utility opcode and branches to correct processor. If opcode is FUR or librarian, branch to FC$ or LC$, respectively.
FC$	.FC$	Convert FUR call to new call block. This routine converts the call, calls UC$ to execute it, calls CLEAN to unbuffer the call, then returns to FU$.
LC$	.LC$	Convert librarian calls to new call block. This routine converts the call, processes it, calls CLEAN to unbuffer the call block, then returns to FU$.
UC$	.FU$	Normal utility call processor. This routine checks for bad opcodes, looks up the correct processor for the given opcode (table is in FU$ also), and branches to that processor. The processor returns to UC$, which returns to the caller (either FU$, LC$, or FC$).
AA$	.AA$	Add alias opcode processor. This routine adds an alias to an existing file. A LUNO must have been previously assigned to the file. Return is to calling task (UC$).
I$ADR	.AA$	Initialize alias descriptor record. This routine initializes a buffered ADR (see section on disk data structures) and then returns to AA$.
AL$	.AL$	Assign LUNO opcode processor. This routine assigns a LUNO to either a file, a device, or a temporary file. It also builds the necessary FCB/LDT tree in the system table area.

Table 9-9 File Utility Routines -- Part 2 of 5

Routine	Source Module Pathname	Description
CF$	.CF$	Create file opcode processor. This routine creates a file, including an FDR on disk, disk allocation for the file, and an FCB in memory.
DP$	.DP$	Delete protect opcode processor. This routine sets the delete protect flag bit in both the FCB and the FDR (on disk) for the specified file.
RF$	.RF$	Rename file opcode processor. This routine moves the existing FDR to the destination directory and releases the old FDR directory entry. If an existing file has the new pathname, it is replaced by the renamed file.
RL$	.RL$	Release LUNO opcode processor. This routine sets up registers using values from the buffered call block, calls RL$LUN to release the LUNO, then returns.
RL$LUN	.RL$	Internal entry point to release LUNO opcode processor. This routine calls LDTSCH to find the LDT for the specified LUNO. The LDT is delinked from all chains, and any file buffers associated with the released LUNO are flushed (released). The LDT is released to system table area.
SF$	.SF$	Set forced write flag opcode processor. This routine sets the forced write flag in the specified LDT to the specified state.
UP$	.UP$	Unprotect file opcode processor. This routine reads the FDR for the specified file from its parent directory, resets the protection flags in the FDR to zero, and rewrites the FDR back to the directory.

Table 9-9 File Utility Routines -- Part 3 of 5

Routine	Source Module Pathname	Description
VP$	.VP$	Verify pathname opcode processor. This routine checks a pathname for valid syntax, and then tries to find the file. If the file exists, relevant information is returned.
WP$	.WP$	Write protect opcode processor. This routine sets the write protect bit in the FDR for the specified file.
DA$	.DA$	Delete alias opcode processor. This routine delinks the alias descriptor record for the specified alias from the alias list in the directory file (see the section on disk data structures).
DF$	.DF$	Delete file opcode processor. This routine releases all primary and secondary file allocations on the disk and releases all directory entries (FDRs and ADRs).
AL$PNC	.AL$	Assign LUNO to pathname component. This routine returns the FCB address for the specified pathname component. If no FCB exists,one is built and added to the FCB/LDT tree.
T$FILE	.AL$	Assign LUNO to temporary file. This routine generates a unique pathname for the temporary file, then calls AL$ to assign the LUNO to it.
T$ADD	.AL$	Add a new FCB node to the FCB/LDT tree.
GENLUN	.AL$	Generate a unique LUNO. This routine searches a given LDT list and returns a LUNO not currently existing in the list.
DEVSCH	.AL$	Search PDT list for device name. If the desired device is found, the PDT address is returned.

Table 9-9 File Utility Routines -- Part 4 of 5

Routine	Source Module Pathname	Description
VOLSCH	.AL$	Search the volume tables for volume name. If the specified volume has been installed, the PDT address of the drive on which it isinstalled is returned.
AL$DEV	.AL$	Assign LUNO if a device. This routine searches the PDT list for the desired device and assigns a LUNO to it.
AL$FIL	.AL$	Assign LUNO to file. This routine gets the system table area and builds an LDT for a LUNO to assign to a file.
AL$PAR	.AL$	Assign LUNO to parent. This routine assigns LUNO >CA to the parent directory file of the specified file.
B$FDR	.CF$	Build a file descriptor record and write it to the specified directory record.
F$INIT	.CF$	Initialize a file based on its file type.
C$DFLT	.CF$	Compute file creation parameter defaults, based on file type.
CRBLK	.CF$	Compute the number of file physical records required for a file based on file type and specified initial allocation.
A$BLK	.CF$	Allocate the specified number of physical records on the disk. This routine calls the disk manager to allocate the required disk space.
R$DSC	.DF$	Release disk space. Calls the disk manager to release the primary file allocation and all secondary allocations.

Table 9-9 File Utility Routines -- Part 5 of 5

Routine	Source Module Pathname	Description
R$FDR	.DF$	Release file descriptor records. Releases all FDR entries in the directory for the file being released.
RSALS	.DF$	Release aliases. Releases all ADR entries in the directory for aliases of the file being deleted.
FLLRMV	.RL$	Remove and clean up file LDT. Delinks a file LDT from all chains. If the file to which the LDT was assigned has no more LUNOs assigned and no descendants, its FCB is released.
CLEAN	.FU$	Clean up request. Calls TMQUE to queue the buffered call block for the SVC clean up task, SVCCLN.

Many routines commonly used by file utility processors are contained in the file .FUTIL.SOURCE.US$. Some of the routines included are:

... LDTSCH - search the LDT tree at the specified level (task, station, global) for the specified LDT

... FNDLUN - find the specified LDT (level does not matter)

... LDTRMV - remove the specified LDT from all chains

... LDTENT - enter the specified LDT in the LDT tree

... HASH - compute a hash key value for the specified pathname component

... LOOKUP - look up the specified file by name

... I$BLK - initialize file blocking buffer

... G$REC - transfer the specified directory record to the caller's buffer

... FILE10 - performs all disk I/O for FUTIL

... T$CLEN - clean up the FCB tree (release all unnecessary FDBs on the upward path from a single leaf node)

... FDRFCB - translate a buffered FDR and associated blocks into an FCB

9.9 FILE MANAGER

File management under DX10 consists of a pool of memory resident, queue serving tasks and four system overlays. The main driver of each task is a routine called FM$TSK. This routine is activated whenever the I/O supervisor, DXIOS, places an entry on its queue. FM$TSK dequeues each entry and passes control to the correct processor for the specified I/O opcode. The processor returns to FM$TSK, which unbuffers the I/O call block, reactivates the calling task, and gets the next entry on the queue. FM$TSK issues an SVC >24 when its queue is empty. Table 9-10 shows the major routines that process file I/O calls. All source modules are in directory .FILMGR.SOURCE.

Table 9-10 File I/O Processors -- Part 1 of 2

Routine	Description	Overlay	Source Module Name
FM$TSK	Main driver of file manager. Looks up opcode in table, branches to correct processor.	N	.FM$TSK
FMOPEN	Open file processor. Checks access privileges for conflicts.	N	.FMOPEN
FMCLOS or FMCLUN	Close file processor. File buffers are updated to disk, locked records are unlocked.	N	.FMCLOS
FMCLEF	Close file with EOF. Writes end-of-file, then calls FMCLOS.	Y	.FMCLEF
FMOPRW	Open and rewind file processor. Calls FMOPEN, then to FMRWND.	Y	.FMOPRW
FMRDST	Read file status processor. Calls BM$MAP to map in user buffer, then writes file characteristics in buffer.	Y	.FMRDST
FMFBSP	Forward/backward space processor. Calls FBSP to reset LDT pointers.	Y	.FMFMBSP
FMREAD	Read ASCII and read direct processor. Calls BM$MAP to map in user buffer, then calls BM$RD (blocked file) or FM$I/O (unblocked) to get proper physical record. Transfers proper logical record to user buffer and releases file buffer (if blocked file).	N	.FMREAD

Figure 9-10 File I/O Processors -- Part 2 of 2

Routine	Description	Overlay	Source Module Name
FMWRIT	Write ASCII and write direct processor. Calls BM$MAP to map in user buffer, then gets proper physical record through BM$RD (blocked file). New logical record transferred (via FM$IO if unblocked). Buffer (if any) is released.	N	.FMWRIT
FMWEOF	Write end-of-file processor. Writes an EOF (zero length) record to a sequential file. EOFs to relative record files are ignored.	Y	.FMWEOF
FMRWND	Rewind file processor. Resets LDT pointers to show that the file is rewound (before the first file record).	Y	.FMOPRW
FMRWRT	Rewrite record processor. Backs up one record on a sequential or relative record file and writes the user's buffer to that record (calls FMWRIT to write).	Y	.FMRWRT
FMACES	Modify access privileges processor. Checks for access conflicts between different users of a file; if none, modifies LDT flags to reflect new privileges.	N	.FMACES
FMOPXT	Open extend processor. Calls FMOPEN to open the file, sets LDT pointers to end-of-medium, then backspaces over any EOFs at end of file.	Y	.FMOPXT
FMOPUB	Open for Unblocked I/O. Calls FMOPEN to open the file, then sets flag in LDT to allow unblocked I/O to the file.	Y	.FMOPUB
FM$IO	File manager disk I/O routine. Maps file physical record no. into ADU/sector offset disk disk address and transfers data between user buffer and disk.	N	.FM$IO

9.9.1 KEY INDEXED FILES.

Key indexed file I/O processing is a major part of file management. KIF I/O processing routines are contained in one memory resident linked object module, KIF, and five system overlays. Table 9-11 shows the major routines involved in key indexed file I/O processing. All source modules are in directory .KIFILE.SOURCE.

Table 9-11 Key Indexed File I/O Processors -- Part 1 of 2

Routine	Description	Overlay	Source Module Name
KI$BEG	Key indexed file I/O driver. Receives key indexed file I/O requests from FM$TSK, decodes opcode, and branches to correct processor.	N	.KI$BEG
OLN002	Insert record into key indexed file, using the primary key. Key is hashed to get a block number and then record is inserted. Key is added to B-tree for primary key.	Y	.OLN002
OLN008	Delete a record from a key indexed file, either by key value or currency information. Record is removed from block and key values from B-trees.	Y	.OLN008
KI$BTD	Delete an entry from a B-tree. Searches a B-tree for the specified key. If found, entry is deleted. If the B-tree entry is a B-tree divider (see section on disk organization), OLN009 is called to delete the entry from the next higher node in the B-tree.	N	.KI$BTD
OLN009	Continuation of B-tree delete routine.	Y	.OLN009
KI$RR	Read a record from a key indexed file. If no currency information is provided, searches B-tree for specified key and sets up currency. Calls BM$MAP to map in user buffer, reads desired record using currency information, releases file blocking buffer.	N	.KI$RR

Table 9-11 Key Indexed File I/O Processors -- Part 2 of 2

Routine	Description	Overlay	Source Module Name
KI$SC	Driver for set currency commands. Sets user's currency information to point to the data record and the B-tree position corresponding to the specified key value.	N	.KI$SC
KI$RN	Read next record. Uses currency information to find record containing next largest key, calls KI$RD (same as BM$RD) to read the record.	N	.KI$RN
KI$BIT	Insert an entry into a B-tree. Inserts a new key value in the proper leaf node of the B-tree. If the node becomes full, calls OLN001 to split the leaf into two new nodes and add an entry to the next higher node.	N	.KI$BIT
OLN001	B-tree split routine.	Y	.OLN001
KI$BTS	Search B-tree for specified key value. If found, a stack that traces the path down the B-tree to the correct leaf node is created; otherwise, the routine finds the leaf node in which the key should fit if it existed, and the stack is still created.	N	.KI$BTS
KI$GRF	Get free block. This routine is used to get an overflow block or B-tree block from the free block chain. If the last free block is returned, another secondary allocation is made to the file.	N	.KI$GFR
OLN004	Open random and close opcode processor.	Y	.OLN004
OLN003	Subroutines used by KI$RW (rewrite). This overlay contains three pieces of code used by KI$RW (rewrite).	Y	.OLN003

SECTION 10

SYSTEM COMMAND INTERPRETER

10.1 GENERAL

This section describes the routines that make up SCI, as well as the data structures and files used by the command interpreter. In this description, SCI is divided into four parts: the command interpreter, the background resource manager, the background task bidder, and the output queuer. The following paragraphs describe those parts of SCI.

10.2 SYSTEM COMMAND INTERPRETER

The function of the command interpreter, SCI990, is to interpret commands entered at a user terminal or listed in a sequential file. The command language consists of a set of primitive operations, whose names start with a period (e.g., ".BID"), and a procedure definition and parameter gathering facility that permits the user to extend the set of commands.

SCI990 executes in both batch and interactive modes. In interactive mode, the terminal user may be prompted for the values of command parameters. In batch mode, parameters are specified by keyword assignments in the command statement. Except for accessing the commands and their parameters, the command interpreter is essentially indifferent to the mode of operation. This document mentions a mode of operation (Batch, VDT, TTY) only when the description does not apply to all modes.

10.2.1 STRUCTURE OF SCI.

SCI990 has two functions, parsing and then executing a command. The parsing function includes: displaying menus and messages; and reprompting for invalid input. The execution of a command may be performed internally (for primitive operations) or result in evaluation of a procedure definition. Figure 10-1 shows a generalized flow of control through SCI990.

The structure of the system command interpreter is composed of direct procedure calls. Indirect calls ("A" calls "B" calls "C") are listed in a cross reference table at the end of this section.

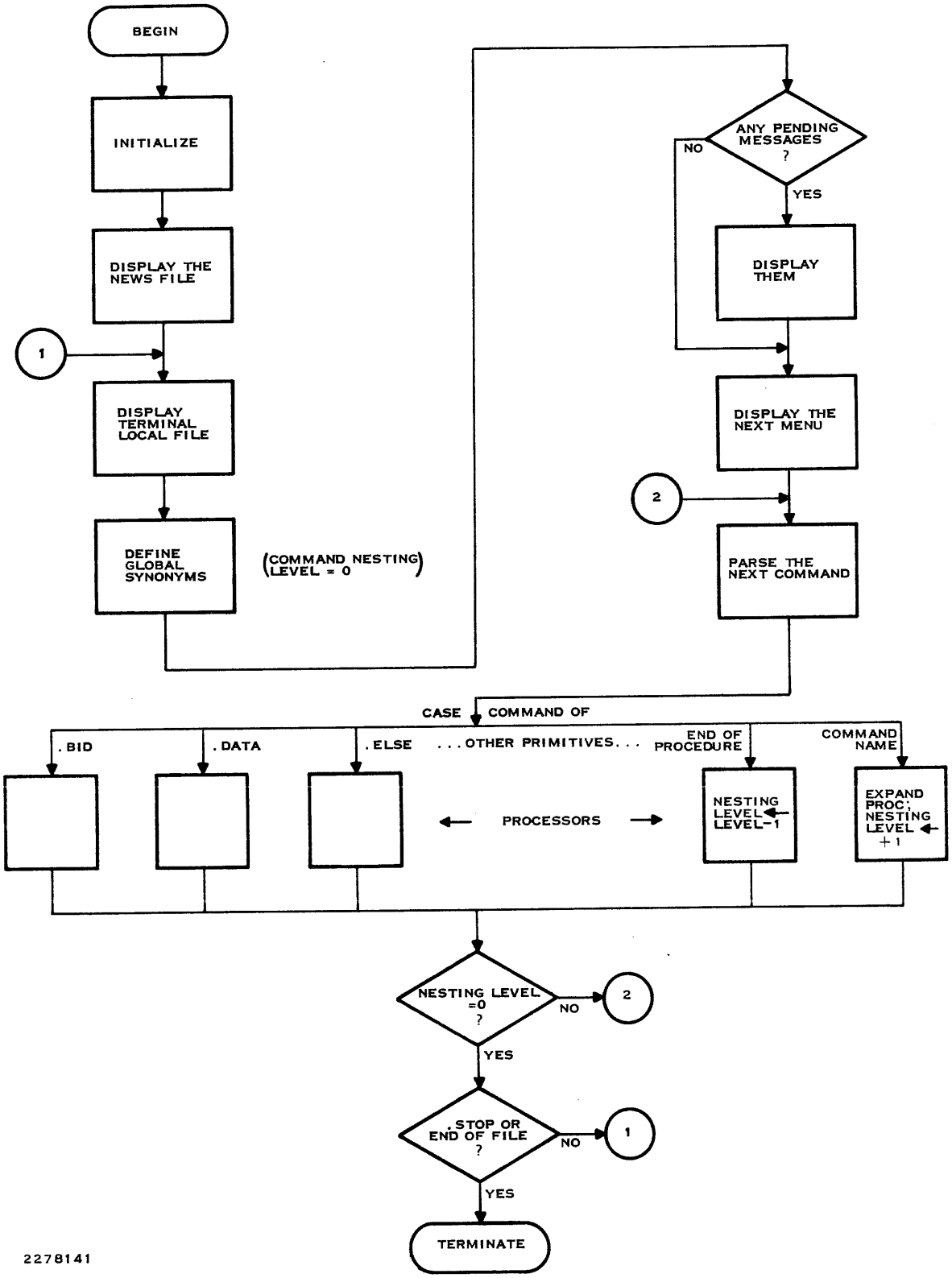


Figure 10-1 SCI Flow of Control

10.2.2 OVERLAY STRATEGY.

There are two kinds of overlays for the system command interpreter task. One kind is the overlay structure designed for use with the .OVLY primitive command (executed by the S$OVLY routine). The text editor and debugger are examples of this kind. Other overlays, such as PARSER and secondary overlays of the debugger, do not use the .OVLY structure, but are loaded by means of a Load Overlay SVC.

The command interpreter itself has two overlays that are part of its basic operation. PARSER contains the bulk of the routines used for parsing a command,expanding a procedure definition, and executing a primitive command. The PARSER overlay is loaded as needed by the GETCMD routine. DERROR contains the error formatting and display routines (including the English language text for messages) and the log-in/log-out routines. The DERROR routines are accessed by means of S$OVLY while the PARSER routines are accessed by means of a Load Overlay SVC.

A third overlay, TINFO, may be considered part of SCI990 proper. TINFO contains the command processors for the following commands:

LTS - List Terminal Status
MTS - Modify Terminal Status
SBS - Show Background Status
KBT - Kill Background Task

MSG - Display Message at Own Terminal
CM - Create Message

AUI - Assign User ID
DUI - Delete User ID
MUI - Modify User ID
LUI - List User IDs

WAIT - Wait For Background Task To Complete

A fourth overlay, OUTQUE, contains the command processors for the output queuer:

PF - Print File At Device
HO - Halt Output At Device
RO - Resume Output At Device
KO - Kill Output At Device
SOS - Show Output Status

Other overlays are supporting routines for various command processors, such as the text editor and debugger.

The general strategy for partitioning the command interpreter between the shared procedure area, SCI, and the PARSER overlay is to attempt to keep the size of the shared procedures plus the task area plus the largest overlay as small as possible, but to make the shared area as large as possible. Any of the routines in the PARSER overlay can be moved into the shared procedure SCI should PARSER become the largest

overlay. However, care should be taken in moving routines from SCI to PARSER, since PARSER is only loaded by GETCMD and many of the routines (e.g., PUTLINE, GETFIL) may be called when PARSER is not loaded. For example, GETFIL is called during error recovery from a procedure expansion, which may occur while a user overlay is loaded.

10.2.3 DATA STRUCTURES.

The following paragraphs describe the internal data structures created and maintained by SCI.

10.2.3.1 System Communication Area (SCA).

The SCA is an area of memory shared (as a "dirty" procedure) by all command interpreters, the background resource manager and the background request handlers (QBID, OQUEUE). The SCA contains a table of global system data, such as global LUNOs and task bid IDs, plus a table called an SCA entry for each terminal that uses SCI990.

10.2.3.2 SCA Entry.

Each terminal has a 32-byte entry in the SCA for communication between the system command interpreter and the background resource manager (BRM). The fields of the SCA entry are labeled and defined as follows.

```
SCA$TT    EQU  0     Terminal Type
SCA$TI    EQU  1     Terminal ID
SCA$DV    EQU  2     Terminal Device Name
SCA$UI    EQU  6     User ID
SCA$FO    EQU  12    FG Opcode
SCA$BO    EQU  13    BG Opcode
SCA$FE    EQU  14    FG Error Code
SCA$BE    EQU  15    BG Error Code
SCA$FS    EQU  16    FG Status
SCA$BS    EQU  17    BG Status
SCA$FT    EQU  18    FG Task ID
SCA$BT    EQU  19    BG Task ID
SCA$FL    EQU  20    FG Task LUNO
SCA$BL    EQU  21    BG Task LUNO
SCA$FC    EQU  22    FG "CODE" Value
SCA$BC    EQU  23    BG "CODE" Value
SCA$FR    EQU  24    FG Return Code (CC)
SCA$BR    EQU  25    BG Return Code (CC)
SCA$FI    EQU  26    FG SCI Task ID
SCA$BI    EQU  27    BG SCI Task ID
```

The fields SCATT, SCAFS, and SCA$BS are subdivided as follows:

SCA$TT BYTE 0 Terminal/User Type Information

Bit	
0	1 = Terminal is disabled
1-3	User privilege code (0-7)
4-7	Current terminal mode 0 = Batch mode 1 = TTY mode. F = VDT mode

Memory images of every user's TCA are maintained on disk in a library of user TCA images. During the log-in process, a user's TCA image is loaded into the SCI990 task. When the TCA is passed from one system task to another, it is transferred via the medium of a record on a background or foreground TCA file. The overall layout of the TCA is shown in Figure 10-2.

SCA$FS BYTE 16 Foreground Status

Bit	
0	1 = Login is required
1-3	Default user privileges
4-7	Default terminal mode 0 = Batch mode 1 = TTY mode F = VDT mode

SCA$BS BYTE 17 Background Status

Bit	
0	1 = BG Task pending
1	1 = Message pending
2	1 = BG Task complete
3	1 = BG Task bid error
4-7	Reserved

10.2.3.3 Text String.

Strings of characters are uniformly represented within SCI990 as a series of bytes and a pointer. The first byte, which is addressed by the pointer and may be on an odd byte address, contains the count of the number of characters in the string. The following bytes contain the characters. The null (empty) string is represented as a string with length zero.

An output buffer for a routine which returns a string value usually must have the buffer capacity in the first byte so that buffer overflows can be prevented. Some examples of string constants and buffers follow:

```
STR1    BYTE 10
        TEXT 'STRING ONE'
BUF1    BYTE 31
        BSS  31
```

Note that the first (count) byte is not included in the count. The count refers to the number of following bytes.

10.2.3.4 Terminal Communications Area (TCA).

The terminal communications area (TCA) has three purposes. First, it contains a description of the user currently logged in at the terminal. This description includes his user ID, status, encoded passcode, and allotted terminal time. Secondly, the TCA contains the name correspondence table (NCT) belonging to the user. The NCT contains the user defined synonyms and their values. Thirdly, the TCA is used to pass information, including parameters, from one system task to another. These task parameters are embedded in the NCT.

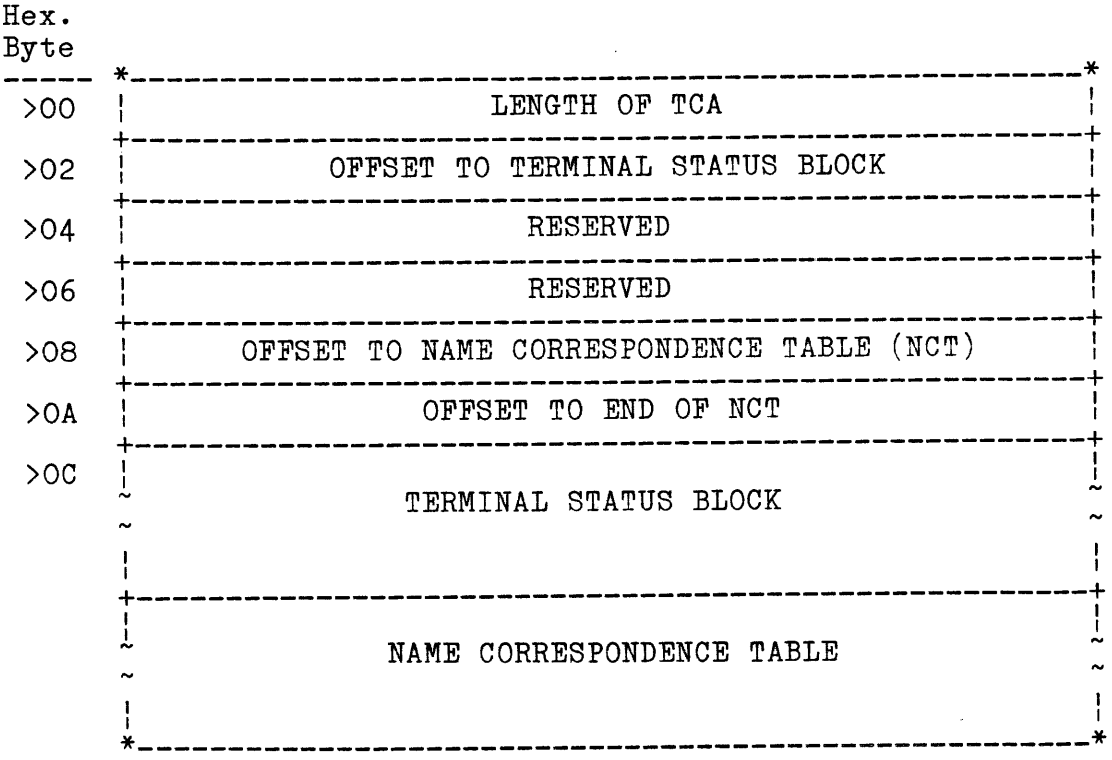


Figure 10-2 TCA Layout

10.2.3.5 Terminal Status Block (TSB).

The TSB, as shown in Figure 10-3, is used to identify the logged-in user to the various command processors. The encoded passcode permits passcode verification without exposing the actual passcode value. The user status information is copied to the terminal SCA entry at log-in, and specifies the level of user capabilities in the system. The FG task completion code is the medium by which FG task completion codes are returned (those tasks executed with ".BID").

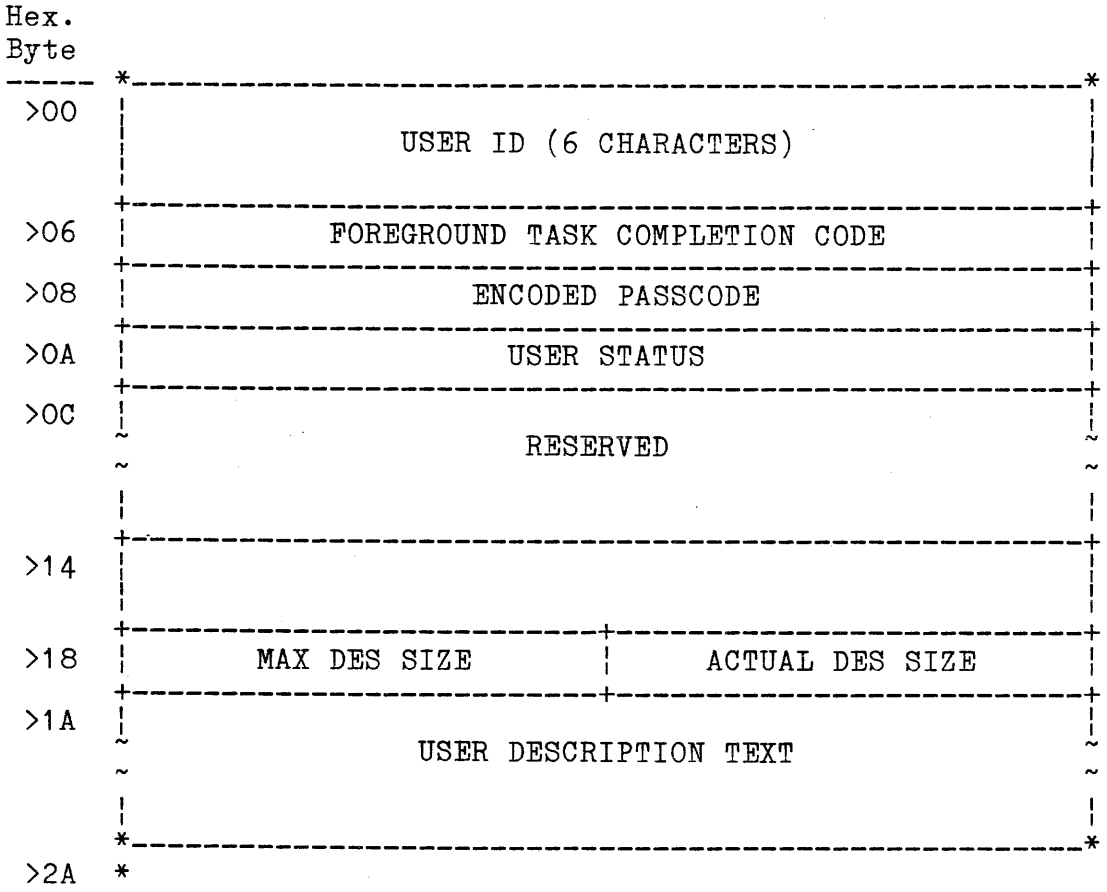


Figure 10-3 Terminal Status Block

10.2.3.6 Name Correspondence Table (NCT).

The NCT, as shown in Figure 10-4, consists of pairs of text strings terminated by a zero byte. Each text string is formed from a series of characters preceded by the count of the number of characters. A length of zero is not permitted. User defined synonyms may consist of printable characters only.

Positional parameters (elements of the "PARMS" list on a .BID, .QBID, or .OVLY command) are transmitted to the command processor program via special entries of the following form:

```
Byte 3, 0, 0, Parameter Number      Synonym
Byte 15
Text 'DS02.SYS.SOURCE'              Value
```

A program completion message (argument R2 of S$STOP) returned by a task executed via a .BID or .QBID command is transmitted back to the FG command interpreter as a bogus parameter 0.

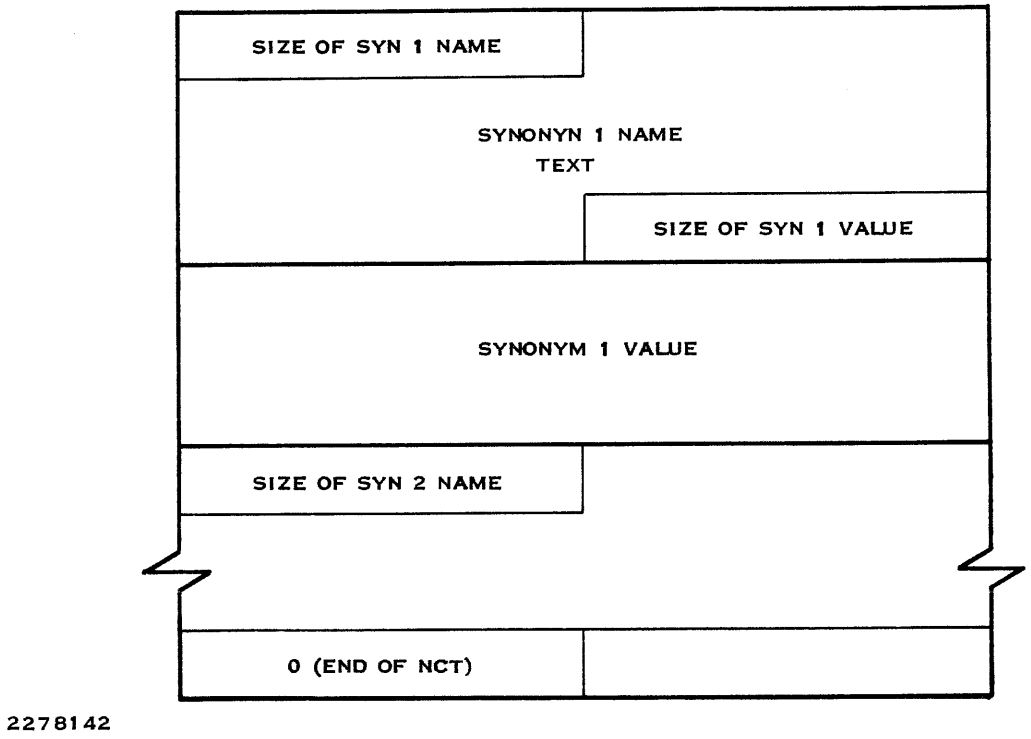


Figure 10-4 Name Correspondence Table

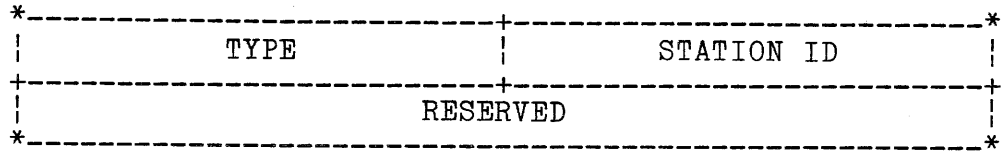
10.2.4 INTERFACES.

SCI interfaces to the user, and to various SCI routines, through the terminal and several files. The following paragraphs describe the interfaces.

10.2.4.1 Calling Sequence.

The calling sequence for SCI990 depends on the mode of operation. In interactive (VDT or TTY) mode, the SCI task is bid by the operating system in response to the user entering RESET followed by an exclamation mark "!". In batch mode, the task is bid by the QBID mechanism (see paragraph 10.4) as a result of an XB command. The mode

is determined by the four bytes passed as parameters of the Bid Task SVC, which are formatted as follows:



The TYPE field is a copy of the SCA$TT (terminal type) field in the terminal SCA entry. The STATION ID is the terminal (station) number. In interactive mode, both the type and STN ID fields are zero and the information is derived from a SELFID SVC and the SCA entry. In batch mode, the fields are (and must be) non-zero to distinguish the mode.

10.2.4.2 Terminal Local File.

The terminal local file (TLF) is a buffer on disk for lines to be displayed to the user. The lines are buffered so that VDT users can scroll back and forth through them and so they can be listed together in batch mode. The name of the file is determined from the SCI mode and the terminal number as follows:

FOREGROUND:	.S$FTLF**	where	** = terminal number
BACKGROUND:	.S$BTLF**	where	** = terminal number

The interactive modes of SCI run in the foreground, while the batch mode executes in background.

The TLF is written to by the S$IO routines: S$OPEN, S$WRIT, S$WEOL, and S$CLOS. Whenever SCI990 prepares to input a new command, the TLF is displayed (TTY, VDT modes) or listed (batch mode) by the S$SHOW routine.

10.2.4.3 System Procedure Library.

Commands are mapped into procedure file names by concatenating them on the end of the current procedure library name. Unless the user overrides the default PROC library name with a .USE command, the standard system PROC library is used. This directory is named .S$PROC.

10.2.4.4 Menu Files.

Menus are displayed in VDT mode by displaying the contents of a menu file. Menu files are named by concatenating the name of the system PROC library, the characters ".M$", and the menu name. The default menu file is .S$PROC.M$LC, which is also displayed in response to "/LC".

10.2.4.5 TCA Library File -- .S$TCALIB.

The terminal communication area (TCA) is described in paragraph 10.2.3.4 A TCA image is created for each user when his user ID is assigned. This TCA image resides on the TCA library file .S$TCALIB while the user is not logged in. Each record of the TCA library file holds exactly one TCA image. The number of the record containing the TCA image for a particular user ID is the same as the numeric part of the ID. For example, a user ID of "ABC010" uses record 10 (or >A) or the TCA library file. The numeric parts of user IDs must therefore be unique. Since the TCA library is implemented as an unblocked, relative record file, DX10 allocates records in blocks. Thus it is desirable that the record numbers be grouped, preferably by assigning them sequentially starting at 1. Unneeded user IDs can be deleted and their TCA library records reused by assigning the same numeric part of the ID. For example, delete user ABC010 and assign user DEF010.

The TCA library file .S$TCALIB is accessed only during log-in and log-out (in the DERROR module) and by the command processors for AUI, DUI, LUI, and MUI (in the TINFO module). The byte field SCA$L5 in the SCA module defines the globally assigned LUNO used for accessing the TCA library file (>03).

10.2.4.6 Foreground TCA File -- .S$FGTCA.

When a user logs in at a terminal, SCI990 reads his TCA image into memory. While the user is logged in, this TCA image is used by the various command processors to maintain his list of synonyms, maintain a history of the status of his operations, and transmit parameter values and messages between the various components of the SCI system. When a command processor is implemented as a task separate from SCI990, the TCA image is written to a record of the foreground TCA file for the command processor task to read it. Routines S$PTCA and S$GTCA put and get the TCA, respectively. The record number of TCA file is the same as the terminal number. The foreground TCA file, S$FGTCA, is accessed through the global LUNO found in byte SCA$L3 of the SCA module.

10.2.4.7 Background TCA File -- .S$BGTCA.

The background TCA file, .S$BGTCA, is similar to the foreground TCA file. This file is used for communication between the batch/background SCI and its command processors. In addition, the background TCA file is used for passing synonyms and parameters to tasks which are executed through QBID. The QBID task "freezes" the foregound TCA image on record "I" of the foreground TCA file (where "I" is the terminal number) onto record "I" of the background TCA file when the background task bid request is made through the background request manager. The background TCA file is accessed through the global LUNO found in byte SCA$L4 of the SCA module.

10.2.5 SVC OVERHEAD ANALYSIS.

The following four subsections detail the use of DX10 SVCs, primarily for I/O, which are necessary for the execution of a typical .BID or .OVLY program. The .BID and .OVLY cases assume a task or overlay which accesses the TCA for parameters or synonyms and generates a listing on the terminal local file. An analysis is presented in paragraph 10.2.5.5.

10.2.5.1 .BID SVC Overhead for Foreground SCI990.

Table 10-1 shows the SVC overhead incurred by SCI990 while processing a .BID command. In the timing estimates, the Bid Task SVC cost includes the overhead for End Task processing. The overlay for the Open-Extend of the TLF is assumed to be on disk.

Table 10-1 .BID SVC Overhead for SCI

Routine	SVC	Time (Msec)	Disk Accesses
XBID	01 - Close the terminal	4.4	0
S$PTCA	00 - Open .S$FGTCA	4.7	0
	0C - Write direct, 1 record	11.4	1
	01 - Close .S$FGTCA	5.9	0
S$BID	2B - Bid task	36.2	2
S$GTCA	00 - Open terminal	4.5	0
	00 - Open .S$FGTCA	4.7	0
	0A - Read direct, 1 record	11.3	1
	01 - Close .S$FGTCA	5.9	0
S$OPEN	12 - Open-extend TLF	25.9	2
	TOTALS	119.6	6

10.2.5.2 .BID SVC Overhead In The Task Being Bid.

Table 10-2 shows the SVC overhead incurred by a task being bid through .BID. In the timing estimates, the (91) LUNO assignment to the TLF requires no disk accesses because the SCI task has a previously assigned LUNO attached to the file. This also reduces the timing from about 42 msec to 17 msec. The (12) open-extend estimate assumes that the overlay is in memory and the TLF is empty. If it is not empty, the estimate would be (18, 2, 1). The TLF is closed during task termination. The overhead for task termination is assumed in the Task Bid in paragraph 10.2.5.1.

Table 10-2 Overhead in the Bid Task

Routine	SVC	Time (Msec)	Disk Accesses
S$NEW	17 - Get Bid Parameters	.2	0
S$GTCA	00 - Open .S$FGTCA	4.7	0
	0A - Read direct, 1 record	11.3	1
	01 - Close .S$FGTCA	5.9	0
S$OPEN	91 - Assign LUNO to TLF	17.0	0
	12 - Open-Extend TLF	9.7	0
S$PTCA	00 - Open .S$FGTCA	4.7	0
	0C - Write direct, 1 record	11.4	1
	01 - Close .S$FGTCA	5.9	0
	TOTALS	70.8	2

10.2.5.3 .OVLY SVC Overhead for SCI990.

Table 10-3 shows the overhead incurred by SCI in executing an .OVLY command. Since an overlay is part of the SCI task, the TCA is available in memory and the TLF is already open.

Table 10-3 .OVLY SVC Overhead for SCI

Routine	SVC	Time (Msec)	Disk Accesses
S$OVLY	14 - Load Overlay	19.2	2

10.2.5.4 .OVLY SVC Overhead in the Overlay.

For overlays, S$GTCA and S$RTCA access the TCA directly in memory and require no I/O. The TLF is not actually opened and closed by S$OPEN and S$CLOS, so the only TLF I/O overhead is in the actual write SVCs, which are ignored in this analysis. Therefore, no overhead is incurred by the overlay being bid through an .OVLY command.

10.2.5.5 Analysis.

The assumptions made for this analysis are neither best nor worst case and are probably typical. It is further assumed that the cost of a disk access is approximately 100 msec for a DS31/32 disk drive and 60 msec for a DS25/50 disk drive. The SVC overhead for a typical .BID is then about 986 msec (114.9 + 70.8=185.7 msec, 6+2=8 disk accesses). The SVC overhead for the same program executed as an .OVLY is then 219 msec (19.2 msec, 2 disk accesses). The overhead for writing lines to the TLF is the same in both cases and is ignored, as is all other processing done by the program. For short functions, a .BID costs four

times as much as an .OVLY in system overhead and terminal response time. In both cases, the response time is on the order of a second or less.

10.3 BACKGROUND RESOURCE MANAGER

The Background Resource Manager (BRM) manages the background resources of the DX10 system command interpreter. These resources include the background task execution facility (QBID) and the output queuer (OQUEUE). BRM polls the SCA for background service requests from user terminals and bids the appropriate program to handle the request. It does not service requests itself and is not aware of the meaning of the requests it manages. Adding background services may be accomplished by adding background tasks to the system.

10.3.1 STRUCTURE OF BRM.

The BRM program consists of three modules, a task and two procedures. The first procedure is the system communication area (SCA), which contains the SCA entries which are polled for service requests. The second procedure is the background communication area (BCA) through which BRM communicates with QBID and OQUEUE. The task contains the actual BRM code and data. Both the SCA and BCA are "dirty" shared procedures. The SCA is described in the earlier discussion of SCI990. The BCA is described in paragraph 10.3.3.

10.3.2 CALLING SEQUENCE.

The BRM must be placed into execution before any SCI background service request is made through the SCA. This is normally done by bidding BRM from the DX10 system restart task. BRM then waits in a time delay for a service request.

A service request is made by placing an opcode value in the SCA$FO (FG SCI opcode) or SCA$BJO (BG SCI opcode) field of a terminal's SCA entry and executing an Activate Time Delay Task SVC to wake up the BRM. The run ID of the BRM task is initialized in byte SCA$L2 of the SCA. The requesting SCI then enters a time delay loop waiting for the SCA$FO (or SCA$BO) field to clear.

Background service request opcodes are byte values consisting of two hexadecimal digits. The first digit identifies the program that processes the request. The second digit specifies a particular operation. BRM uses only the first digit to determine the task ID to bid.

Specific opcode values are documented in the QBID and OQUEUE descriptions (paragraphs 10.4 and 10.5).

10.3.3 BACKGROUND COMMUNICATIONS AREA (BCA).

The BCA consists of two parallel vectors, the TASKID and BUSY tables, which are used for communication with the background request handler tasks. The first digit of a service request opcode is used to index these vectors. The TASKID vector maps the opcode into a DX10 task bid ID. The BUSY vector informs the BRM whether the indicated task/opcode is currently in execution or must be bid to handle the request. The BRM sets the busy flag when it successfully bids the task. The task resets it when it terminates.

Entry 0 of each vector is meaningless since an opcode field value of 0 indicates no request.

10.4 QUEUED TASK BID HANDLER (QBID)

QBID supervises the enqueueing and bidding of background tasks. In this context, Background (abbreviated BG) means that a task execution is to be initiated at a terminal via the .QBID SCI language primitive. Such tasks are then managed by the user indirectly through commands that access the QBID program.

10.4.1 STRUCTURE OF QBID.

The QBID program consists of three segments. Two shared "dirty" procedures, the SCA and BCA, are used for communication with the background resource manager and the command interpreter making a QBID service request. These procedures are described in the documentation for the background resource manager and SCI.

The task segment of the QBID program consists of several modules which contain the QBID code and data. The code is organized as a supervisor and four major components. The supervisor executes the four routines BOOKIE, POLLER, BIDDER, and WAITER repeatedly until WAITER determines that no work remains.

The bookkeeper routine BOOKIE keeps track of the status of all tasks being managed by QBID. Its principal duties are: .Try to remove blocks from blocked tasks .Remove queue entries of expired tasks .Calculate the current level of background task activity

Blocked tasks are those that cannot be bid at a particular time but are expected to be bidable later. Examples are tasks that are not replicable but are currently in execution. Expired tasks are those that have been successfully bid and have subsequently terminated.

The POLLER routine periodically examines the SCA for QBID service request opcodes. Opcodes in the range >10 - >1F are handled immediately by invoking the appropriate routine:

Opcode	Routine	Purpose
10	BUILDQ	Build a queue entry
11	STATUS	Check status of queued task
12	KILLQT	Kill a queued task
13	DBID	Bid task in halted state

The BIDDER routine attempts to bid tasks waiting on the background task queue within the constraints of the sysgen-imposed threshold count. Tasks are bid only if the current background activity level count is not exceeded. Candidate tasks on the queue must not be currently executing or marked as blocked. Bid attempts that fail because the specified task is not replicable or because the system table area is full cause the task to be marked as blocked.

The WAITER routine decides whether QBID has further work to do. If not (i.e., the queue is empty), it informs the BRM via the BUSY vector in the BCA that is terminating and does so. If there is more work to do, WAITER executes a time delay SVC and exits. The supervisor then repeats the execution of the four primary routines.

Table 10-4 shows the structure of QBID as a table of direct subroutine calls.

Table 10-4 QBID Subroutine Call Table

Calling Routine	Routines Called
QBID	BIDDER,BOOKIE,POLLER,WAITER
BIDDER	--
BOOKIE	TSTATE
POLLER	BUILDQ,DBID,KILLQT,STATUS
WAITER	--
BUILDQ	TCASVC
DBID	BUILDQ
KILLQT	STATUS
STATUS	TSTATE
TCASVC	--
TSTATE	--

10.4.2 DATA STRUCTURES.

The following paragraphs describe the data structures used by QBID routines.

10.4.2.1 System Communication Area (SCA).

The SCA is an area of memory shared as a "dirty" procedure by all command interpreters, the BRM, and the background request handlers (QBID,OQUEUE). The SCA contains a table of global system data, such a global LUNOs and task bid IDs, plus a table called an SCA entry for each terminal that will use SCI990. The SCA is documented in the description of SCI990 (see paragraph 10.2).

10.4.2.2 Background Communication Area (BCA).

The BCA consists of two parallel vectors, the TASKID and BUSY tables, which are used for communications between the BRM, QBID, and OQUEUE. The first digit of a service request opcode is used to index these vectors. The TASKID vector maps the opcode into a DX10 task bid ID. The BUSY vector informs the BRM whether the indicated task/opcode is currently in execution or must be bid to handle the request. The BRM sets the busy flag when it successfully bids the task. QBID resets it when it terminates.

10.4.2.3 Task Queue Entry.

Each task to be bid by QBID has an associated queue entry, which is created by QBID in its own task area. Each queue entry describes a task, information necessary to bid the task, and the current execution status. The fields of a queue entry are labeled and defined as follows:

```
QNEXT    EQU   0        Address of next queue entry
*                       -- QNEXT must be zero --
QSTAT    EQU   2        Status of queue entry
QTERM    EQU   3        Terminal ID
QUSRID   EQU   4        User ID
QBTASK   EQU   10       Task bid ID
QLUNO    EQU   11       LUNO
QCODE    EQU   12       Code value
QRTASK   EQU   13       Task Run ID
QTIME1   EQU   14       Time place in queue
QTIME2   EQU   18       Time task was bid
*
QESIZE   EQU   22       No. of bytes in a queue entry
```

The status byte (QSTAT) is divided as follows:

Bit	
0	1 = Task has been bid
1	1 = Task is blocked (task ID in use)
2	1 = Task is running
3-7	Reserved

10.4.3 CALLING SEQUENCE.

QBID Is always bid by the background resource manager (BRM) task in response to a background service request from a terminal command interpreter (FG or BG). SCA opcodes in the range 10 through 1F are directed to QBID. The currently defined opcodes are:

10	Enqueue a background task bid (.QBID)
11	Check status of an enqueued task
12	Kill an enqueued task
13	Execute a BG task in debug mode (.DBID)

Associated with these opcodes are fields within the SCA entry that specify parameter values. These are:

SCA$BC	BG "CODE" Value
SCA$BL	BG Program File LUNO
SCA$BS	BG Status
SCA$BT	BG Task ID
SCA$FE	Returned error code (FG)
SCA$TI	Terminal ID

10.4.4 FILES.

The only files accessed by the QBID task are the foreground and background TCA files, .S$FGTCA and .S$BGTCA, via the LUNOs specified in the SCA. When a task is enqueued for BG execution, the FG TCA image for the terminal is copied from the FG TCA file to the BG TCA file. The TCA image is a single record in each case. The record number is the same as the terminal number. The TCA image is "frozen" in this manner so that the parameters specified on the corresponding .QBID language primitive will be available to the enqueued task when it executes. Also, all synonyms defined at the terminal will be available to the task.

The TCA image is described in detail in the SCI990 discussion (see paragraph 10.2)

10.4.5 ERROR CODES.

The following error codes are returned in the SCA$FE field of the SCA entry for the terminal making the QBID request.

Code	Meaning
00	No error - request serviced
01	Unable to allocate a queue entry block
02	Unable to access the TCA
02	BG already pending (should be caught by the command interpreter first)
03	Bid SVC failed for ".DBID" request
80	Unknown SCA opcode
FF	No queue entry found (Status)

10.5 QUEUED OUTPUT HANDLER (OQUEUE)

OQUEUE supervises the enqueuing of files and output to devices and of messages to and from terminals. Files are queued up for output by name, rather than by copying the named file to a spooled data area. Any number of files may be queued for any number of devices. OQUEUE also enqueues messages for transmission to terminals.

10.5.1 STRUCTURE.

The OQUEUE program is divided into two tasks, OQ$COPY and OQ$MGR, each of which consists of three segments. Both tasks share "dirty" procedures, the SCA and BCA which are used for communication with the SCI990 task making an OQUEUE service request. These procedures are described in the documentation for the background resource manager and the command interpreter.

The task segment of the OQ$MGR program consists of several modules. The code is organized as a supervisor and four major components. The supervisor executes the three routines POLLER, BOOKIE, and WAITER repeatedly until WAITER determines that no work remains.

The POLLER routine periodically examines the SCA for OQUEUE service request opcodes. Opcodes in the range >20->2F are handled immediately by invoking the appropriate routine:

Opcode	Routine	Purpose
20	BUILDQ	Build a queue entry
21	STATUS	Show output status
22	KILLDV	Kill output at a device
23	HALTIT	Halt output to a device
24	RESUME	Resume output to a device
2E	SMSG	Send a message
2F	RMSG	Receive a message

The bookkeeper routine BOOKIE tracks current I/O and message activity, ensuring that each queue entry is ultimately processed. BOOKIE calls MBOOKY to check each pending message destination against each currently logged-in SCA entry. When a terminal for which a message is pending is discovered to be activated, the message pending flag in the background

status field (SCA$BS) of the SCA entry is set. BOOKIE then calls OBOOKY, which attempts to assign an output processor to a file queue entry waiting for access to an output device, and assigns an OQ$COPY task to each device for which output is queued. Thus, queued output may be directed to many devices simultaneously.

OQ$COPY is a replicative task that copies files to a device. The file and device must have been assigned global LUNOs prior to execution of OQ$COPY. OQ$COPY is bid by the BOOKIE portion of OQ$MGR. The Device Status Table (DST) in the BCA contains all the information necessary to copy file records to the device.

The WAITER routine decides whether OQUEUE has further work to do. If not (i.e., the queues are empty), it informs the BRM via the BUSY vector in the VCA that it is terminating and does so. If there is more work to do, WAITER executes a time delay SVC and exits. The amount of the time delay is determined dynamically by WAITER. After the time delay, WAITER exits. The supervisor then repeats the execution of the four primary routines.

Table 10-5 shows the structure of OQUEUE by its subroutine call linkages.

Table 10-5 OQUEUE Subroutine Call Table

Calling Routine	Routines Called
OQ$MGR	BOOKIE,POLLER,SLICER,WAITER
BOOKIE	MBOOKY,OBOOKY
POLLER	BUILDQ,HALTIT,KILLDV,RESUME,RMSG,SMSG,STATUS
WAITER	-
MBOOKY	-
OBOOKY	GETFET
BUILDQ	GETBLK,GTCA,S$PARM
HALTIT	FINDDV,OPARMS
KILLDV	FINDDV,OPARMS
RESUME	FINDDV,OPARMS
RMSG	GTCA,S$SETS,PTCA
SMSG	GETBLK,GTCA,S$PARM
STATUS	GTCA,S$PARM,Q$OPEN,Q$WRIT,Q$WEOL,WRSTAT,Q$CLOS
WRITER	OPENDV,OPENFL,COPYFD,CLOSEF,CLOSED,WSTOP
GETFET	-
GETBLK	-
GTCA	TCHELP
S$PARM	-
FINDDV	-
OPARMS	GTCA.S$PARM
S$SETS	-
PTCA	TCHELP
Q$OPEN	CLRBUF
Q$WRIT	-
Q$WEOL	CLRBUF
WRSTAT	Q$WRIT,Q$WEOL
Q$CLOS	-
OPENDV	SVC$R2,OPN$R2
OPENFL	SVC$R1
COPYFD	SVC$R1,FORMAT,SVC$R2,OPN$R2,WAIT
CLOSEF	SVC$R1
CLOSED	SVCR2,WAIT
WSTOP	-
TCHELP	-
CLRBUF	-
SVC$R2	WAIT
OPN$R2	SVC$R2
SVC$R1	WAIT
FORMAT	-
WAIT	-

10.5.2 DATA STRUCTURES.

The internal data structures used by OQUEUE routines are described in the following paragraphs.

10.5.2.1 System Communication Area (SCA).

The SCA is an area of memory shared (as a "dirty" procedure) by all command interpreters, the BRM, and the background request handlers (QBID, OQUEUE). The SCA contains a table of global system data, such as global LUNOs and task bid IDs, plus a table called an SCA entry for each terminal which may use SCI990 (see paragraph 10.2).

10.5.2.2 Background Communication Area (BCA).

The BCA consists of two parallel vectors, the TASKID and BUSY tables, which are used for communication between the BRM, OBID, and OQUEUE. The first digit of a service request opcode is used to index these vectors. The TASKID vector maps the opcode in a DX10 task bit ID. The BUSY vector informs the BRM whether the indicated task/opcode is currently in execution or must be bid to handle the request. The BRM sets the busy flag when it successfully bids the task. OQUEUE resets it when it terminates.

10.5.2.3 Output Queue Entry.

Each output entry describes a file and a device to which the file is to be copied. The fields of the output queue entry are labeled and defined as follows:

```
QNEXT    EQU  0              Address of next queue entry
*                            -- QNEXT must be zero --
QSTAT    EQU  2              Status of queue entry
QTERM    EQU  3              Terminal ID
QUSRID   EQU  4              User ID
QTIME1   EQU  10             Time placed in queue
QTIME2   EQU  14             Time task was bid
QDLMAX   EQU  18             Max length of message name
QULMAX   EQU  18             Max length of message user ID
QDNLEN   EQU  19             Actual length of device name
QUILEN   EQU  19             Actual length of message user ID
QDVNM    EQU  20             Device access name/message user ID
QALMAX   EQU  26             Max length of access name
QTLMAX   EQU  26             Max length of message text
QANLEN   EQU  27             Actual length of access name
QMTLEN   EQU  27             Actual length of message text
QACNM    EQU  28             File access name/message text
*
QASIZE   EQU  80             Max No. of bytes in a file name
QDSIZE   EQU  6              Max No. of bytes in device name
QESIZE   EQU  QACNM+QASIZE   No. of bytes in queue entry
```

The status byte (QSTAT) is subdivided as follows:

Bit	
0	1 = Output has begun
1	1 = ANSI Format
2	1 = I/O completed
3-7	Reserved

10.5.2.4 File Environment Table.

A file environment table (FET) is assigned to each output device for which files are queued. The FET contains the data specifying the file and device, a workspace, and all data and buffers used by a WRITER routine to copy records from the file to the device. The FETs and WRITER are analogous to DX10 tasks and a shared procedure. The fields of a FET are labeled and defined as follows:

```
WFETRO    EQU  0           32 BYTE WORKSPACE
WNFET     EQU  32          @ NEXT FET IN FET QUEUE
WPC       EQU  34          @ NEXT ENTRY INTO WRITER
WQNTRY    EQU  36          @ INPUT FILE QUEUE ENTRY
WLINES    EQU  38          # LINES LEFT ON PAGE
WDVTYP    EQU  40          DEVICE TYPE CODE
WSTAT     EQU  41          STATUS
WDEVNM    EQU  42          DEVICE NAME
WSTACK    EQU  46          RETURN STACK
WPRBI     EQU  54          INPUT PRB
WPRBO     EQU  94          OUTPUT PRB
WBUFF1    EQU  134         BUFFER 1
WFSIZE    EQU  WBUFF1+140  END OF FET
```

The FET status byte (WSTAT) is subdivided as follows:

Bit	
0	1 = Halt output immediately
1	1 = Halt output at EOF
2	1 = Kill output of current file
3	1 = Kill all output at device
4-7	Reserved

10.5.3 CALLING SEQUENCE.

OQUEUE is always bid by the background resource manager (BRM) task in response to a background service request from a terminal command interpreter (FG or BG). SCA opcodes in the range of >20 through >2F are directed to OQUEUE. The currently defined opcodes are:

>20	Release file to queue
>21	Show output status
>22	Kill output to a device

>23	Halt output to a device
>24	Resume output to a device
>2E	Send message
>2F	Receive message

Associated with these opcodes are fields within the SCA entry that specify parameter values. These are:

SCA$BS	BG Status
SCA$DV	Terminal device name
SCA$FE	Returned error code (FG)
SCA$TI	Terminal ID
SCA$UI	User ID

10.5.4 FILES.

OQUEUE uses the TCA file and a listing file, as described below.

10.5.4.1 TCA File.

The foreground or background TCA file record corresponding to the number of the terminal making the OQUEUE service request is read (by module OQ$TCA) so that the opcode handling routines can access the parameters in the NCT.

The TCA image is described in detail in the documentation for the system command interpreter program, SCI990.

10.5.4.2 Listing File.

The Show Output Status function has a listing file as a parameter. This listing file is normally the foreground or background terminal local file (TLF), at the option of the SCI PROC which invokes OQUEUE. The module OQ$TLF includes the routines used to write to the listing file.

10.5.5 ERROR CODES.

The following error codes are returned in the SCA$FE field of the SCA entry for the terminal making the OQUEUE request.

Code	Meaning
00	No error - request serviced
01	No queue block available
80	Unknown SCA opcode
FA	Invalid argument
FD	NCT error (internal)
FD	TLF Error
FD	TCA error
FF	Queue entry not found

APPENDIX A

SYSTEM CRASH ANALYSIS

A.1 GENERAL

When the DX10 operating system detects a system failure, it displays an error code on the front panel lights and idles the CPU, as described in the DX10 Operating System Production Operation Manual. The first step that must be taken in order to analyze the system crash is to dump memory to the predefined crash file on disk. This is accomplished by pressing the HALT and then the RUN buttons on the front panel. The next steps in analyzing the crash are: reboot the system (as explained in the DX10 Operating System Production Operation Manual); bid SCI at a terminal; and execute the crash analyzer, ANALZ, using an XANAL command.

A.2 OPERATING PROCEDURE

When ANALZ is activated, using the XANAL command, five parameters are prompted. The parameters are:

*CONTROL ACCESS NAME:	the name of the file or device from which ANALZ expects commands. The default is ME.
*LISTING ACCESS NAME:	the name of the file or device to which ANALZ should write its output.
*ANALYZE RUNNING SYSTEM?:	answer YES to analyze the currently running system, (rather than a crash dump). If the dump is to be analyzed, answer NO. The default is NO.
*DISK DEVICE NAME:	the name of the disk unit on which the crash file is written. The default is DS01.
*CRASH FILE NAME:	the name of the file containing the crash dump. The default is S$CRASH.

If ANALZ is running in batch mode (i.e., a file or sequential input device was specified as the control access name), each input value or command must start in column one of a separate record (card). Use of batch input to ANALZ allows the user to keep a standard ANALZ command stream on file or cards which can be easily and quickly executed after every system crash.

A.3 COMMANDS

The commands given to ANALZ cause it to select portions of the memory dump on the crash file (or actual memory, if the running system is being analyzed) and write them to the listing device in a formatted form. Blocks of memory are written eight words per line, preceeded by the address of the first word. All numbers are in hexadecimal. An abbreviated ASCII representation of the eight words is written to the right of the hexadecimal representation. Byte values that do not represent a character are written as periods (".").

Table A-1 shows the ANALZ commands and the action each performs.

Table A-1. ANALZ Commands

Command	Action
GI	List general information about the crash
TS	List the task states of all currently executing tasks
SS	List memory images of the system structures
MM	List a memory map
AQ	List a representation of the four active queues
PQ	List a representation of the other system queues
TR	List the workspace registers for all tasks in memory
TA	List memory images of every task area in memory
AL	Do all of the above, in the order given
DM	List a specific area of memory
DI	List disk information
QU	Terminate execution

Normally, the user needs only execute the AL command to obtain enough information about a crash. The following paragraphs describe in more detail the results of some of the commands. The commands are described in the order in which they are executed by the AL command. Useful information is also provided for determining the reason for the crash.

A.3.1 GENERAL INFORMATION (GI) COMMAND.

The GI command lists general information about the crash.

A.3.1.1 Crash Code.

The first entry is the system crash code that was displayed on the front panel when the crash occurred, as well as an English translation of the code (see the DX10 Operating System Error Reporting and Recovery Manual for a description of the crash codes).

A.3.1.2 Executing Task.

The second entry is the address of the task status block (TSB) of the task that was executing when the crash occurred. When this value is zero, the crash occurred within the operating system, probably during a scheduling cycle.

A.3.1.3 Location of Failure.

This entry is the address at which the crash routine was called. In some cases, this entry points to the exact location of the crash. However, in most cases, this value is the location of a common crash point that is entered from any of several locations.

A.3.1.4 Status Register.

This entry lists the value of the status register when the crash occurred. The status register information is valuable when the crash code is >20. The last four bits (last digit in the entry) of the status register is the interrupt mask. When the interrupt mask value is in the range >2 - >E, the crash was caused by an illegal interrupt. When the interrupt mask value is 1, the crash was caused by one of the task error states occurring within the system or system task.

A.3.1.5 System Variables.

A set of system variables is printed after the status register value. Most of these do not contain useful information. Three of the variables may be of some importance. They are:

MEMSIZ	Total amount of memory in the system expressed in beets (32-byte blocks). This is useful in determining the amount of memory space available for system and user tasks.
NUMDEC	Negation of the number of time units

since the last scheduling action. If this number is unusually large, a task may be in a loop with the scheduler suspended or the scheduler itself may be in error.

RSTRSW — Flag that tells if the system has completed initialization. If the flag is set to -1 (>FFFF), the crash occurred because the system was not initialized.

A.3.1.6 Fixed and Runtime Task IDs.

These entries are bit maps of all fixed and runtime task IDs. They are generally of no value in a crash analysis.

A.3.1.7 System Patch Area.

A dump of all out-of-line patches that have been applied to the system by the MEMRES patch file is listed. This is used to verify that all out-of-line patches have been applied to the system successfully. The beginning address of the patch area should be equal to the value assigned to S$PAT. The last five lines of the patch area will give the revision letter of the release. The revision letter should be checked to make sure that all recent patches have been applied.

A.3.1.8 Monitor Registers and Stack.

The monitor registers are the registers of the executing task at the time of the crash. R0 of these registers will often contain the code of an error that caused the task to abort the system. The error codes are as follows:

1	Memory Parity Error	
2	Illegal Instruction	
3	TILINE Time Out	
5	Mapping Violation	
A	Execute Protect Violation	(990/12)
B	Write Protect Violation	(990/12)
C	Stack Overflow	(990/12)
D	Hardware Breakpoint	(990/12)
E	12 Millisecond Clock	(990/12)
F	Arithmetic Overflow	(990/12)

R10 is the stack pointer for the task. A segment of the stack will be printed following the workspace registers. The stack pointer can be used to trace back through the stack to determine if a lower tier routine has passed an error code.

A.3.1.9 Interrupt and XOP Vectors.

The hardware interrupt and XOP trap vectors should be examined to verify that they are intact. Since these values reside in the first locations of physical memory, they are often destroyed by a system or privileged task that branches to memory location zero. Usually, the locations that are destroyed are locations 0-3 (power-up interrupt) or locations 1A-1F (interrupts six and seven). Location 0 is often loaded with a bad value when a data word required by a task is not specified. Locations 1A-1F are destroyed when a task executes a BLWP instruction

to location 0. When the BLWP instruction occurs, the return context of the task will be in locations 1A-1F. When a >20 crash occurs, and the interrupt mask indicates a defined interrupt (mask value minus one), the interrupt trap values should be checked to determine if they are within the proper address range. Except for interrupt levels 0, 1, 2, and 5, the workspace pointer and program counter for each trap should point to about the same locations.

A.3.1.10 Internal Workspaces.

The last data printed are the workspaces for the clock processor, the interrupt 2 processor, and the SVC processor. These routines are entered through context switches, so the return context will be found in registers R13-R15 of these workspaces. The clock workspace will contain the location in the executing task where the last clock interrupt occurred. The SVC workspace will contain the location of the last supervisor call. These two locations can sometimes help determine where a task was at the time of the crash. The interrupt 2 workspace contains diagnostic information about a >20 crash. R13-R15 contains the context of the crash within the system. R1 contains the error code (listed above). When looking at the saved status register for interrupt 2 (R15), check to see if bit 8 is set or reset. If bit 8 is set, the crash occurred in task driven code. If bit 8 is reset, the crash occurred in system code. It may be necessary to dump memory around the location pointed to by R14 to check the listing to determine if the code has been modified.

A.3.2 TASK STATE (TS) COMMAND.

The TS command lists the most commonly referenced terms from the TSBs of the tasks in the system. The list contains, for each entry, the task ID, the task context at the last time the task was scheduled or performed a supervisor call, the current state of the task, the task flags, and the TSB address. The task status block list follows the listing of commonly referenced terms. The task flags contain useful information in bit 5: when this bit is set, the task has been rolled out to disk; when this bit is reset, the task is in memory. By examining this bit for each task, the user can determine which tasks were in memory and may be associated with the crash.

A.3.3 SYSTEM STRUCTURES (SS) COMMAND.

The SS command lists the TSB, PSB, PDT, FCB, and LDT information from the system. The system templates and/or data structure descriptions are helpful when examining the SS command lisitng.

A.3.3.1 Task Status Block (TSB).

The TSB listing gives the entire TSB for every task in the system. The TSB contains all the information about the task that is used by the system. The most important fields in the TSB listing are the end action address, the second flag word, the task diagnostic field, and the task map file.

The end action address entry contains the starting address of the memory task segment. This parameter is helpful when partial links of the system segments are available (provided with DX10 source). By taking the end action address entry, and locating where the task segment is in one of the partial links, the location of other system parts can be found. The length of the root segment can be found from the partial links. The starting address of DMGR, which is the start of the memory resident task segment, can be located in the TSB listing. By subtracting the length of the root segment from the starting address of DMGR, the starting address of the root can be determined. The starting address of the root segment is useful in determining if the crash was caused by DX10.

The second flag word in the TSBs contains information about the task. If the task is being debugged, the first seven bits of the flag word contain information regarding the debugger; otherwise the first seven bits are set to zero. If the task is suspected as causing the crash, the flags should by verified with the templates, and TMRWT and the DEBUGGER modules should be examined for possible problems.

The diagnostic information field in the TSB contains the interrupt 2 values for a task that has taken end action. This field is helpful when analyzing a crash code of >27. The >27 crash is caused by TMEXT being non-zero when a task is killed. Often this occurs when a system task sets TM$EXT to suspend scheduling, then takes end action. Thus, even though a task may have a unique crash code, a >27 crash may occur because of these circumstances. The diagnostic information contains the context vector of the task at the time it was aborted. By taking these values, the listings can be examined to determine what conditions caused the abort (if the source listings are available). If the listings are not available, this information is useful in discovering a faulty module, if subsequent crashes occur within the same task area.

The task map file is also kept in the TSB. The map file consists of three segments, with each segment containing a limit and a base. When a memory management problem is being considered, or an unexplainable map violation occurred from the task, the task map file should be examined. To find the upper limit of a task, find the last limit register that is not equal to >FFFF and negate it. The task should be able to address memory locations up to that point. To check for a memory management problem or a TILINE time out problem, the location of each segment in physical memory should be determined. This is done by the following formulas:

Segment	Beet Address	Beet Length
1	B1	-L1/32
2	B2+(-L1/32)	(L1-L2)/32
3	B3+(-L2/32)	(L3-L2)/32

where the map file = L1, B1, L2, B2, L3, B3

The segment beet address can be checked against MEMSIZ (GI command listing) to see if the beet address exceeds the memory limit. Also, the segment can be checked on the time ordered list (MM command listing) if a memory management error is suspected.

A.3.3.2 Procedure Status Block (PSB).

The PSB list follows the TSB list. These entries contain the memory Beet address and the length of the segment in Beets. It may be necessary to verify the procedure locations in memory and check for their entry on the time ordered list (MM command listing). The procedures have a count of tasks that are attached to them. If the procedure is flagged as memory resident, then this count must always be at least one. There is no link to the attached task(s) in the PSB; this link is maintained by the tasks in the TSB.

A.3.3.3 Physical Device Tables (PDT) and Device Buffers.

The next items listed are the PDTs. The PDTs define every device in the system. The device buffers are shown with the PDTs; these are meaningful only if the device is marked assigned and busy in the PDT flag field. The first two words in the PDT contain the PDT link and the map file address and are not included in the PDT workspace. If a crash occurred when the system was in a DSR, the general location of the PC can sometimes be found by looking at the PDT workspace registers 5 and 6. These registers often contain either the interrupt entry vector or a BLWP vector used in the DSR. R5 contains the workspace pointer, and R6 contains the program counter.

All keyboard devices have a keyboard status block (KSB) as the last part of the PDT. The KSB contains a ring buffer for the characters being input. By looking at this buffer, the last 6 characters input

can be determined. Also, the KSB flags contain information about an SCI bid and whether SCI is active at the terminal.

The TILINE and diskette drive PDTs have an extension for each controller. The controller can have up to 4 drives associated with it. There is one PDT per drive. The extension is found on the first drive PDT. The extension indicates the number of drives that the controller services. The PDTs for TILINE devices also contain the last TILINE image that was sent to or from the device. These parameters are useful in isolating disk problems when the crash was related to the disk or the crash was forced by the user, when disk activity was 100% and the system was in a "hung" state.

A.3.3.4 File Control Blocks (FCB).

The FCB list gives all the files that were assigned at the time of the crash. Each disk drive has a list consisting of its VCATALOG entry and all the directories and files under it. Each FCB contains information the file name, pointers to its parent directory, disk allocation space, and logical and physical record sizes. This list is most useful on a running system when a drive cannot be released because of LUNO assignment. When the FCB LUNO count is zero, the FCB should be released. If an FCB is in memory with a LUNO count of zero (see system templates), the file is released by assigning a LUNO to that file, and then releasing it. This allows FUTIL another chance at removing the FCB. A disk will not be released until the only file opened under it is VCATALOG. The FCBs are also helpful with FUTIL and file management crashes. The FCB list shows all the files in the system at the time of the crash. The FCBs can then be examined to determine which file or files were involved in the crash.

A.3.3.5 Disk Partial Bit Maps.

Each disk that is installed has information concerning its ADU allocation printed in the dump. A 3-entry list is given for each sector of track 0 that contains disk allocation information. The list entries contain the size of the biggest ADU block for each sector, the size of the first block, and the size of the last block in that sector. There is room in each sector of the bit map to define 2032 ADUs. The disk bit map is printed next. Each entry has one word of overhead and 127 words of bit maps. The allocation information is of little use when analyzing a crash; it tells only if the disk is full.

A.3.3.6 Logical Device Tables (LDT).

The last listing printed by the SS command is the LDT list. The LDTs provide information about LUNO assignment in the system. Every task, station, and global LUNO is listed, with global LUNOs listed first, then station and task LUNOs listed in order of TSB. The LDT contains the LUNO number and the pointer to the associated file or device. When the LUNO is assigned, the TSB field will contain a non-zero value pointing to the TSB that owns the LUNO (see system templates). The LDTs are structured as a forward-linked list. The task LUNOs will point to the global LUNO list, either directly or through a list of task LUNOs. The LDT information is most helpful when using ANALZ on a running system. The LDT information can show why a LUNO is not being released and what it is pointing to. When examining the listing for a particular LUNO, the first LDT with the LUNO desired is looked at first. There may be more than one LDT with the same LUNO. The information in the first LDT is checked, and if this does not solve the problem, the next is examined. The LDT list can also be used in diagnosing file utility (FUTIL) problems in a crashed system.

A.3.4 LIST MEMORY MAP (MM) COMMAND.

The MM command lists information about the system mapping scheme, memory within the system table area, memory in the user task area, and system overlay segment usage. This set of entries should be examined whenever memory management might be involved with a crash.

A.3.4.1 System Memory Maps.

The system memory maps listing contains the current pointer to the map 0 map file, followed by all the system map files defined in the system. Each entry in the map file list contains seven words. The first word given is the overlay ID of the system image. This should correspond to one of the overlay IDs on the link map of the system. (The IDs listed in the dump are in hexadecimal, while the IDs in the link map are in decimal. This is used only if the link maps of the system are available). The remaining six words are the map file registers. These are the same as explained for the TSB listing. The first several entries are always the same, as follows:

1 File Management and Key Indexed Files
2 Memory Resident Tasks
3 User Common Area
4 I/O Common Area
5 Scheduler
6 Disk Device Service Routine (DSR)
7+ DSR for Each Remaining Device

The current map file (CURMAP) pointer is usually pointing to the scheduler map file, entry five in the map file list. When a crash

occurs and CURMAP is not pointing to the schedular map file, the problem is within one of the DSRs. Whenever a crash occurs and the interrupt mask (bits 12-15 of the status register) does not equal >F, CURMAP and the map file it points to should be checked. The addresses of the device DSR maps are found in the second word of the device PDT. When the PDT is dumped, the map file can be verified by checking that the second word of the PDT points to one of the listed map files. The memory for the DSR is examined by supplying the PDT address (not the TSB address) for the Dump Memory (DM) command.

A.3.4.2 System Table Area.

The system table area contains system usage information and a list of available memory. Crashes that are caused by too little table area can be determined by looking at the system table header. A >30 crash occurs when the system table overflows. The overflow can be determined by adding the starting address of the table (located in the header) and the total length of the table (also located in the header). If this sum is greater than the highest address allocated, a system table overflow has occurred. When the system crashes because of a table overflow, DX10 should be generated with a larger table area.

The remainder of the table area defines the free space chain of available memory. The header information gives the address of the first byte available. The remaining entries give the length of that block in bytes and an address pointer to the next available block. The list is ended with a zero pointer.

The system table area contains many structures of 10-100 bytes. When the system table area is heavily fragmented and approaching 100% utilization, some devices (such as diskettes) may not be able to obtain memory for the device buffer, and cause a table overflow crash. When this occurs, the size of the system should be evaluated and resizing the system by system generation should be considered.

Tasks with a coding error can cause a table overflow crash by using an unusual amount of table area. When this occurs, the table area listing contains many structures of the same type and size. Each entry in the system table lists its size in bytes. An example of this type of crash occurs when a COBOL program performs record locking on every record of a very large (20,000) record file. The system table area will have many RTL entries and will overflow. Programs that cause this type of crash should be rewritten.

A.3.4.3 User Memory Area.

The user memory area consists of three parts: the user memory header, the available memory list, and the time ordered list. All values in these tables are given in beets (32-byte blocks). The user memory is considered to be all of the physical memory that is not taken up by the memory resident segments of the operating system.

The user area header contains a pointer to the first available block of memory, the starting address of user memory, and the total length of user memory. No user task can be greater than the total length of memory. A crash normally does not occur from this unless the available memory space is less than 32K words or >800 beets. Also, no address on any of the three lists should be greater than the total length of memory (MEMSIZ). If this occurs, it indicates an error in memory management.

The available space list is a list of the blocks of memory available for user programs and their beet lengths. This is a linked list, with each entry containing a length word and a pointer to the next available block. These entries are kept in the first two words of the first beet of the block.

All user tasks, procedures, and blocking buffers that are not defined as memory resident are found on the time ordered list (TOL). Each of these segments have one beet of overhead that links them on the time ordered list. The TOL beet contains the block length, the associated structure address, the forward link, the backward link, and the segment type. Blocks on the TOL are located by looking at the pointers in the preceeding or succeeding block, which contain the beet address of the desired block. The beet address of each block is not found at the front of the entry.

The TOL is a circular list that is headed by a beet found at the start of memory. The segment types are as follows:

>FFFF	Header Entry
0	Blocking Buffer
1	Task Segment
2	Procedure Segment
3	Available Block

The header entry appears first on the list and should not be repeated. No buffer on the TOL should have a segment type of three.

The user area list is useful for diagnosing a crash that was forced, due to roll in/roll out deadlocks. Deadlocks can be caused by an area of memory being "lost" to the system; i.e., the pointer to a memory block may have been accidentally deleted.

Memory can be verified by checking all memory on the TOL and the available space list. Memory is verified by finding the first segment of the user area, then adding its length to its starting address to

find the second segment. That segment should be located on either the TOL or the available space list. Each segment of memory is then checked in similar fashion, until all of the segments are accounted for or the missing segment is found. If a segment is missing, this indicates a problem in memory management which should be handled by a Texas Instruments representative. When checking the available memory list, care must be exersized when the task loader was active at the time of the crash. The loader may have been in the process of delinking a block of memory.

Anytime an entry or pointer in the lists is greater than MEMSIZ, the entry is in error. This is usually caused by memory management allocating a second block of memory over the top of the first block allocated. The segment that overwrote the first should be examined for conditions that would cause memory management to put that segment at that location in memory.

NOTE

Do not try to look at the TOL on an active system. The memory chain will change while ANALZ is scanning the list, causing ANALZ to print meaningless data.

A.3.4.4 System Overlay Areas.

The system overlay area information gives the address of each overlay area, the status of the area, and the overlay ID. This listing is useful only if the crash occurred when the system was executing in one of the overlays. When the crash occurs in an overlay, this area is used to find the overlay. The system listings indicate the correct code that should be in the overlay area.

A.3.5 LIST QUEUES (AQ and PQ) COMMAND.

The AQ command is used to list the four active queues of the system. The PQ command is used to list other queues in the system. The active queues show the tasks queued for execution at the four priority levels of the system. The other queues include the file utility queue (FUTIL), the waiting on memory queue, the system log queue, and the device queues. These queues may be of importance during a system crash.

The queues listed by the PQ commands should not have more than three entries, except for the waiting on memory queue, where the length of the queue is a factor of system load versus system memory. When a queue is found to be unusually long, the queue processor should be checked to see if it is still active and the state it is in.

A.3.5.1 FUTIL Queue.

The file utility routine (FUTIL) services all create file functions and all assign LUNO functions to files and devices. The FUTIL queue is sometimes large because of the type of operation it is performing, such as creating large key index files. The queue processor should be checked when this occurs. Sometimes the FUTIL queue and processor will be blocked when performing I/O to a device. Some devices, such as line printers, have a long time-out associated with them. When the device is offline, and FUTIL is trying to assign a LUNO to the device, FUTIL is suspended until the time-out is performed. During that time, no other entry on the FUTIL queue can be serviced. Users may force a crash when this occurs, because it appears that the system is in a roll in/roll out deadlock. Some crashes are caused by a FUTIL roll in/roll out deadlock. Sometimes FUTIL will be rolled out of memory with a longer than average queue, and will not be able to obtain memory for roll in. When this occurs, it must be determined why FUTIL cannot get memory and what routine is supposed to handle that condition. The listings of the task loader area should contain the routine that handles FUTIL memory management.

A.3.5.2 Waiting on Memory Queue.

The waiting on memory queue contains the task status blocks (TSBs) of all tasks that are ready to execute but need memory. This queue grows in proportion to the number of tasks that are in the system and the size of available memory. There are times when no task is executing, and every task is on this queue. When this happens, the system deadlocks and crashes are usually forced. Conditions that lead to this are when tasks lock other tasks into memory without following the rules of the operating system. The most common occurrence of this is when a device tries to simulate TILINE I/O, which locks a task into memory, and the task does not set the supervisor control block (SCB) flag when the I/O completes. The SCB flag indicates to the system schedular that end-of-record processing must be performed, and that the task can be rolled out of memory. When this flag is not set, there is the possibility that the roll out algorithm will be caught in an endless loop. Other situations that lock tasks into memory and cause deadlocks are file management requests and alternate TSB servers.

Another situation that causes the waiting for memory queue to grow, and causes system deadlocks, is problems with the system disk. Sometimes the disk will go offline and roll in/roll out cannot be performed. The system log and system log queue should contain error messages when the system disk is in question.

A.3.5.3 System Log Queue.

The system log queue contains all the system log messages that are waiting to be written to the system log device. System log messages will be kept on the queue if the system log device has not been initialized or if the queue is getting more messages than the system

log device can handle. In this second case, the messages on the queue will often give a clue as to what went wrong with the system before the crash occurred (for example, when there are problems with the system disk).

A.3.5.4 Device Queues.

Each device has a queue list anchored in the PDT. This queue is a list of I/O requests made to that device. When a particular device is off-line or down, and the device has no time out value, the list may become long. When a crash is forced because a task would not come back from an I/O request, the device queues should be checked for the task requesting I/O and for unusual queue length. The system templates indicate where the queue pointer is in the PDT.

A.3.6 TASK REGISTERS (TR) COMMAND.

The TR command lists all of the workspace registers of the tasks that are in memory at the time of the crash. Workspace registers 10 and 11 are useful in determining where the task was executing at the time of the crash. Registers 10 and 11 give the stack location for the task and where the last return from a branch and link (BL) was in the task. These registers are used with the task listing to locate where the task was executing, and the data structures it was executing on. Register 10 points to the area containing information on routines called and parameters passed to them.

A.3.7 TASK AREA (TA) COMMAND.

The TA command lists two parts of memory: the task memory area and the system data base.

A.3.7.1 Task Area.

The task area listing shows the raw memory dumps of the task segments. The procedure segments are not shown; these must be listed with the dump memory (DM) command. The push/pop stack is usually kept in the task segment and will be listed by this command when it is. Many times it is necessary to determine if the task memory space has been altered. By checking the task area with a listing of the task, it can be determined if any task memory has been wiped out. If several words have been destroyed, it may be possible to determine which processor wiped out the code and what type of operation was being performed. File management and TILINE data transfers often cause destroyed code. By checking the SVC call blocks, the buffer areas of the transfer can be determined. An incorrect address in the SCBs will cause code to be wiped out. Another cause of destroyed code is caused by the TILINE device service routine not getting the right physical address of the output buffer. Common places where this is discovered is in the first 100 words of the system, or at addresses greater than >C000. The device service routines may have bugs when this occurs.

A.3.7.2 System Memory.

This listing is a dump of the system data base. This area is the first module of the operating system, with the largest part being the system table area. It is often necessary to examine data structures that are not listed in the above sections of the ANALZ dump, but are found in the system table area. Sometimes the table area is modified, similar to the modifications of the task area listed above. This part of the dump is important, because it lists the system data base without being restrained to data structures.

A.3.8 ALL (AL) COMMAND.

The AL command allows the user to do all of the functions listed above, in the order given, with one command. This is the most useful way of performing a crash dump analysis. It allows for easy referencing between different parts of the system without having to enter separate commands for each part to be analyzed. Two other commands are available for the analysis; these are listed below.

A.3.9 DUMP MEMORY (DM) COMMAND.

The DM command is used to list memory not shown by any of the above commands. The DM command can be used in three ways. The first is to give a task status block address and a relative address within the task. This gives a list of the task area not given by one of the above commands. The second method is is to give the address of one of the physical device tables for the device. This gives the memory available for the DSR. This second method also allows for the viewing the system root and I/O common area. The third method is to list absolute memory.

This requires a TSB address of zero, and a 21 bit absolute address. This method should be used when examining the time ordered list memory (i.e., the memory pointed to by entries on the time ordered list).

A.3.10 DISK INFORMATION (DI) COMMAND.

The DI command provides information on each file and directory of the system disk. This command is rarely used in a crash analysis. The information given by the DI command, for each file, is a follows:

* File Name: the name of the file.
* LRL: the logical record length of the file.
* PRL: the physical record length of the file.
* Size: the number of ADUs in the file.
* Address: the starting ADU address of the file.
* Logical EDM: the logical end of file address.
* Block EDM: the physical end of file address.

A.4 ANALYZING >20 AND >27 CRASHES

The most common types of crashes are >20 and >27 crashes. The following paragraphs give suggestions on how to start analyzing these crashes and what conditions to look for.

A.4.1 >20 CRASHES.

The >20 crash is caused by an illegal internal interrupt. In most cases it is an error interrupt within the system. The following steps indicate what should be looked for when a >20 crash occurs.

A.4.1.1 Status Register.

When a >20 crash occurs, the status register should be checked first. If the last 4 bits of the status register do not equal one, the crash is caused by an illegal internal interrupt. This indicates that a device interrupted at an interrupt level that is not defined, or that a device interrupted within an expansion chassis and the device position in that expansion chassis is not defined. The value of status register bits 12 through 15 plus one indicate the interrupt level that was taken. The hardware configuration and the system generation listing should be checked to determine if the interrupt level taken is a legal interrupt. If the interrupt is legal, then the interrupt workspace is checked for a bad chassis position interrupt. A pointer to that

workspace is at the beginning of memory at location interrupt level times four. Workspace register 9 will have the position value times 8 that caused the the interrupt. Divide the contents of R9 by eight, and check the hardware configuration to verify the chassis position.

A.4.1.2 Interrupt Two Workspace.

In most cases, the status will be >C001. This indicates a task error within the operating system. In the error interrupt workspace, register 1 will contain the task error code that caused the crash. These error codes are listed above under the description of the level 2 workspace. When the error code is found, the contents of registers 13, 14, and 15 show the location of the crash, and the status at the time of the crash. If bit 8 of workspace register 15 is set to one, the crash occurred in a user task; if it is set to zero, the crash occurred in a system task. The contents of register 14 is the program counter at the time of the crash. This code around this location should indicate what caused the crash. The workspace pointer is in register 13; the task workspace is used to check indexed addresses. The task listing should be consulted to determine what instructions were being executed at the time of the crash.

A.4.2 >27 CRASHES.

The >27 crash is the same as a >20 crash, except that the crash occurs within a system task. The definition of a >27 crash is "TM$EXT non-zero during kill task", which is caused by a task taking end action while it has suspended the scheduler. The crash occurs before the end action is taken. When this crash occurs, the value of ETSK shows the executing task at the time of the crash. The TSB for this task (found in the TSB list) has a diagnostic information field containing the error code and the context of the task at the time of the area. With this information, the >27 crash is handled the same as the >20 crash.

APPENDIX B

REGENERATING DX10, SCI, SDSMAC, & XLE FROM SOURCE

B.1 GENERAL INFORMATION

A DX10 Release 3.2 (or later) system with FORTRAN installed must be used when it is desirable to regenerate DX10. Because of the large volume of data required, the regeneration process normally requires a 4-disk system to complete the process. The disks required are:

System disk	-- Contains a DX10 system, 5000 contiguous sections of temporary space, and the FORTRAN compiler and runtime package.
Source disk	-- Contains source and object files.
Listings disk	-- Contains the source listings.
Build disk	-- Disk onto which to build the new system.
Batch listings disk	-- Contains the batch execution listings. Not necessarily a different disk.

The system disk must be a DS10 or larger disk. The source disk must be a DS25 or larger disk (64176 sectors required).

The listings disk must be larger than a DS25 to contain all the source listings (102543 sectors required). The assemblies may be done so that the listing disk capacity requirement may be reduced by changing the disk during the course of the assemblies. It is recommended that a DS25 disk be the minimum capacity disk used. The additional files for the microfiche require an additional 6624 sectors for a total of 109167 sectors.

The batch execution listings disk may be a DS31 or larger disk. The batch listings require 5082 sectors. These listings may go to the system disk, the listngs disk, or any other disk in the system (excluding the source disk).

A single magnetic tape drive is required if the magnetic tape disk build media is to be done.

To execute the batch stream named VOLSRC.BATCH.LINKBLD, the user must have a privilege code of 6 or 7.

B.2 ASSEMBLING AND COMPILING DX10

The DX10 source is logically divided into directories according to operating system function. These directories are described in this system design document. A batch stream is supplied for each operating system directory that will perform the conversion of source to object. These batch streams recompile or reassemble all source modules if the proper synonyms are correctly assigned. The required synonyms are:

Synonym	Value
VOLSRC	SYSBLD (volume name of source disk)
VOLOBJ	SYSBLD (volume name of object disk)
VOLLST	LIST3X (volume name of listing disk)
VOLSYS	?????? (volume name of executing system disk)
VOL$$BAT	LIST3X (volume name of batch listings disk)

The source disk supplied by Texas Instruments is named SYSBLD. Normally, the listing disk contains the release level in the volume name when the procedure is executed by Texas Instruments personnel. The names have been LIST30 for Release 3.0 and LIST31 for Release 3.1. The name may be changed without impacting the procedure.

The following is a list of the batch streams to reassemble/compile DX10:

```
SYSBLD.BATCH.ASM.ANALYZ
SYSBLD.BATCH.ASM.DEBUGR
SYSBLD.BATCH.ASM.DEVDSR
SYSBLD.BATCH.ASM.DSCBLD
SYSBLD.BATCH.ASM.DSCMGR
SYSBLD.BATCH.ASM.DX10
SYSBLD.BATCH.ASM.DXMISC
SYSBLD.BATCH.ASM.DXUTIL
SYSBLD.BATCH.ASM.FILMGR
SYSBLD.BATCH.ASM.FUTIL
SYSBLD.BATCH.ASM.GEN990
SYSBLD.BATCH.ASM.KIFILE
SYSBLD.BATCH.ASM.MEMMGR
SYSBLD.BATCH.ASM.NOSHIP
SYSBLD.BATCH.ASM.PGFILE
SYSBLD.BATCH.ASM.SYSTSK
SYSBLD.BATCH.ASM.TSKMGR
SYSBLD.BATCH.ASM.UTCOMM
SYSBLD.BATCH.ASM.UTDIRP
SYSBLD.BATCH.ASM.UTDXTX
SYSBLD.BATCH.ASM.UTGENR
SYSBLD.BATCH.ASM.UTSVC
```

Assign the synonyms and execute each of the batch streams with the XB command. The approximate run time is 13 hours.

B.3 ASSEMBLING SCI990

To assemble SCI990, execute the following batch stream with the same synonyms assigned as required for DX10 reassemblies:

SYSBLD.SCI990.BATCH.ASM

This batch stream runs approximately 3.25 hours.

B.4 ASSEMBLING SDSMAC

To assemble SDSMAC, execute the following batch stream with the same synonyms assigned as required for DX10 reassemblies:

SYSBLD.SDSMAC.BATCH.ASM

This batch stream runs approximately 2.5 hours.

B.5 TRANSLITERATING THE LINK EDITOR

To transliterate the link editor, execute the batch stream named:

SYSBLD.LINKER.UTILITY.INSTALL.

This installs the transliterator required to translate and assemble the link editor source. Use the same synonyms assigned for DX10 reassemblies. When this batch stream has completed, execute the batch stream named:

SYSBLD.LINKER.BATCH.ASM

This batch stream runs approximately 2.2 hours.

B.6 LINK EDITING DX10

Assign the following synonyms and then execute the following batch streams:

Synonym	Value
VOLSRC	SYSBLD (volume name of source disk)
VOLOBJ	SYSBLD (volume name of object disk)
VOLBLD	REL32 (volume name of disk to build)
VOLSYS	?????? (volume name of executing disk)
VOL$$BAT	LIST3X (volume name of batch listings disk)

Batch Stream	Function
SYSBLD.BATCH.DXLINKS	Pre-links and links of DX10 parts
SYSBLD.BATCH.UTLINKS	Pre-links and links of all utilities
SYSBLD.BATCH.GENPARTS	Compress object for SYSGEN parts
SYSBLD.SCI990.BATCH.LINK	Link of command interpreter parts
SYSBLD.SDSMAC.BATCH.LINK	Link of macro assembler
SYSBLD.LINKER.BATCH.LINK	Link of link editor

The approximate run time is 2.9 hours.

B.7 BUILDING THE DX10 SYSTEM DISK

Install a new disk in drive DS03 (must be DS03 for MVI commands). Initialize the volume using the INV command and the following responses:

```
[] INV

INITIALIZE NEW VOLUME
                    UNIT NAME:  DS03
                  VOLUME NAME:  REL33
  NUMBER OF VCATALOG ENTRIES:   100 (DS32), 200 (DS10),
                                342 (DS25, DS50)
       BAD TRACK ACCESS NAME:   DUMY
```

NOTE

If drive DS03 is not available for this purpose, it may be changed but with difficulty. One must text edit a file on the master source disk to change the operand for the disk drive for the MVI command. The file pathname is SYSBLD.MVICONT. The first line of the file contains the entry DS03 that must be changed to the name of the available drive. This change and changing the operand for the drive response of the INV command above are the only changes required. Proceed with caution since the master disk must be written to when QUITING the edit session.

Assign the synonyms as described for the links and execute the following batch streams:

Batch Stream	Function
SYSBLD.BATCH.BLDX10	Build the new DX10 system disk
SYSBLD.SCI990.BATCH.SCI990	Install the command interpreter
SYSBLD.BATCH.PROC0	Install SCI procedures (level 0)
SYSBLD.BATCH.PROC2	Install SCI procedures (level 2)
SYSBLD.BATCH.PROC4	Install SCI procedures (level 4)
SYSBLD.BATCH.PROC6	Install SCI procedures (level 6)
SYSBLD.BATCH.MENU	Install the SCI procedure menus
SYSBLD.BATCH.SDSMAC	Install the macro assembler
SYSBLD.LINKER.BATCH.INSTALL	Install the link editor
SYSBLD.BATCH.PROTCT	Delete protect the system tasks

The approximate time is 1.1 hours.

Patch the system just built by copying and editing the file named REL33.PATCH.MEMRES. The instructions for editing the file are contained in the first few lines of the file itself. Use the Copy/Concatenate utility to copy the file:

```
[]CC
COPY/CONCATENATE

    INPUT ACCESS NAME(S): REL33,PATCH.MEMRES
      OUTPUT ACCESS NAME: REL33.PATCH.XMEMRES
                REPLACE?: NO
  MAXIMUM RECORD LENGTH: 80
```

Edit the file named REL33.PATCH.XMEMRES to assign the required synonyms. The link map for the built system is in the file named VOLOBJ.DXLINK.MAP.S$IMAGES. Assign the synonym $$DSC$ to the value REL33 and execute the batch stream named REL33.PATCH.XMEMRES. The runtime is approximately 5 minutes.

Execute the batch stream named REL33.PATCH.PROGA to patch the system program file. This takes about 5 minutes.

B.8 BUILDING THE DX10 DISK BUILD MAGNETIC TAPES

To make the magnetic tapes for the magnetic tape disk build procedure, follow the instructions listed below. The synonyms assigned are to be used during the entire process. Three magnetic tapes are made requiring the following resources:

System Disk	-- Contains system and temporary space.
Source Disk	-- Contains source, object, and batch streams (SYSBLD).
Built DX10 System	-- Contains a DX10 system as the result of the preceding procedure for building the DX10 system disk.

Assign the following synonyms:

Synonym	Value
DSC	SYSBLD
DSC2	REL33 (new DX10 system disk)

*For Tape 1: .Mount a 1200-foot magnetic tape in drive one (MT01) with a write-enable ring installed. Then, execute the batch stream named SYSBLD.BATCH.BST1. The approximate runtime is 25 minutes.

*For Tape 2: .Mount a second 1200-foot magnetic tape in drive one (MT01) with a write-enable ring installed and execute the batch stream named SYSBLD.BATCH.BST2. The approximate runtime is one minute.

*For Tape 3: .Mount a third 1200-foot magnetic tape in drive one (MT01) with a write-enable ring installed and execute the batch stream named SYSBLD.BATCH.BST3. The approximate runtime is 10 minutes.

APPENDIX C

SCHEDULER STRUCTURE AND OPERATION

C.1 FLOW OF CONTROL FOR DX10 SCHEDULER

The following list is a detailed flow of control for the DX10 task scheduler. The scheduler is entered when one of the following five conditions is met:

* When the executing task suspends.
* When a time delay task is due to become active.
* When a task time slices out (if time slicing is enabled).
* When a device service routine (DSR) bids a task.
* When the task sentry decides to lower the priority of the executing task.

The scheduler is entered from the routine named TRAPRT, which is the common exit point for XOPRT2, XOPRT3, TM$DEC, and TM$CLR. The scheduler will not be entered if the scheduler inhibit flag (TM$EXT) is high, if the time slice extended flag (TMESLC) is non-zero, or if the previously executing task was in map file 0.

The variable ETSK is a word in the root of DX10 that points to the TSB of the currently executing task. When no task is executing, ETSK is null.

The flow of control for the task scheduler is given in the steps that follow.

0.0 MAIN ENTRY POINT (defined as SLCSUS)

1.0 ETSK NULL?

If ETSK points to a TSB when the scheduler is entered, then either this task was time sliced out, a device service routine (DSR) bid a task, a time delay task was due to become active, or the task sentry decided to lower this task's priority.

YES - GO TO 3.0

2.0 Put the TSB pointed to by ETSK on the active queue, task sentry for CPU-bound tasks, and set ETSK to zero.

3.0 CLEAR:
These flags are cleared each scheduling cycle:
Time slice ended flag (TM$DFR);
Scheduler inhibit flag (TM$EXT);
Time slice extended flag (TMESLC).

4.0 UPDATE TIME AND DATE
The number of elapsed system time units is added to the computer clock calendar.

5.0 UPDATE TIME DELAY TASKS
Time delay tasks also are updated by the number of elapsed system units. If the task is due to become active, the scheduler puts it in an active status.

6.0 HAS A SYSTEM TIME UNIT ELAPSED?
The timeout logic for device service routines should be entered at 50 millisecond intervals; i.e., each system time unit.

YES - GO TO 8.0

7.0 HAS A DSR BID UP A TASK?
If a DSR bids a task, then the global variable BIDTSK is non-zero. If this is the case, the DSR timeout must be entered now.

NO -- GO TO 9.0

8.0 CHECK REENTER-ME AND TIMEOUTS FOR PDTS (BLWP TMOUT)

9.0 IS TILINE END OF RECORD OUTSTANDING?
Since tasks that do TILINE I/O are locked into memory, do a TILINE end of record as soon as posible to allow the task to be rolled. If TILINE EOR is outstanding, the global variable SCB is non-zero.

YES -- GO TO 24.5

10.0 IS ACTIVE QUEUE EMPTY?

YES -- GO TO 13.0

11.0 GET HIGHEST PRIORITY TASK OFF ACTIVE QUEUE;
SET ETSK TO PONT TO TSB OF HIGHEST PRIORITY TASK.

12.0 IS ETSK A REAL TIME OR SYSTEM PRIORITY?
If a real time or system task wants to execute, the scheduler does not attempt to service SCI bids during this scheduling cycle.

YES -- GO TO 19.0

13.0 IS RESTART IN PROGRESS?
No SCI bids are serviced until the system restart task has initialized the system.

YES -- GO TO 15.0

14.0 SCAN KSBs FOR SCI BIDS
The scheduler bids SCI for each KSB that requests it using the routine named TMBIDO.

15.0 HAS A TASK BEEN SELECTED (Is ETGK not null)?

YES -- GO TO 19.0

16.0 IS TASK LOAD IN PROGRESS (TMMLIP not null)?
If the scheduler has previously requested a task to be rolled in, there is no need to try to roll in another task.

YES -- GO TO 18.0

17.0 IS ANY TASK WAITING ON MEMORY (TMWOMO not null)?

YES -- GO TO 21.0

18.0 GO IDLE (wait for interrupt).

GO TO 3.0

19.0 IS THERE AN ALTERNATE TSB FOR ETSK?

YES -- GO TO 22.0

20.0 SHOULD A TASK WAITING ON MEMORY BE LOADED?
If a task waiting on memory is of higher priority or equal priority and ETSK has had a minimum number of time

slices, if time slicing is available, then the task loader is given the CPU to load the waiting task.

NO -- GO TO 22.0

21.0 REQUEUE ETSK ON ACTIVE QUEUE IF NOT NULL;
SET ETSK TO POINT TO TASK LOADER TSB;
SET TMMLIP TO NONZERO (SIGNAL LOAD IN PROGRESS).

22.0 SET TIME SLICE COUNT
Initialize the number of clock ticks for a time slice. If time slicing is disabled, a non-zero number is used here.

23.0 IS END OF RECORD OUTSTANDING FOR ETSK?
Since TILINE EOR was checked earlier, this is a check for CRU I/O.

NO -- GO TO 25.0

24.0 REQUEUE ETSK ON THE ACTIVE QUEUE

24.5 SET ETSK TO POINT TO DEVICE DRIVER TASK;

GO TO 34.0

25.0 IS THERE ANY ALTERNATE TSB PRESENT FOR ETSK?

NO -- GO TO 27.0

26.0 SET ETSK TO POINT TO ALTERNATE TSB;

GO TO 34.0

27.0 IS THIS TASK QUIETING?
Tasks that are quieting have been selected to be rolled and are kept on the active queue until their I/O is finished.

NO -- GO TO 30.0

28.0 IS I/O IN PROGRESS?

YES -- GO TO 2.0

29.0 PUT ETSK ON QUIET QUEUE;
The task loader is a dedicated queue server of the quiet queue and will be activated when something is placed on its queue.
CLEAR ETSK;

GO TO 3.0

30.0 IS LEAVE ALONG FLAG SET FOR ETSK?

YES -- GO TO 34.0

31.0 IS ABORT FLAG SET?

NO -- GO TO 34.0

32.0 IS I/O COMPLETE?

NO -- GO TO 2.0

33.0 BRANCH TO END ACTION

34.0 PROCESS GET CHARACTER SVC
The processing necessary for the get character SVC is small and the SVC overhead is saved by moving the character from the TSB, placed there by the DSR, to task register 0.

35.0 DOES AN OVERLAY NEED TO BE READ IN FOR ETSK?

NO -- GO TO 37.0

36.0 PLACE ETSK ON OVERLAY LOADER QUEUE;

GO TO 3.0

37.0 IS ETSK UNDER CONTROL?
This is a flag set in the TSB.

NO -- GO TO 39.0

38.0 PLACE ETSK IN STATE 6;
Task suspended by the scheduler.
CLEAR ETSK;

GO TO 2.0

39.0 SET UP TASK FOR EXECUTION
Set task in state 7;
Increment loader roll count (TSBT1);
Put task memory on time ordered list (if not memory resi dent)

40.0 GIVE CONTROL TO ETSK VIA A RTWP USING PC, WS, AND STATUS IN THE TSB POINTED TO BY ETSK.

C.2 PREEMPTIVE EXECUTION

Task scheduler operation is based on the concept of preemptive execution. Preemptive execution allows a higher priority task to have the CPU whenever it becomes active. For example, if task T1 with a priority of R5 is executing, and a task T2 of priority R2 is bid, T1 is suspended and T2 is placed in execution. If task T3 of priority R0 is subsequently bid, T2 is suspended and T3 is put into execution. Preemption is shown in figure C-1.

Time	Task in Execution	Active Task Queue	Action
0 ms.	T1	T1/R5	
30 ms. T2 prempts T1	T2	T2/R2,T1,R5	T2 is bid
50 ms.	T2	T2/R2,T1,R5	
	.	.	
	.	.	
750 ms.	T2	T2/R2,T1/R5	
783 ms. T3 prempts T2	T3	T3/R1,T2/R2,T1/R5	T3 is bid
800 ms.	T3	T3/R1,T2/R2,T1,R5	
841 ms.	T2	T2/R2,T1/R5	T3 has completed
850 ms.	T2	T2/R2,T1/R5	
.	.	.	
.	.	.	

T1 - has an assigned priority of R5
T2 - has an assigned priority of R2
T3 - has an assigned priority of R1

Figure C-1 Preemption

C.3 TIME SLICING

Time slicing is a task scheduler option that can be selected at system generation (sysgen) time. Also specified at sysgen is the length of the time slice. This is defined as a multiple of system time units (at 50 milliseconds each). The default parameters are to have time slicing, with each time slice is to be one system time unit (50

milliseconds). When time slicing is invoked, the time slice value is the amount of CPU time that is given a task each time it executes, and the next task to execute is the next task of the same priority. (This is somewhat different from the pre-DX10 3.2/TX2.3/TX5 task scheduler as this scheduler only slices tasks of
is the only task running. If task T5, priority R2, is bid, followed shortly by task T6, priority R2, then T5, and T6 will alternate running for 50 millisecond time slices. When both have completed, task T4 is allowed to execute again. This is shown in figure C-2.

Time		Task in Execution	Active Task Queue	Action
0 ms.		T4	T4/R5	
50 ms.		T4	T4/R5	
100 ms.		T4	T4/R5	
128 ms.	T5 prempts T4	T5	T5/R2,T4/R5	Task T5 is bid
150 ms.		T5	T5/R2,T4/R5	
183 ms.		T5	T5/R2,T6/R2,T4/R5	Task T6 is bid
200 ms.		T6	T6/R2,T5/R2,T4/R5	
250 ms.		T6	T5/R2,T6/R2,T4/R5	
300 ms.		T6	T6/R2,T5/R2,T4/R5	
350 ms.		T5	T5/R2,T6/R2,T4/R5	
400 ms.		T6	T6/R2,T4/R5	T5 suspends for I/O
450 ms.		T6	T6/R2,T4/R5	
500 ms.		T6	T6/R2,T4/R5	
540 ms.		T6	T6/R2,T5/R2,T4/R5	T5 completes I/O
550 ms.		T5	T5/R2,T6/R2,T4/R5	
600 ms.		T6	T6/R2,T5/R2,T4/R5	
650 ms.		T5	T5/R2,T6/R2,T4/R5	
700 ms.		T6	T6/R2,T5/R2,T4/R5	
717 ms.		T5	T5/R2,T4/R5	T6 terminates
750 ms.		T5	T5/R2,T4/R5	
800 ms.		T5	T5/R2,T4/R5	
850 ms.		T5	T5/R2,T4/R5	
852 ms.		T5	T4/R5	T5 terminates
900 ms.		T4	T4/R5	
950 ms.		T4	T4/R5	

T4 - has an assigned priority of R5.
T5 - has an assigned priority of R2.
T6 - has an assigned priority of R2.

Figure C-2 Time Slicing

C.4 TASK SENTRY

Task sentry is the other scheduler option that can be selected at sysgen time. Also defined at sysgen is the task sentry value (expressed in multiples of system time units), if the task sentry is to be used. The sysgen default parameters are not to have task sentry selected, but if task sentry is selected, the default time value is 60 system time units (three seconds). When task sentry is operational the operating system determines which task is running at the end of each system time unit. If the same task is running now as was running at the last check, an elapsed time counter is decremented. If a different task is running, then the elapsed time counter is reset to the task sentry value specified at sysgen. If the elapsed time counter reaches zero, the task in execution is lowered to the next lower priority value and placed at the bottom of the list for that priority level. The elapsed time counter is then reset, and the top task on the active queue is placed in execution. If task sentry places a task on a populated list, then the task is returned to its loaded priority level when it is next allowed to execute (i.e., task sentry cannot reduce a task priority below that of the next lower priority task in the system). Likewise, if a task terminates or suspends for any reason, its priority is reset to that the priority assigned to it when it was loaded. Figure C-3 describes this action.

Time	Task in Execution	Active Task Queue	Action
0 ms.	T7	T7/R10	
27 ms.	T8	T8/R9,T7/R10	Task T8 is bid
50 ms.	T8	T8/R9,T7/R10	
81 ms.	T9	T9/R8,T8/R9,T7,R10	Task T9 is bid
100 ms.	T9	T9/R8,T8/R9,T7,R10	
150 ms.	T9	T9/R8,T8/R9,T7,R10	
.	.	.	
.	.	.	
.	.	.	
3100 ms.	T8	T8/R9,T9/R9,T7/R10	Task T9 exceeds sentry
3150 ms.	T8	T8/R9,T9/R9,T7/R10	Task T9 resumes execution
3200 ms.	T8	T8/R9,T9/R9,T7/R10	
.	.	.	
.	.	.	
.	.	.	
6500 ms.	T9	T9/R8,T7/R10	
6550 ms.	T9	T9/R9,T7/R10	Task T9 exceeds sentry
.	.	.	
.	.	.	
.	.	.	
9550 ms.	T7	T7/R10,T9/R10	Task T9 exceeds sentry
9600 ms.	T7	T7/R10,T9/R10	
9621 ms.	T9	T9/R8	T7 suspends for I/O
9650 ms.	T9	T9/R8	
9700 ms.	T9	T9/R8	
9750 ms.	T9	T9/R8	
9772 ms.	T9	T9/R8,T7/R10	T7 completes I/O
9781 ms.	T7	T7/R10	T9 suspends for for I/O
9800 ms.	T7	T7/R10	
9850 ms.	T7	T7/R10	
9894 ms.	T9	T9/R8,T7/R10	T9 completes I/O
9900 ms.	T9	T9/R8,T7/R10	
9981 ms.	T7	T7/R10	T9 teminates
.	.	.	
.	.	.	
.	.	.	

T7 - has an assigned priority of R10
T8 - has an assigned priority of R9
T9 - has an assigned priority of R8

Figure C-3 Task Sentry Operation

C.5 SUMMARY OF SCHEDULER OPERATION

The four modes of scheduler operation are summarized in figure C-4. The modes in which either time slicing or task sentry are described in the previous paragraphs. The scheduler mode in which both time slicing and task sentry are used, acts like the time-slicing-only mode for lists with more than one task, and like task sentry only for lists of just one task, providing that the task sentry value is greater than the time slice value. This is because the task sentry timer is reset each time a new task is allowed a time slice. In the simplest scheduler mode neither time slicing nor task sentry is used. In any case, the highest priority task in the system is always in execution.

Most industrial applications use time slicing only or neither time slicing nor task sentry. If timesharing, scientific programming, or other general purpose data processing functions are performed on the same machine with a real-time control function, the time-slicing-only mode should be used. In this case, the general purpose data processing functions can be carried out on priority levels 1-3, and the control programs can occupy priority levels R1-R127. If each control task is assigned a different priority level, no time slicing of the real-time control tasks occurs. The general purpose DP functions are allowed to use any CPU time not required for the real time control function. If only real-time control is carried out by the CPU, neither time slicing mode nor task sentry mode should be used. In this case, all tasks are assigned a separate priority level. Execution is nearly the same as described above for R1-R127 tasks but there is slightly less system overhead since the operating system no longer performs the time slicing function. In critical response environments, this mode may provide the extra fraction of a second response required. Because of the response characterisitics of the real-time priorities, communications software is frequently given a real time priority.

	Task Sentry: YES	Task Sentry: NO
YES	At end of time slice, task is put on the bottom of the list for its priority level. If a task executes more consecutive system time units than specified for the task sentry, then the task is placed at the bottom of the next lowest priority list in the active queue.	At end of time slice task is put at the bottom of the list for its priority level.
NO	If a task executes more consecutive system time units than specified for the task sentry, the task is placed at the bottom of the next lowest priority list in the active queue.	Highest priority task has CPU as long as it wants it.

Figure C-4 Summary of Scheduler Operation

NOTE

The highest priority task is ALWAYS being executed.

C.6 SUPERVISOR CALLS AFFECTING SCHEDULER OPERATION

Another feature that is useful in the industrial application environment is the Do Not Suspend supervisor call. Execution of the SVC causes the task scheduler to be inhibited from suspending the task making the call. This is the only way to override the scheduler functions. Suspension is inhibitied either 200 milliseconds or a specified number of system time units (50 milliseconds. to 12.750 seconds). The task can suspend itself by executing an I/O, Time Delay, Wait for I/O, or Unconditional Wait supervisor call. This should be used in place of the LIMI 0 instruction.

The following supervisor calls also affect scheduler operation:

* Execute Task -- Causes the initiation of a task that has been installed on any program file.
* Activate Suspended Task -- Reactivates a task that has placed itself in a suspended state by using the Unconditional Wait supervisor call.
* Scheduled Bid Task -- Activates a task at a specified time.
* Time Delay -- Suspends the calling task for the specified number of full system time units.
* Change Priority -- Changes the priority of the calling task.
* Unconditional Wait -- Suspends the calling task indefinitely.
* Activate Time Delay Task -- Activates a specified task that is in time delay.

APPENDIX D

DEVICE STATES AND LUNO ASSIGNMENT

Devices supported by DX10 have three possible states: online, offline, and diagnostic. When a device is offline, no LUNO can be assigned to it. When a device is in the diagnostic state, FUTIL sub-opcode >94 must be used when assigning a LUNO to that device.

APPENDIX E

VDT CHARACTER INPUT SVCs

E.1 INTRODUCTION

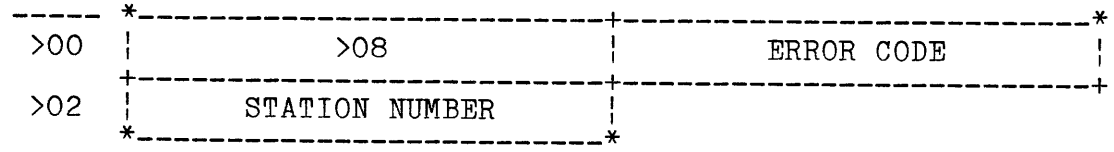
Two supervisor calls (SVCs) are available to input a character from a specified VDT keyboard. They are SVC >08 and SVC >18.

E.1.1 VDT CHARACTER INPUT SUPERVISOR CALL (CODE >08).

This supervisor call inputs a character from a specified station keyboard. The calling task is suspended until the character is transferred. The system places the character in the most significant byte of the task workspace register 0.

The supervisor call block consists of three bytes, and need not be aligned on a word boundary. Byte 0 contains the code, and the system returns a value in byte 1. Byte 2 contains the station number. When the system is unable to locate the station, it returns -1 in byte 1. When the station has not been opened in the character mode, or when power is off at the station, the system returns >80 in byte 1. Otherwise, the system returns zero in that byte.

The VDT character input call block is formatted as follows:



The following is an example of coding for a supervisor call block for a character input from station keyboard supervisor call. The SVC says to input a character from station 2 and place the character in the most significant byte of workspace register 0.

```
SCBC   BYTE   8,0,2
```

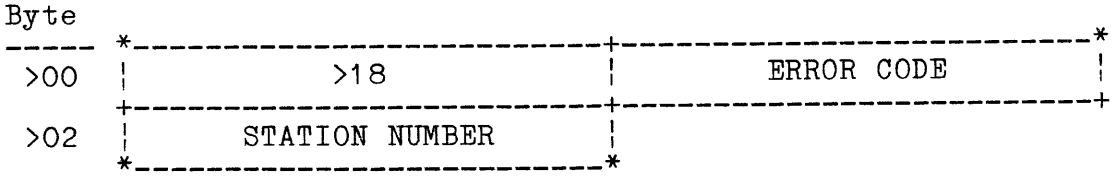
E.1.2 VDT CONDITIONAL CHARACTER INPUT SUPERVISOR CALL (CODE >18).

This supervisor call inputs a character from a specified station keyboard. When a character is entered, the function sets the equal bit

(bit 2) of the status register to 1 and places the character in the most significant byte of workspace register 0. When a character has not been entered, the function sets the equal bit of the status register to 0, indicating a "not equal" status. In either case, the function returns control to the calling task immediately.

The supervisor call block consists of three bytes, and need not be aligned on a word boundary. Byte 0 contains the code, and the system returns a value in byte 1. Byte 2 contains the station number. When the system is unable to locate the station, it returns -1 in byte 1. When the station has not been opened in the character mode, or when power is off at the station, the system returns >80 in byte 1. Otherwise, the system returns zero in that byte.

The VDT conditional character input call block is formatted as follows:



The following is an example of coding for a supervisor call block for a conditional character input from station keyboard supervisor call. The SVC says to input a character from station 5 and place it in the most significant byte of workspace register 0 if a character has been entered at the keyboard.

```
SCBT   BYTE   >18,0,5
```

Appendix F

THE SYSTEM LEVEL DEBUGGER

F.1 GENERAL INFORMATION

The System Level Debugger is used for breakpointing and listing the contents of memory locations and registers when debugging the DX10 system root only. The primary purpose for the System Level Debugger is to debug DSRs. The debugger allows the user to control a program's execution and examine intermediate results in order to determine exactly where problems are occuring.

F.2 HOW TO LINK THE DEBUGGER WITH THE DX10 SYSTEM IMAGE

The debugger is an option included at the time the system is generated. When prompted for DEVICE during system generation the user responds as follows:

```
DEVICE: DEBUG
```

Then, continue with the system generation process.

To delete the debugger from a system enter the following after the DEVICE or NEXT prompt:

```
DEVICE: C DEBUG
```

F.3 PREPARING THE DX10 SYSTEM DEBUGGER

Prior to using the System Level Debugger the debugger image must be modified to specify the target terminal (the default terminal specification is a 913 VDT to which global LUNO 1 is assigned).

F.3.1 SPECIFYING INTERACTIVE TERMINAL TYPE

The value in location >0A6C within the System Level Debugger determines the type of interactive terminal to be used. Consult the system linkmap for the location of the System Level Debugger within the system image. The System Level Debugger is module DEBUG in the system task. The following values indicate the terminal type:

0 - ASR/KSR
2 - 913 VDT
4 - 911 VDT

NOTE

The System Level Debugger assumes the CRU address for a 911 VDT is >100. The CRU address used for 913 VDTs is >0C0. The CRU address for ASR/KSR devices is >000. These CRU addresses may be modified to allow communication with other terminals. To do so, the user must change the locations immediately following the terminal type address (>0A6C).

F.4 ACTIVATING THE SYSTEM LEVEL DEBUGGER

The System Level Debugger is activated through XOP 1,1 breakpoints. The breakpoint may be hard coded in a module or the System Level Debugger may be activated by entering the breakpoint through the front panel. The following procedure will activate the debugger in this manner, if the system is in an idle state.

1. Press HALT.
2. Press PC under DISPLAY.
3. Write down the address displayed.
4. Press MA under ENTER.
5. Press MDD.
6. Write down the value displayed.
7. Press CLR.
8. Enter >2C41 in the display lights.
9. Press MDE.
10. Press RUN.

The debugger activates. Use the debugger to set the value from Step 6 back into the location from Step 3.

F.5 LIST OF THE DEBUGGER COMMANDS

When the debugger is activated, it requests commands by displaying the question mark (?) on the screen. The user may then enter one of the debugger commands listed in Table F-1, and press the RETURN key. The debugger commands are described in the paragraphs following Table F-1. Each debugging command is specified by entering a single character. All command parameters are hexadecimal numbers displayed without the hex (>) sign. A number is automatically terminated after four digits are entered. A number can also be terminated by a period.

Table F-1 List of Debugger Commands

Command	Description
C	Display condition code in status register
G	Execute program being debugged
I	Inspect a range of memory locations
J	Display all workspace registers
L	List all instruction breakpoints
M	Display/alter contents of a memory address
N	Display/alter contents of next memory address
P	Display/alter program counter
Q	Quit debugging session
R	Display/alter contents of a work space register
S	Set instruction breakpoint
W	Display/alter workspace pointer
U	Clear breakpoint
X	Execute single instruction
Z	Resume execution after breakpoint
+	Hexadecimal sum
-	Hexadecimal difference
?	Display menu of commands

F.5.1 C Command -- Display Condition Code in Status Register

The C command displays the contents of the status register (condition code) and allows the user to modify the contents.

Syntax:

```
? C

C= xxxx yyyy
```

Explanation:

xxxx is the current contents of the status register and yyyy is the value entered as the new status register contents (optional entry).

Example:

```
? C

C=C00F C10F
```

The example shown lists the status register contents, >C00F. The contents are changed to >C10F. This sets bit 7 of the status register, placing the computer in the privileged mode. If no value is entered, the register contents remain unchanged.

F.5.2 G Command -- Execute Program Being Debugged

The G command executes the program being debugged. It is used to start the program initially, and can be used to proceed from a breakpoint. It is identical in function to the Z command when used to restart the program.

Syntax:

```
? G
```

Explanation:

This command requires no parameters.

Example:

```
? G
```

The example initiates execution of the user program. The G command is used to start the program or in some cases to restart the program after stopping to examine register contents.

F.5.3 I Command -- Inspect a Range of Memory Locations
This command allows the user to inspect a range of memory locations in the local task address space.

Syntax:

```
? I llll uuuu

xxxx xxxx ... xxxx    yy yy ... yy
```

Explanation:

llll is the lower address of the memory locations to be displayed (this is a required entry). uuuu = is the upper address of the memory locations to be displayed (optional entry). If uuuu is not entered, the debugger displays locations llll through llll + >F. If neither the lower address nor the upper address is entered, a message that an illegal hex value has been entered is displayed. Each xxxx is the displayed hexadecimal contents of a memory location. Each yy is the ASCII representation of the memory contents. The command displays >10 bytes per line, and fills the line of the display on which uuuu is displayed (that is a multiple of >10 bytes is always displayed).

Example:

```
? I C000 C020

2202 0006 0800 ... 5336   ". .. .. ...  S6
0045 3132 3658 ... 0000   .E 12 6X ...  ..
3148 4444 2634 ... 5541   1H DD &4 ...  UA
```

The example displays the memory locations from >C000 to >C020. Memory location >C000 contains >2202, location >C002 contains >0006, etc. Location >C020 contains >3148. The memory locations up to and including location >C02F are displayed, filling out the third line of the example.

F.5.4 J Command -- Show All Local Workspace Registers

This command displays all local workspace registers on one line of the display.

Syntax:

```
? J

xxxx xxxx xxxx ... xxxx
```

Explanation:

The J command requires no parameters. Each xxxx is the value in one of the workspace registers, 0 through 15.

Example:

```
? J

0002 4AC7 0000 0000 ... 0000
```

The example lists the contents of all the workspace registers. Workspace register 0 contains >0002, workspace register 1 contains >4AC7, etc. The listing displays all 16 workspace registers.

F.5.5 L Command -- List All Breakpoints

This command lists the absolute addresses of all breakpoints currently set in the program.

Syntax:

```
? L

LOCATION
xxxx
xxxx
```

Explanation:

This command requires no parameters. Each xxxx is the displayed location of a breakpoint.

Example:

```
? L

LOCATION
B610
BC24
```

The example shows the two breakpoints that have been set in the program, at locations >B610 and >BC24.

F.5.6 M Command -- Display/Alter Contents of Memory Address
The M command displays and allows the user to alter the contents of a specified memory location. Only addresses currently mapped in the task space may be modified with this command.

Syntax:

```
? M nnnn

M nnnn=xxxx yyyy
```

Explanation:

nnnn is the word address of the memory location the user wishes to display (required entry). When nnnn is an odd number, the memory word at nnnn-1 is displayed. xxxx is the displayed value of the memory location. yyyy is the new value the user wishes to place in the memory location (optional entry).

Example:

```
? M B680

M B680=A002 A003
```

The example displays the contents of memory location >B680. The value in that location is >A002, and the contents are changed to >A003.

F.5.7 N Command -- Display/Alter Contents of Next Memory Address

The N command allows the user to display and alter the contents of the next memory address after the address examined previously. This command is used in connection with the R, M or N commands.

Syntax:

```
? N

nnnn=xxxx yyyy
```

Explanation:

nnnn is the displayed address of the next location. xxxx is the displayed value of the next location. yyyy is the new value the user wishes to place in the memory location (optional entry).

Example:

```
? R 4

R4=0022

? N

B00E=0400 0800
```

This example shows an R command and an N command. The R command displays workspace register 4. The N command displays the contents of the next memory address. In this example the next memory address is workspace register 5, at location >B00E. The value in the workspace register, >0400, is changed to >0800.

F.5.8 P Command -- Display/Alter Program Counter

The P command displays the contents of the program counter. The user may alter the displayed contents if desired.

Syntax:

```
? P

PC=xxxx yyyy
```

Explanation:

xxxx is the displayed program counter value. yyyy is the value to replace the currently displayed value. This is an optional entry.

Example:

```
? P

PC=BA20 BA2C
```

The example displays >BA20 as the program counter contents. The contents are changed to >BA2C.

F.5.9 Q Command -- Quit Debugging Session
The Q command terminates the debugging program and the task that is being debugged.

Syntax:

```
? Q
```

Explanation:

This command requires no parameters from the user.

Example:

```
? Q

END DEBUG
```

The message indicates that the system is in an idle state. If the user wishes to exit from the debugger entirely, he or she should enter a G command.

F.5.10 R Command -- Display/Alter Workspace Register
The R command displays the contents of a workspace register. The user may alter the displayed contents if desired.

Syntax:

```
? R

Rn=xxxx yyyy
```

Explanation:

n is the workspace register number, from 0 to F (this is a required entry). xxxx is the displayed workspace register contents. yyyy is the value (this is an optional entry) that replaces the displayed contents.

Example:

? R4

R4=2A60 2A64

The example displays the contents of workspace register 4. The value in workspace register 4 is >2A60. The value is changed to >2A64.

F.5.11 S Command -- Set Breakpoint

The S command is used to set an instruction breakpoint at any address currently mapped in the task space. When the breakpoint is reached, the contents of the program counter, workspace pointer, status register, and all current workspace registers (0 through 15) are displayed.

Syntax:

? S yyyy

Explanation:

yyyy is the address for the instruction breakpoint (this is a required entry).

Example:

? S C006

The example sets a breakpoint at memory location >C006. When the task is executed, it stops at location >C006 to allow the user to examine registers, memory, etc., and make changes if necessary. The following values are displayed at a breakpoint:

```
PC=C006 WP=473E ST=0003 (PC)=2C40
0000 23C0 1000 3348 4566 7290 0000 ...  2448
```

The first line of the display shows the current contents of the program counter, the workspace pointer, the status register, and the word at the address in the program counter. The second line of the display shows the contents of the 16 workspace registers, 0 through 15.

F.5.12 U Command -- Clear a Breakpoint

The U command clears (removes) one or more instruction breakpoints.

Syntax:

```
? U yyyy
```

Explanation:

yyyy is the location of the breakpoint that is to be removed (this is an optional entry).

If yyyy is not entered, then all breakpoints are removed.

Example:

```
? U C006
```

The example deletes an instruction breakpoint set at location >C006.

F.5.13 W Command -- Display/Alter Workspace Pointer

The W command displays the contents of the workspace pointer. You can alter the displayed contents if desired.

Syntax:

```
? W

WP=xxxx yyyy
```

Explanation:

xxxx is the displayed workspace counter contents. yyyy is the value that serves as a replacement for the displayed value (optional entry).

Example

```
? W

WP=B010 B030
```

The example displays the workspace pointer contents, >B010. The displayed contents are changed to >B030.

F.5.14 X Command -- Execute a Single Instruction
The X command allows the user to step through the program being debugged, one instruction at a time. The contents of all current workspace registers are displayed after executing each instruction. The X command executes the instruction at the address in the program counter.

Syntax:

```
? X

PC=xxxx

rrrr rrrr rrrr ... rrrr
```

Explanation:

This command requires no parameters from the user. xxxx is the displayed contents of the program counter. Each rrrr is the displayed contents of one of the workspace registers, 0 through 15.

Example:

```
? X

PC=B250

0002 A020 0000 ... 0000
```

The example illustrates an X command to execute the instruction at the address in the program counter. The value in the program counter, >B250, and the workspace registers are displayed. Workspace register 0 contains >0002, register 1 contains >A020, etc.

F.5.15 Z Command -- Resume Execution After Breakpoint
When the Z command is entered, execution proceeds from the current breakpoint to the next breakpoint (if another breakpoint is present). The breakpoint is not cleared, and can be used again, unless a U command is issued.

Syntax:

```
? Z
```

Explanation:

This command requires no parameters from the user.

Example:

? Z

This example restarts program execution after it has stopped at a breakpoint. The Z command is used after a breakpoint is reached, and the user has examined and changed memory, register contents, etc., as desired.

F.5.16 + Command -- Add Hexadecimal Numbers
The + command is used to display the hexadecimal sum of two hexadecimal numbers.

Syntax:

? + xxxx yyyy = zzzz

Explanation:

xxxx is a hexadecimal number (required entry). yyyy is a hexadecimal number that is to be added to xxxx (yyyy is a required entry). zzzz is the hexadecimal sum of xxxx + yyyy.

Example:

? + C420 2184 = E5A4

The example adds >C420 to >2184 and displays the sum of >E5A4.

F.5.17 - Command -- Subtract Hexadecimal Numbers
The (-) command displays the hexadecimal difference of two hexadecimal numbers.

Syntax:

? - xxxx yyyy = zzzz

Explanation:

xxxx is a hexadecimal number (required entry). yyyy is the hexadecimal number from which xxxx is to be subtracted (required entry). zzzz is the difference of the two numbers.

Example:

```
? - 0048 21CE = 2186
```

The example subtracts >48 from >21CE and displays the difference, >2186.

F.5.18 ? Command -- Display A Menu of Commands

The ? command is used to display a menu of all available debug commands, showing the format of the commands and a brief description of each.

Syntax:

```
? ?
```

Explanation:

This command requires no parameters from the user.

Example:

```
? ?

P - PROGRAM COUNTER
W - WORKSPACE POINTER
C - STATUS REGISTER
R - MODIFY WORKSPACE REGISTER
M - MODIFY MEMORY WORD
I - INSPECT MEMORY
J - DUMP ALL REGISTERS
S - SET BREAKPOINT
U - REMOVE BREAKPOINTS
L - LIST ALL BREAKPOINTS
X - SINGLE STEP PROGRAM
Z - CONTINUE EXECUTION AFTER BREAKPOINT
N - DISPLAY NEXT WORD
+ - ADD
- - SUBTRACT
Q - QUIT DEBUGGER ?
```

The user may select any of the commands or exit from the debugger by entering Q, waiting for the END DEBUG response, then entering a G command.

CUT ALONG LINE

USER'S RESPONSE SHEET

Manual Title: Model 990 Computer DX10 Operating System Release 3.4
System Design Document (939153-9701)

Manual Date: 1 October 1981 **Date of This Letter:** _____

User's Name: _____ **Telephone:** _____

Company: _____ **Office/Department:** _____

Street Address: _____

City/State/Zip Code: _____

Please list any discrepancy found in this manual by page, paragraph, figure, or table number in the following space. If there are any other suggestions that you wish to make, feel free to include them. Thank you.

Location in Manual	Comment/Suggestion
_____	_____
	_____
	_____
	_____
_____	_____
	_____
	_____
	_____
_____	_____
	_____
	_____
	_____

NO POSTAGE NECESSARY IF MAILED IN U.S.A.
FOLD ON TWO LINES (LOCATED ON REVERSE SIDE), TAPE AND MAIL

FOLD

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 7284 DALLAS, TX

POSTAGE WILL BE PAID BY ADDRESSEE

TEXAS INSTRUMENTS INCORPORATED
DIGITAL SYSTEMS GROUP
ATTN: TECHNICAL PUBLICATIONS
P.O. Box 2909 M/S 2146
Austin, Texas 78769

FOLD

Texas Instruments U.S. District Sales and Service Offices

(A complete listing of U.S. offices is available from the district office nearest your location)

California
831 S. Douglas Street
El Segundo, California 90245
(213) 973-2571

100 California Street
Suite 480
San Francisco, California 94111
(415) 781-9470

776 Palomar Avenue
P.O. Box 9064
Sunnyvale, California 94086
(408) 732-1840*

3186 Airway
Suite J
Costa Mesa, California 92626
(714) 540-7311

Colorado
9725 East Hampden Avenue
Suite 301
Denver, Colorado 80231
(303) 751-1780

Florida
1850 Lee Road
Suite 115
Winter Park, Florida 32789
(305) 644-3535

Georgia
3300 Northeast Expressway
Building 9
Atlanta, Georgia 30341
(404) 458-7791

*Service telephone number

Illinois
515 West Algonquin Road
Arlington Heights, Illinois 60005
(312) 640-2900
(800) 942-0609*

Massachusetts
504 Totten Pond Road
Waltham, Massachusetts 02154
(617) 890-7400

Michigan
24293 Telegraph Road
Southfield, Michigan 48034
(313) 353-0830
(800) 572-8740*

Minnesota
7625 Parklawn Avenue
Minneapolis, Minnesota 55435
(612) 830-1600

Missouri
2368 Schuetz
St. Louis, Missouri 63141
(314) 569-0801*

New Jersey
1245 Westfield Avenue
Clark, New Jersey 07066
(201) 574-9800

Ohio
4124 Linden Avenue
Dayton, Ohio 45432
(513) 258-3877

Pennsylvania
420 Rouser Road
Coraopolis, Pennsylvania 15108
(412) 771-8550

Texas
8001 Stemmons Expressway
P.O. Box 226080
M/S 3108
Dallas, Texas 75266
(214) 689-4460

13510 North Central Expressway
P.O. Box 225214
M/S 393
Dallas, Texas 75265
(214) 238-3881

9000 Southwest Freeway, Suite 400
Houston, Texas 77074
(713) 776-6577

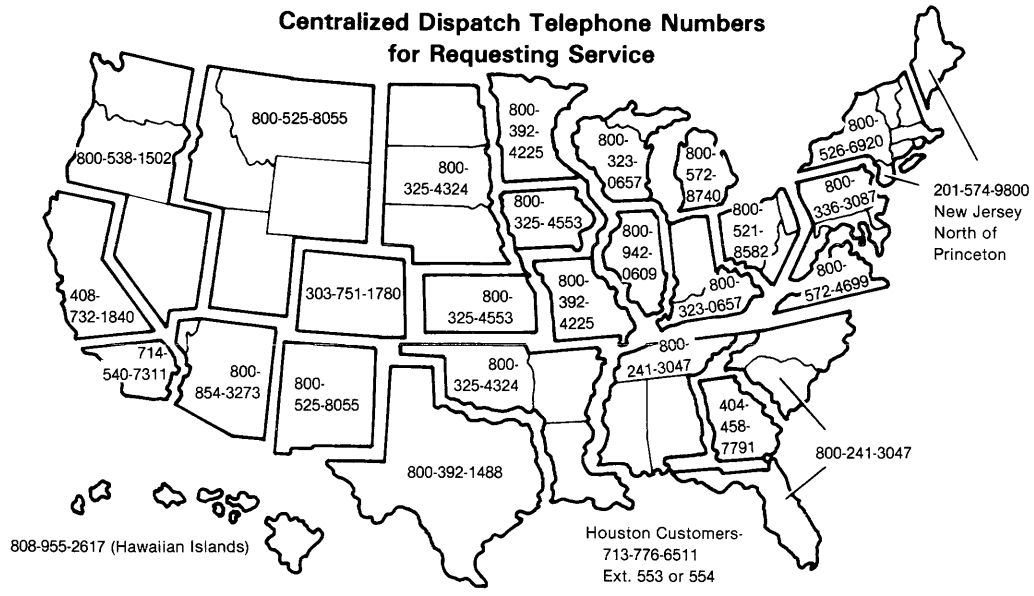
8585 Commerce Drive, Suite 518
Houston, Texas 77036
(713) 776-6531
(713) 776-6553*

Virginia
1745 Jefferson Davis Highway
Crystal Square 4, Suite 600
Arlington, Virginia 22202
(703) 553-2200

Wisconsin
205 Bishops Way
Suite 214
Brookfield, Wisconsin 53005
(414) 784-1323

TI-CARE*

Centralized Dispatch Telephone Numbers for Requesting Service



*Service mark of Texas Instruments

The **TI Customer Support Line** is available to answer our customers' complex technical questions. The extensive experience of a selected group of TI senior engineers and systems analysts is made available directly to our customers. The **TI Customer Support Line** telephone number is **(512) 250-7407.**

TEXAS INSTRUMENTS
INCORPORATED
DIGITAL SYSTEMS GROUP
POST OFFICE BOX 2909 AUSTIN, TEXAS