PERKIN-ELMER

OS/32
APPLICATION LEVEL PROGRAMMER

Reference Manual

48-039 F00 R01

The information in this document is subject to change without notice and should not be construed as a commitment by the Perkin-Elmer Corporation. The Perkin-Elmer Corporation assumes no responsibility for any errors that may appear in this document.

The software described in this document is furnished under a license, and it can be used or copied only in a manner permitted by that license. Any copy of the described software must include the Perkin-Elmer copyright notice. Title to and ownership of the described software and any copies thereof shall remain in The Perkin-Elmer Corporation.

The Perkin-Elmer Corporation assumes no responsibility for the use or reliability of its software on equipment that is not supplied by Perkin-Elmer.

The Perkin-Elmer Corporation, Data Systems Group, 2 Crescent Place, Oceanport, New Jersey 07757

© 1981, 1983 by The Perkin-Elmer Corporation

Printed in the United States of America

TABLE OF CONTENTS

PREFACE v

CHAPTERS

1 PROGRAMMING IN AN OS/32 ENVIRONMENT

1.1 OS/32 OPERATIONAL OVERVIEW 1-1

1.2 OS/32 REAL-TIME ENVIRONMENT 1-3

1.3 OS/32 MULTI-TERMINAL MONITOR (MTM) TIME-SHARING ENVIRONMENT 1-4

1.4 THE OS/32 APPLICATION PROGRAMMER 1-5

2 TASK STRUCTURE AND EXECUTION CONTROL

2.1 INTRODUCTION 2-1

2.2 IMAGE FILE FORMAT 2-2

2.3 LOADING A USER-TASK (U-TASK) INTO MEMORY 2-5

2.4 TASK STATES AND PRIORITIES 2-9

2.5 MONITOR TASKS 2-13
2.5.1 The OS/32 Multi-Terminal Monitor (MTM) 2-13

2.6 RESTRICTIONS ON INTERTASK COMMUNICATION 2-14

2.7 ACCESSING OS/32 SYSTEM SERVICES 2-14

3 INTERRUPT SERVICING IN A REAL-TIME ENVIRONMENT

3.1 INTRODUCTION 3-1

3.2 TASK STATUS WORD (TSW) 3-2

3.3 TRAPS HANDLED BY OS/32 3-5

CHAPTERS (Continued)

3.4	TRAPS HANDLED BY USER-WRITTEN TASKS	3-8
3.4.1	Task Queue Trap-Causing Events	3-8
3.4.2	User-Defined Trap-Causing Events	3-12
3.5	THE TASK STATUS WORD (TSW) SWAP	3-12
3.6	WRITING TASKS THAT HANDLE TASK TRAPS	3-18
3.6.1	Handling Task Queue Traps	3-22
3.6.2	Tips for Writing Task Trap Handling Routines	3-23
3.6.3	Handling Traps from Trap-Generating Devices	3-24
3.6.4	Sample Task Trap Handling Program	3-25
3.6.5	Using the OS/32 System Macro Library to Handle Traps	3-27
3.6.6	Writing FORTRAN Trap Handling Programs	3-27
3.6.7	Writing Pascal Trap Handling Programs	3-28

4 OS/32 DISK FILE MANAGEMENT SERVICES

4.1	INTRODUCTION TO THE OS/32 FILE MANAGER	4-1
4.2	SYSTEM RESOURCE MANAGEMENT	4-2
4.3	FILE ORGANIZATION	4-3
4.3.1	Linked-List Indexed Organization	4-5
4.3.2	Contiguous Organization	4-6
4.4	SUPPORTED DISK FILE TYPES	4-6
4.4.1	Contiguous Files	4-7
4.4.2	Indexed and Nonbuffered Indexed Files	4-7
4.4.3	Extendable Contiguous Files	4-7
4.5	DISK SPACE MANAGEMENT	4-8
4.5.1	File Directories	4-9
4.5.2	Bit Map	4-12
4.5.3	Permanent and Temporary File Allocation	4-13
4.6	ASSIGNING FILES TO A TASK	4-13
4.7	ACCESS METHODS	4-14
4.7.1	Buffered Input/Output (I/O) (Indexed Files)	4-16
4.7.2	Nonbuffered Input/Output (I/O)	4-16
4.7.2.1	Accessing Contiguous Files	4-17
4.7.2.2	Accessing Nonbuffered Indexed Files	4-17
4.7.2.3	Accessing Extendable Contiguous Files	4-17
4.8	FILE SECURITY	4-18

CHAPTERS (Continued)

4.9	CHOOSING THE RIGHT FILE TYPE	4-22
4.9.1	Using Contiguous Files	4-22
4.9.2	Using Indexed Files	4-23
4.9.3	Using Nonbuffered Indexed Files	4-23
4.9.4	Using Extendable Contiguous Files	4-24
4.9.5	Disk Fragmentation	4-24

5 WRITING PROGRAMS THAT ACCESS OS/32 SYSTEM SERVICES

5.1	INTRODUCTION	5-1
5.2	BUILDING A SUPERVISOR CALL (SVC) PARAMETER BLOCK	5-2
5.2.1	Accessing Input/Output (I/O) System Services	5-2
5.2.2	Accessing File Management Services	5-5
5.3	USING THE OS/32 SYSTEM MACRO LIBRARY TO ACCESS SYSTEM SERVICES	5-8
5.4	WRITING A FORTRAN PROGRAM THAT ACCESSES SYSTEM SERVICES	5-9
5.5	WRITING A PASCAL PROGRAM THAT ACCESSES SYSTEM SERVICES	5-10

FIGURES

1-1	Summary of OS/32 Features	1-2
2-1	Conversion of Object Modules into Task Image by Linkage Editor	2-1
2-2	Task Image File Format for a Segmented Task	2-3
2-3	Task Address Space on a MAC Machine	2-6
2-4	Task Address Space on a MAT Machine	2-7
2-5	Segmented Task Loaded into Memory	2-8
2-6	Task States	2-12
3-1	Task Status Word	3-2
3-2	Perkin-Elmer Standard Circular List	3-8
3-3	Circular List with Task Queue Entries for Subtask State Change	3-11
3-4	Task Queue Entry for APU Signal	3-12
3-5	User-Dedicated Location	3-13
3-6	Task Status Word Swap	3-15

FIGURES (Continued)

4-1	Formatted Disk Surface	4-4
4-2	Linked-List Indexed File Organization	4-5
4-3	Volume Descriptor	4-8
4-4	Primary Directory Block	4-9
4-5	Primary Directory Entry	4-10
4-6	Secondary File Directory (SYSTEM.DIR)	4-12
4-7	Task Interfaces to Access Methods	4-14
5-1	Task Interface to OS/32 Executor Routines	5-2
5-2	SVC 1 Parameter Block Defined by $SVC 1	5-3
5-3	SVC 7 Parameter Block Defined by $SVC 7	5-6
5-4	SVC 2 Code 16 Parameter Block	5-6

TABLES

2-1	TASK WAIT STATES	2-10
3-1	TSW BIT SETTINGS	3-3
3-2	ARITHMETIC FAULT TRAP-CAUSING EVENTS	3-6
3-3	MEMORY ACCESS FAULT TRAP-CAUSING EVENTS	3-6
3-4	DATA FORMAT/ALIGNMENT FAULT TRAP-CAUSING EVENTS	3-7
3-5	TASK QUEUE TRAP-CAUSING EVENTS	3-9
3-6	SUBTASK REASON CODES AND CORRESPONDING STATE CHANGES	3-11
3-7	UDL FIELDS USED TO HANDLE TASK TRAPS	3-16
3-8	SUMMARY OF TASK STRUCTURES USED FOR HANDLING TRAPS	3-20
4-1	ACCESS PRIVILEGE COMPATIBILITY	4-19
4-2	ALLOWABLE ACCESS PRIVILEGE CHANGES	4-20
4-3	READ/WRITE KEYS	4-21
4-4	FILE TYPE SUMMARY	4-22
5-1	SVC 1 FUNCTION CODES	5-3

INDEX Ind-1

PREFACE

This manual describes the facilities available to a programmer implementing an application in an OS/32 real-time or OS/32 Multi-Terminal Monitor (MTM) environment. Chapter 1 introduces the fundamental environmental concepts of the system. Chapters 2, 3, 4, and 5 give specific details of the data structures and programming methods used to access OS/32 system services. Included in these chapters are descriptions of task structure, trap handling, file management services and the OS/32 supervisor calls (SVCs). To aid programmers who prefer to work with high level languages, programming examples are given in FORTRAN VII and Pascal, as well as assembly language. Full details on the SVCs used in these examples can be found in the OS/32 Supervisor Call (SVC) Reference Manual.

System programmers wishing to implement privileged software for writing system level control programs are referred to the OS/32 System Level Programmer Reference Manual.

This manual is intended for use with the OS/32 R06.2 release or higher. Additional material specifically related to the Model 3200MPS System has also been included. These Model 3200MPS System features are supported by the OS/32 R07.1 software and higher. Throughout the text, these features are identified as applicable only to the Model 3200MPS System.

For further information on the contents of all Perkin-Elmer 32-bit manuals, see the 32-Bit Systems User Documentation Summary.

CHAPTER 1
PROGRAMMING IN AN OS/32 ENVIRONMENT

1.1 OS/32 OPERATIONAL OVERVIEW

The Perkin-Elmer OS/32 operating system is designed to facilitate programming in a real-time environment. Whether a task is to collect data from a transducer, maintain inventory, or process seismic data, OS/32 provides software service routines that can save programming time and effort. These services include program execution scheduling, memory management, file management, fault handling, device-dependent and device-independent input/output (I/O) services, and intertask communication and control.

How OS/32 actually implements these services depends on the environment within which a program is executed. OS/32 provides three types of operating environments for application programs:

- real-time
- time-sharing
- transaction processing

All three environments can exist simultaneously on the same Perkin-Elmer 32-bit processor.

The OS/32 real-time operating environment is an event-driven, priority-based, multitasking environment within which the other operating environments exist as monitors. Monitors are specialized real-time systems that manipulate the real-time scheduling components of OS/32 to create an environment that employs higher level scheduling algorithms. OS/32 is augmented for time-sharing by the OS/32 Multi-Terminal Monitor (MTM). Transaction processing is provided by RELIANCE. Features of the OS/32 operating environments are summarized in Figure 1-1.

This manual deals exclusively with the real-time and time-sharing environments. For more information on transaction processing, see the RELIANCE Overview Manual.

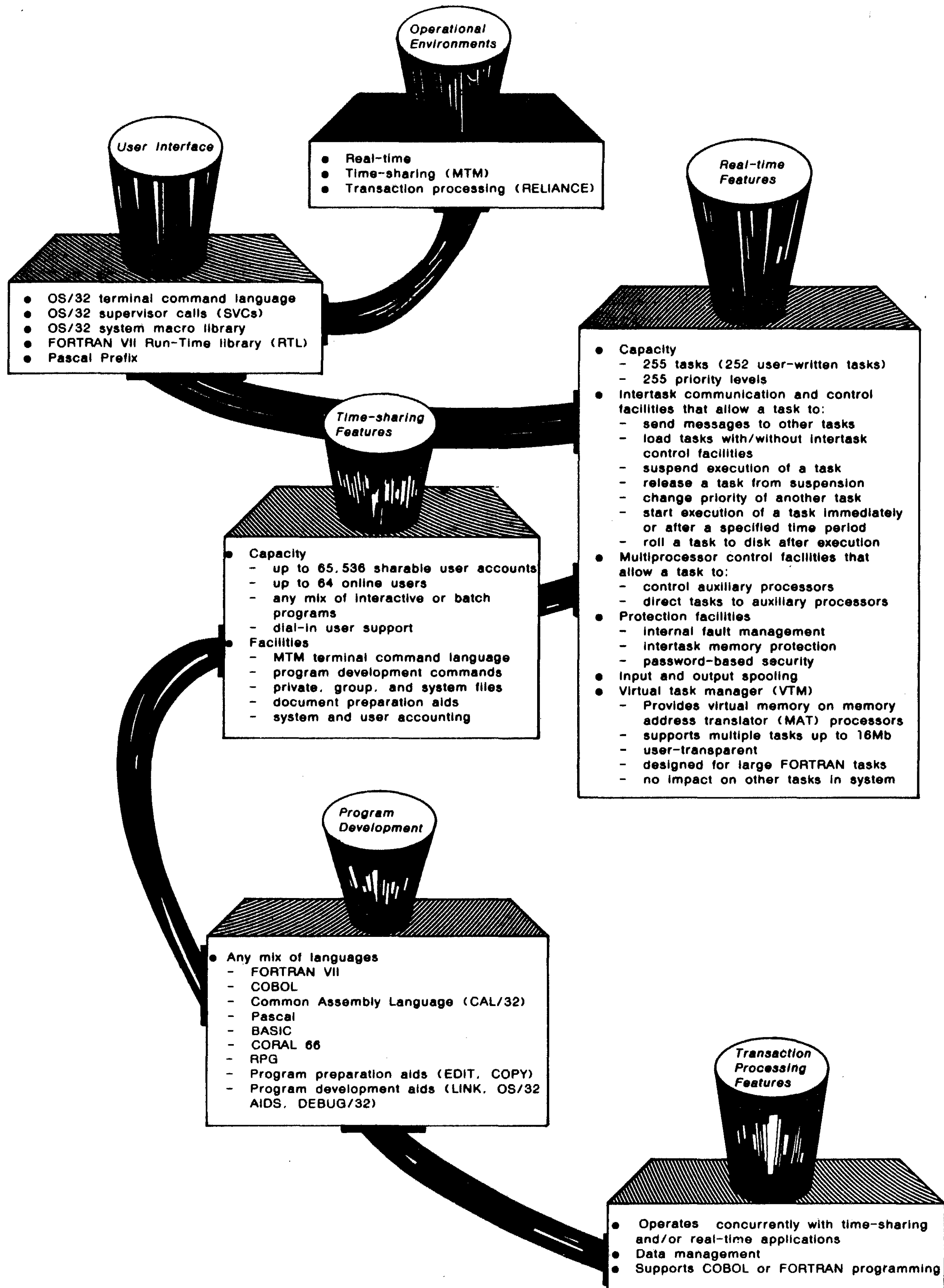


Figure 1-1 Summary of OS/32 Features

1.2 OS/32 REAL-TIME ENVIRONMENT

OS/32 real-time operations enhance the hardware facilities provided by a Perkin-Elmer computer. As a multitasking system, OS/32 can support up to 252 user-written programs executing concurrently. One of these programs can run as a background program while the others are executing in the foreground.

OS/32 uses a priority driven scheduling algorithm with up to 255 priority levels to decide when and how long each program should execute. Priorities are scheduled so that transient events monitored by a real-time program are captured and evaluated quickly, and that all system peripherals are used effectively.

OS/32 also includes a timer facility that can be used to control the start of a procedure or detect whether a procedure has overrun its course. Other control services allow a program to reassign the priority of itself or another executing program.

When internal fault conditions (such as arithmetic overflow, power restore, or incorrect data formats) are detected by the processor, the currently executing program is suspended and control is returned to the operating system. OS/32 handles these traps through default system trap handling routines. Other services are provided, however, that allow application programs to provide customized trap handling routines for handling their own fault conditions.

As the need to access larger programs and data bases increases, the greater the effect the memory addressing capability of a system will have on performance. The 32-bit architecture of OS/32 provides a memory addressing capability of up to 16Mb for each of the 255 programs running on the system. In addition, Link, the OS/32 linkage editor, supports the virtual task manager (VTM). VTM provides a user-transparent virtual memory capability on a task-by-task basis. This capability allows tasks consisting of up to 16Mb of code and data to execute in as little as 128kb of memory. See the OS/32 Link Reference Manual for more information on VTM.

OS/32 provides both device dependent and device independent I/O services. By using device independent services to perform I/O transfers, devices can be reconfigured without reprogramming the application. Device dependent I/O allows control of device-specific functions such as density select on magnetic tape, screen access on a block mode terminal, or connect/disconnect on a dial-up line.

OS/32 also supports user-transparent queuing of I/O requests to files and devices. Each time an I/O transfer is completed, OS/32 activates any outstanding eligible I/O requests before returning control to the executing program.

In a real-time environment, central processing unit (CPU) idling can be critical. The spooling utilities available with OS/32 help eliminate the CPU idling that can occur when writing to slow devices. When a spooler is used, all output is assigned to a pseudo device rather than an actual physical device. A spooler redirects this output to files on disk. Later, the spooler writes these files to the correct physical device on a priority basis. OS/32 provides two spoolers: SPL/32, a flexible, dynamic program designed to meet the high volume of printing required of the commercial user; and OS/32 Spooler, a smaller program for users with fewer printers and less memory. See the SPL/32 System Administration Reference Manual and OS/32 Multi-Terminal Monitor (MTM) Reference Manual for more information on spooling in an OS/32 environment.

OS/32 file management services provide four different file types: indexed, nonbuffered indexed, contiguous, and extendable contiguous. Each file type is designed to meet the requirements of specific real-time situations. For example, nonbuffered indexed and extendable contiguous files are designed for applications that involve random I/O and require variable length files that can be extended without system buffering overhead. Applications which require a fixed-file length and no buffering overhead can use the contiguous file type. For applications that involve sequential I/O (such as compiling a program), the indexed file type is preferred.

In order to operate smoothly, a multitasking system should allow communication among executing programs. OS/32 provides a queue message service that gives each program its own private message queue consisting of a chain or ring of message buffers. Two types of message services are provided. One type passes fixed-length (64-byte) messages. The other type allows variable length messages with no limit on the message length other than the amount of memory available to the task.

1.3 OS/32 MULTI-TERMINAL MONITOR (MTM) TIME-SHARING ENVIRONMENT

OS/32 MTM adds another dimension to the OS/32 real-time facilities. This dimension is time-sharing. Under MTM, up to 64 users can be simultaneously signed on to a Perkin-Elmer system in any combination of interactive or batch modes. To prevent one user from tying up the CPU to the exclusion of others signed on the system, MTM schedules processor time according to the jobs performed by the programs. Compute intensive jobs are given lower priority but longer time slices than I/O intensive jobs, which are given higher priority levels and shorter time slices.

One of the main uses of the time-sharing environment is program development. MTM users can develop programs in Common Assembly Language (CAL/32), FORTRAN VII, Pascal, COBOL, BASIC, CORAL 66, or RPG. The MTM terminal command language allows users to develop their own command files for compiling, assembling, linking, and running a program. See the OS/32 Multi-Terminal Monitor (MTM) Reference Manual for more information on developing programs in an MTM environment.

To support a diverse community of users, Perkin-Elmer provides MTM users with the authorized user utility. This utility allows users to specify certain privileges for each of the 64K (65,536) accounts supported on the system. Once privileges have been specified for an account, all users signed on that account will be allowed to use those privileges. For information on what privileges can be assigned, see the OS/32 Multi-Terminal Monitor (MTM) System Planning and Operator Reference Manual.

Because MTM operates as a monitor within the OS/32 real-time environment, MTM can serve as a low-priority background environment for real-time applications or the primary environment in the system.

1.4 THE OS/32 APPLICATION PROGRAMMER

An application programmer is responsible for writing programs that result in optimum system response and throughput for a particular application. By using the OS/32 software services described in this manual, the user can greatly reduce the programming effort needed to achieve greater performance. The following chapters describe the basic data structures that should be understood to use OS/32 system services effectively.

CHAPTER 2
TASK STRUCTURE AND EXECUTION CONTROL

2.1 INTRODUCTION

When a program is compiled or assembled, it is converted into object modules that are stored in one or more object files on disk. These object modules must be converted into an executable form before the program can be run. This executable form is called a task.

As shown in Figure 2-1, the OS/32 linkage editor, Link, performs this conversion by creating an image of the task from the modules in the object files. Link stores the task image, with instructions for loading the task, into an image file on disk.

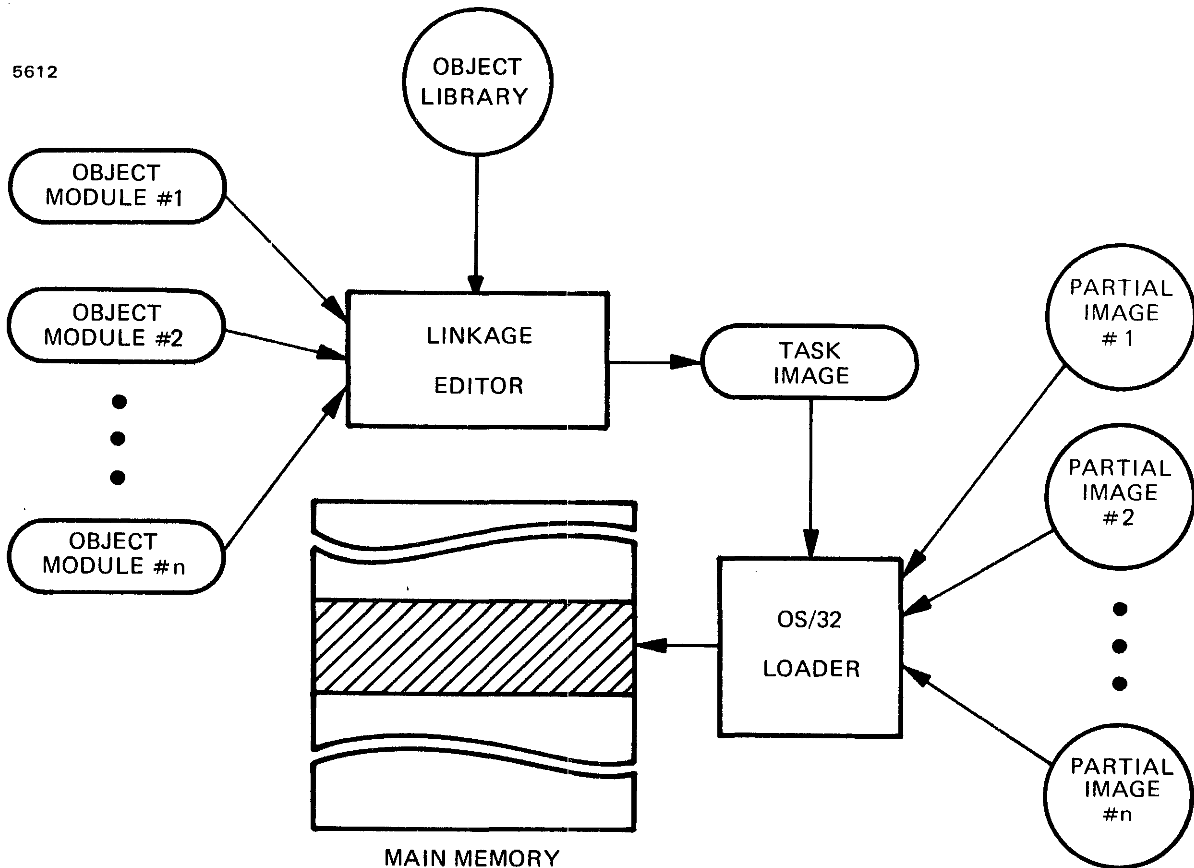


Figure 2-1 Conversion of Object Modules into Task Image By Linkage Editor

An application task can be linked as an executive-task (e-task), diagnostic task (d-task), or a user-task (u-task). E-tasks run with memory relocation/protection hardware turned off and are allowed to execute all instructions provided by the Perkin-Elmer processor hardware. D-tasks, like e-tasks, can execute all instructions; however, they run with the relocation/protection hardware enabled. U-tasks run with the relocation/protection hardware enabled and are restricted to a subset of machine instructions known as nonprivileged instructions. This manual pertains to nonprivileged u-tasks only. For more information on e-tasks, d-tasks, and privileged u-tasks, see the OS/32 Systems Level Programmer Reference Manual.

The following sections describe the format of the image file for a u-task, how a u-task is actually loaded from this file into memory, and what happens to a task after it is loaded. These sections also refer to a number of Link and OS/32 operator commands that are used by the programmer to develop programs. To learn more about the commands discussed in these sections, see the OS/32 Link Reference Manual or OS/32 Operator Reference Manual.

2.2 IMAGE FILE FORMAT

Figure 2-2 shows the format of an image file for a task. The first segment in the task image file is the loader information block (LIB). The LIB tells the OS/32 loader how to load the image into memory. While the task is loaded, the LIB is kept in the loader's private memory area, not in the task address space, until the loader no longer requires it.

Following the LIB is the history records area. The history records are created by OS/32 PATCH. PATCH is a utility that allows the user to update a program by making changes to its image or object file instead of the source. Any changes made to the task or its LIB via PATCH, are recorded in the history records area. See the OS/32 PATCH Reference Manual for more information.

The task image that is actually loaded into memory consists of at least one private image segment. The linkage editor creates the private image with read, write, and execute privileges following the LIB and any history records. The private image contains the impure code from the included object modules. Impure code is code that cannot be shared by other executing tasks. It can consist of the user program code, data that the user designates as impure, and common data areas such as those used by the FORTRAN COMMON statement to store variables. If NSEGMENTED is specified as a task option in the Link OPTION command when the task is built, the pure code is also included in the private image. Pure code can be read or executed by other tasks.

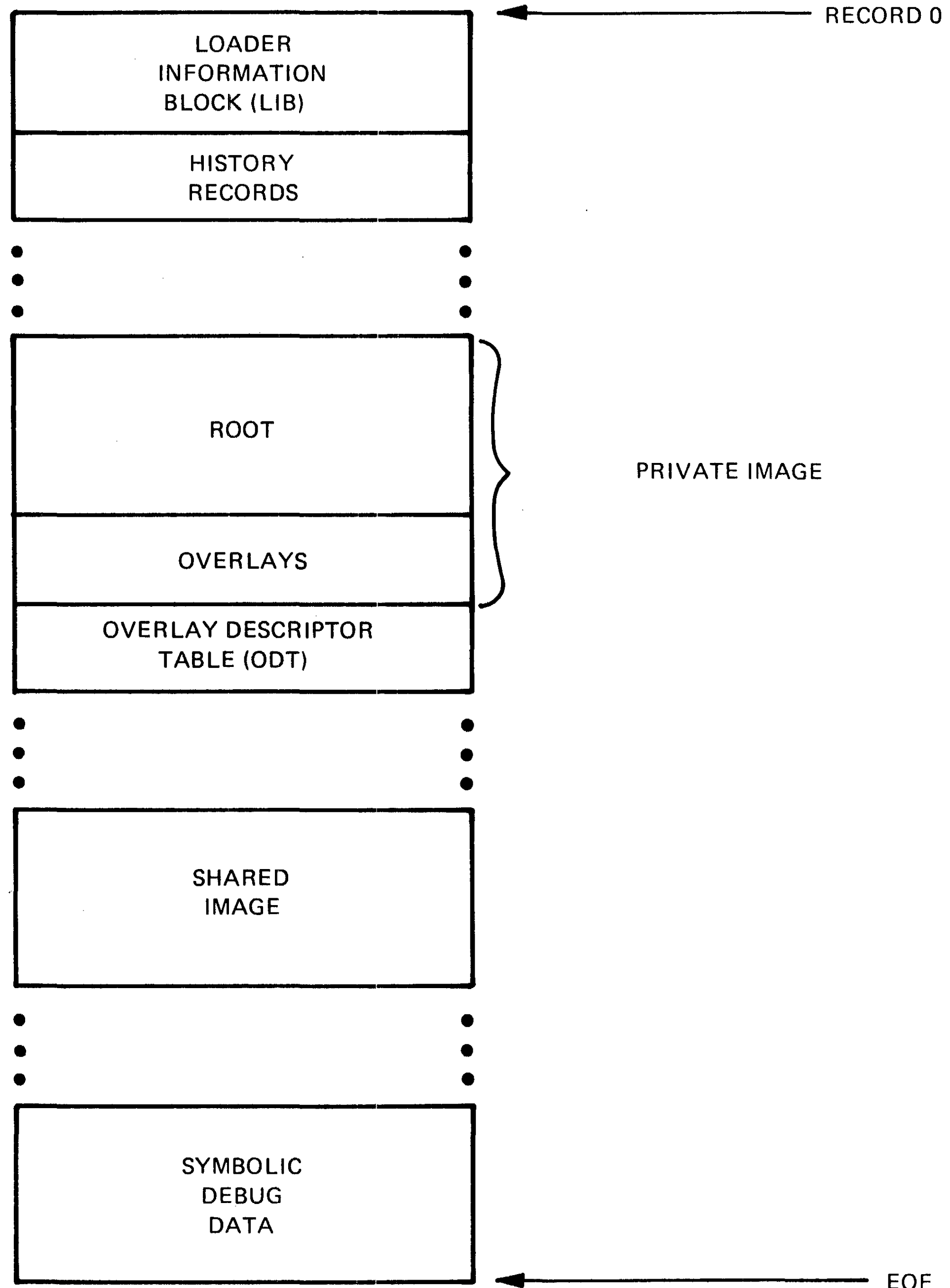


Figure 2-2 Task Image File Format for a Segmented Task

Each user who loads the task is provided with a copy of the private image. The first segment of the private image is known as the root segment. The root contains the primary task workspace, the impure code, the user-dedicated location (UDL), and, if the task is nonsegmented, the pure code. In addition, any absolute code found in the object modules is located in the root.

Within the UDL is the task status word (TSW). Each task has an active TSW which defines the enabled state of task interrupts and task queue entries as well as the current program location. (See Chapter 3.) The TSW should not be confused with the program status word (PSW). The TSW is a convention of an OS/32 task, while the PSW is a convention of a processor.

If a task is to use overlays; i.e., after the task is loaded, certain subroutines (overlays) are to remain in the image file and be fetched into the root as needed, they are formatted in the private image overlay area following the root. Link is instructed to construct overlays through the OVERLAY command.

The overlay descriptor table (ODT) following the overlay area contains instructions that tell the loader when to load the overlays into memory. The ODT is loaded into the task control block (TCB) after the task is loaded. In a multitasking system, each loaded task is assigned a TCB in dynamic system space. All task status information is stored in the TCB during task execution.

If the task is segmented, all pure code from the object modules is placed in the shared image segment of the image file. This area has only read and execute access privileges. When the first copy of the segmented task is loaded into memory, both a private and a shared image segment are created. If more than one user loads the task concurrently, each user is given a copy of the private image, but they all share the first copy of the shared image. Hence, only one copy of the shared image remains in memory during multiple simultaneous executions of the task.

If the task is to be debugged using the Perkin-Elmer Symbolic Debugger (DEBUG/32), Link places task data required by the debugger following the shared image segment. This data remains in the image file during task execution so that it is always available for use by the debugger.

A task may require access to subroutines or data areas in addition to those created by the programmer and contained in the task's object modules. OS/32 supports two types of external code and data. One type is an object module such as the FORTRAN or Pascal Run-Time Library (RTL). Routines in object libraries are included in a task's root segment or shared segment using the Link LIBRARY command.

The other type of external code or data is called a partial image. A partial image may consist of code (e.g., an RTL routine) or data (e.g., a shared common block). Partial images are built by separate runs of the linkage editor, and each partial image exists in its own image file. A partial image is included in a task's address space by the Link RESOLVE command. In addition, an uninitialized shared common image can be created in memory either by the TCOM command at system generation (sysgen) or by the OS/32 operator TCOM command.

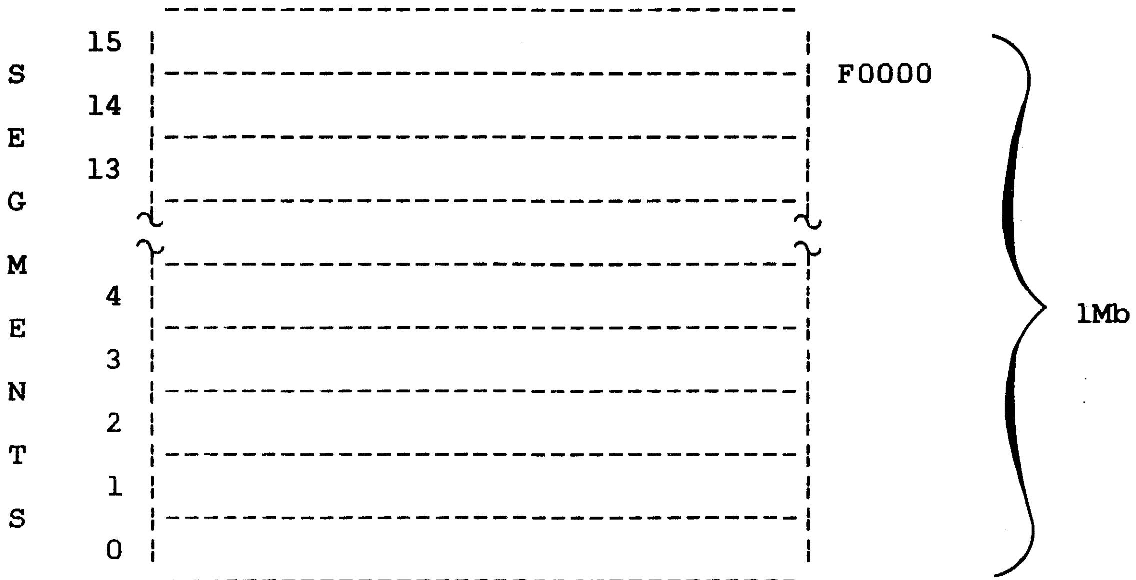
2.3 LOADING A USER-TASK (U-TASK) INTO MEMORY

In a multitasking system, u-tasks loaded into memory must be prevented from executing code in common data areas as well as any of the privileged instructions designated for the exclusive use of the operating system; e.g., input/output (I/O) and processor state change instructions. Likewise, a u-task needs protection from other tasks which might attempt to interfere with its execution. The relocation/protection hardware provides this protection.

Perkin-Elmer processors use one of two types of relocation/protection hardware. The Models 7/32, 8/32, and 3220 processors use the memory access controller (MAC); the Models 3210, 3230, 3240, 3250 and 3200MPS processors use the memory address translator (MAT).

When a u-task is loaded into memory, the relocation/protection hardware automatically allocates the first relative address in the task's root segment to the task's first physical address in memory. To the programmer, the task appears to be loaded at location 0 in memory. Actually, the MAC or MAT maps a range of task logical addresses into the available physical memory addresses. Thus, any program address referred to during program execution is translated and relocated to the correct physical address before memory is accessed.

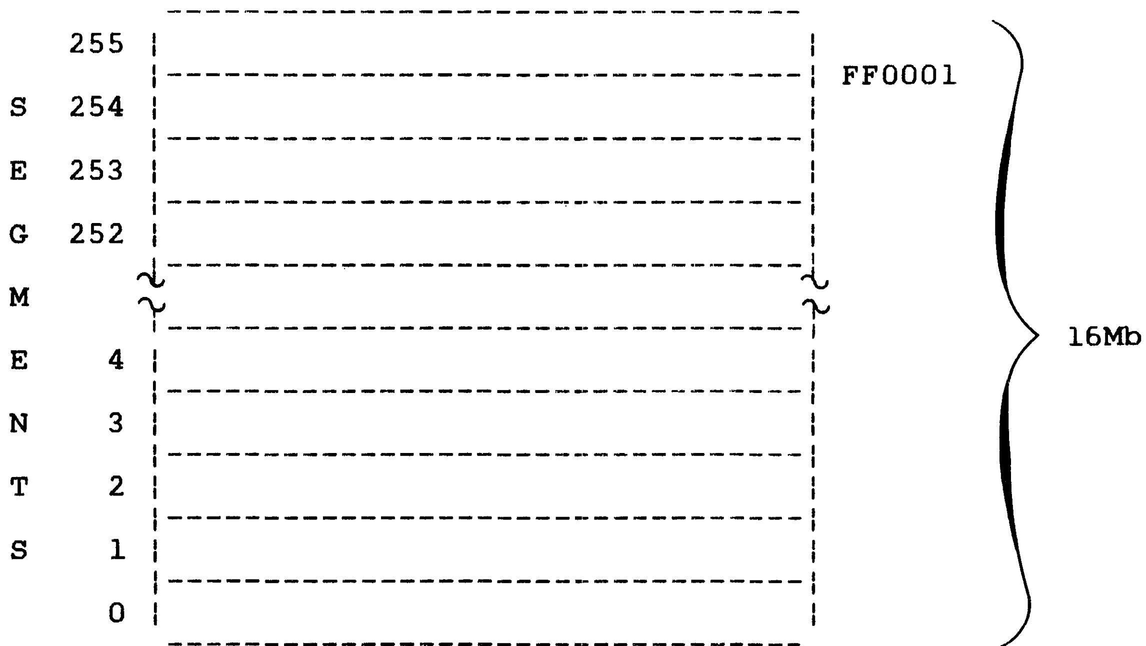
The range of addresses mapped for each task make up the u-task logical address space. Figures 2-3 and 2-4 show how the Perkin-Elmer relocation/protection hardware maps the u-task address space into segments. As shown in Figure 2-3, each segment mapped by MAC starts on a 256-byte boundary. Up to a maximum of sixteen 64kb segments are allocated by MAC for each task, providing a maximum task address space of 1Mb. Each segment is further divided into 256 pages. A page is a set of 256 contiguous one-byte locations beginning on an even 256-byte boundary. MAC locates the first address of each segment of the task on a 256-byte boundary; e.g., 00000, 10000, up to F0000.



NOTE

Each MAC segment consists of 256 256-byte pages or 64kb.

Figure 2-3 Task Address Space on a MAC Machine



NOTE

Each MAT segment consists of 32 2048-byte pages or 64kb.

Figure 2-4 Task Address Space on a MAT Machine

Figure 2-4 illustrates how MAT describes the u-task address space. On MAT machines, a maximum of 256 64kb segments or 16Mb are available for each u-task. Each segment is divided into 32 2,048-byte pages. MAT locates the first address of each segment of the task on an even 2,048-byte boundary.

As described in Section 2.2, a task may reference partial images resolved when the task is link-edited. Based on information recorded by Link in the LIB, the loader will ensure that the required partial images are in memory and automatically load any that are not. The partial images are then mapped into the appropriate ranges of the task's logical address space.

If the image is formatted as a segmented task, the task is loaded into its address space as two distinct (possibly discontiguous) areas, private and shared, as shown in Figure 2-5. Every task has a private area. This area contains the private image code (root, UDL, plus any overlay areas required by the task). The relocation/protection hardware starts loading this code at the beginning of segment 0 in the task address space.

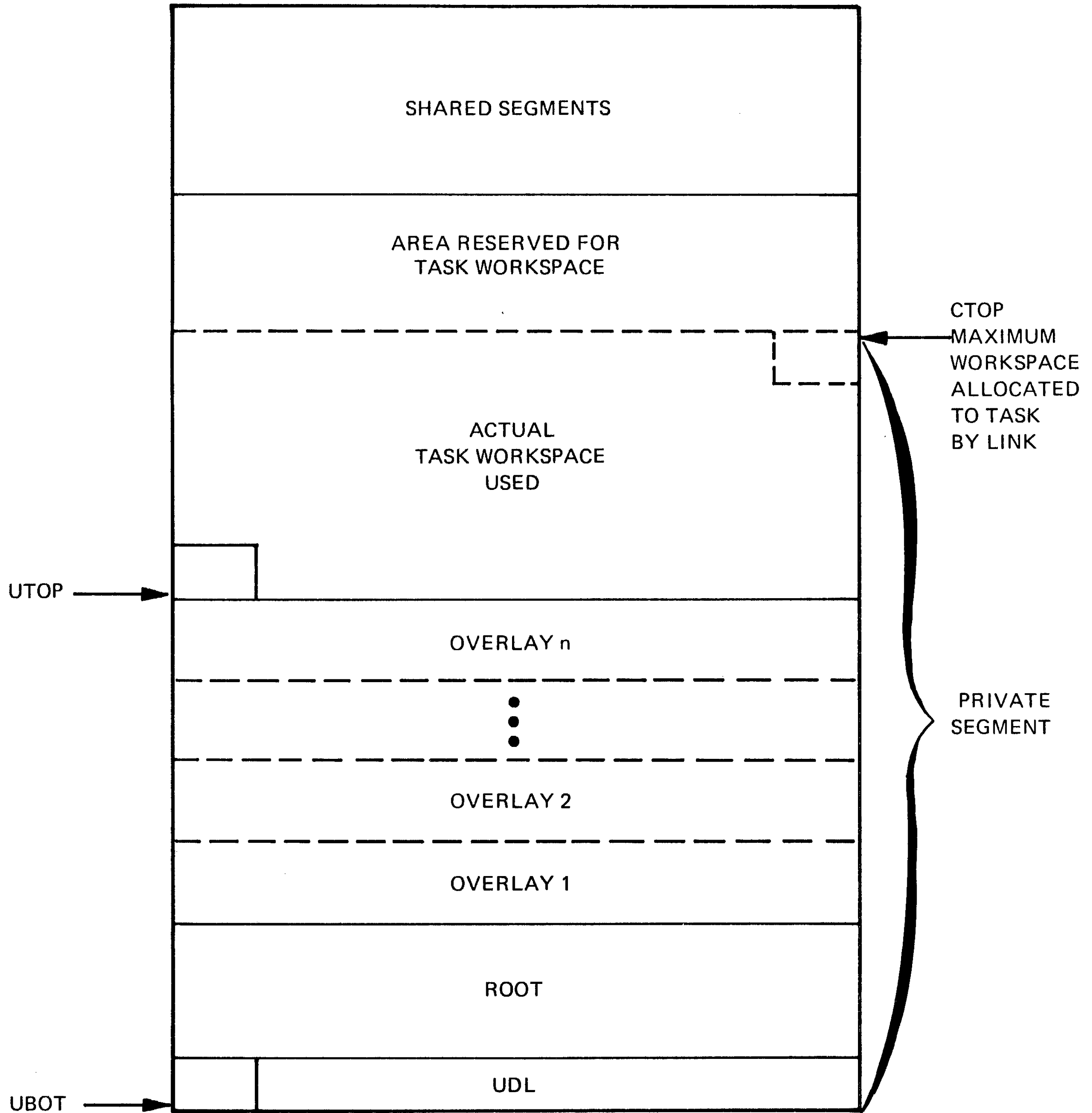


Figure 2-5 Segmented Task Loaded into Memory

The relative task address of the first fullword of the private area is called UBOT. For u-tasks, UBOT is always 0. Starting at UBOT, the loader loads the UDL into the first 256 bytes of segment 0. Following the UDL is the root node, which consists of segments that hold the user program code that cannot be shared by other tasks in the system.

If a task is to be executed using overlays, segments mapped out for these overlays are placed sequentially above the root. Above the overlays, the loader reserves a workspace area for the task. For example, some tasks require workspace to build and store symbol tables during execution. The amount of workspace that is reserved for a task is determined by the workspace parameters given in the OPTION WORK command when the task is built. These parameters are UTOP and CTOP. UTOP is the address of the first byte of the task workspace; CTOP is the address of the last addressable halfword in the task workspace. To override CTOP, specify a larger workspace area in the operator LOAD command when the task is loaded.

The pure code of a segmented task is loaded into the segments directly above the reserved task workspace. As noted in Section 2.2, if two or more users load the task concurrently, the copy of the shared segment loaded into memory for the first task is mapped into the logical address space of each of the later tasks. For each task to receive its own copy of the pure code from the image file, the task should be link-edited with the NSEGMENTED task option enabled. This option causes the pure code to be loaded in the root.

2.4 TASK STATES AND PRIORITIES

In a multitasking system, the task scheduler enforces the scheduling algorithm that determines which loaded task should be executed next. Tasks are scheduled according to the level of priority assigned to them. The OS/32 task scheduler can accommodate 255 levels of task priorities ranging from 1 through 255, with priority 1 being the highest. Several tasks can exist at the same priority level.

An executing task is said to be in the current state. Other tasks that have been started, but which are lower in priority than the executing task, are in the ready state. The task scheduler initiates execution of the highest priority ready task.

If the executing task becomes suspended, (either by the operating system, the operator, or another task), the task is said to be in the wait state. For example, a task becomes suspended when it requests a service from the operating system, such as an I/O transfer or get the time of day. This task remains in a wait state until the operation is completed, at which time the task enters the ready state. Table 2-1 lists the conditions under which a task can become suspended.

TABLE 2-1 TASK WAIT STATES

WAIT STATE	MEANING
I/O wait	Wait for an I/O operation to complete
Connection wait	Wait for a system resource
Timer wait	Wait for an interval to elapse or for a particular time of day to occur
Trap wait	Wait for a task handled trap to occur
Load wait	Wait for a requested load operation to complete
Task wait	Wait to be continued by another task
Roll wait	Wait to be rolled out
Terminal wait	Wait for I/O to complete to a terminal device (applies to terminal tasks only)
I/O block wait	Wait for an I/O block to be freed when task reaches its I/O control block limit
Accounting wait	Counters overflowed; task waiting for accounting facility to collect accounting data and remove wait
Intercept wait	Wait for a supervisor call (SVC) to be executed
Console wait	Wait for system operator, user, or another task to instruct an interrupted task to continue execution
Dormant wait	Wait for system operator, user, or another task to initiate a task. After a task is loaded, it enters the dormant state and remains there until execution is initiated. When a resident task goes to end of task, it reenters the dormant state.

A special wait state is the dormant state. When a task is loaded, it is initially in the dormant state. When a task is started (either by the OS/32 START command or another task), the task is removed from the dormant state and placed in the ready state. Once a task has been started, it can only become dormant again if it is a resident task; i.e., the task remains in memory after it reaches end of task. Because it enters the dormant state after execution, a resident task can be made ready through the OS/32 START command.

Nonresident tasks cannot reenter the dormant state after execution because they are removed from memory at end of task. Hence, nonresident tasks must always be loaded and placed in the dormant state before they can be started. A task can be made resident or nonresident by specifying these task options in the OPTION command when the task is built.

Nonresident tasks can enter the rolled state. A task becomes rolled when the task scheduler writes the task's private image segments to disk to make room for a higher priority task. A rolled task enters the ready state as soon as it becomes the highest priority rolled task and sufficient memory is available to accommodate it.

OS/32 control of task states during and after task execution is illustrated in Figure 2-6.

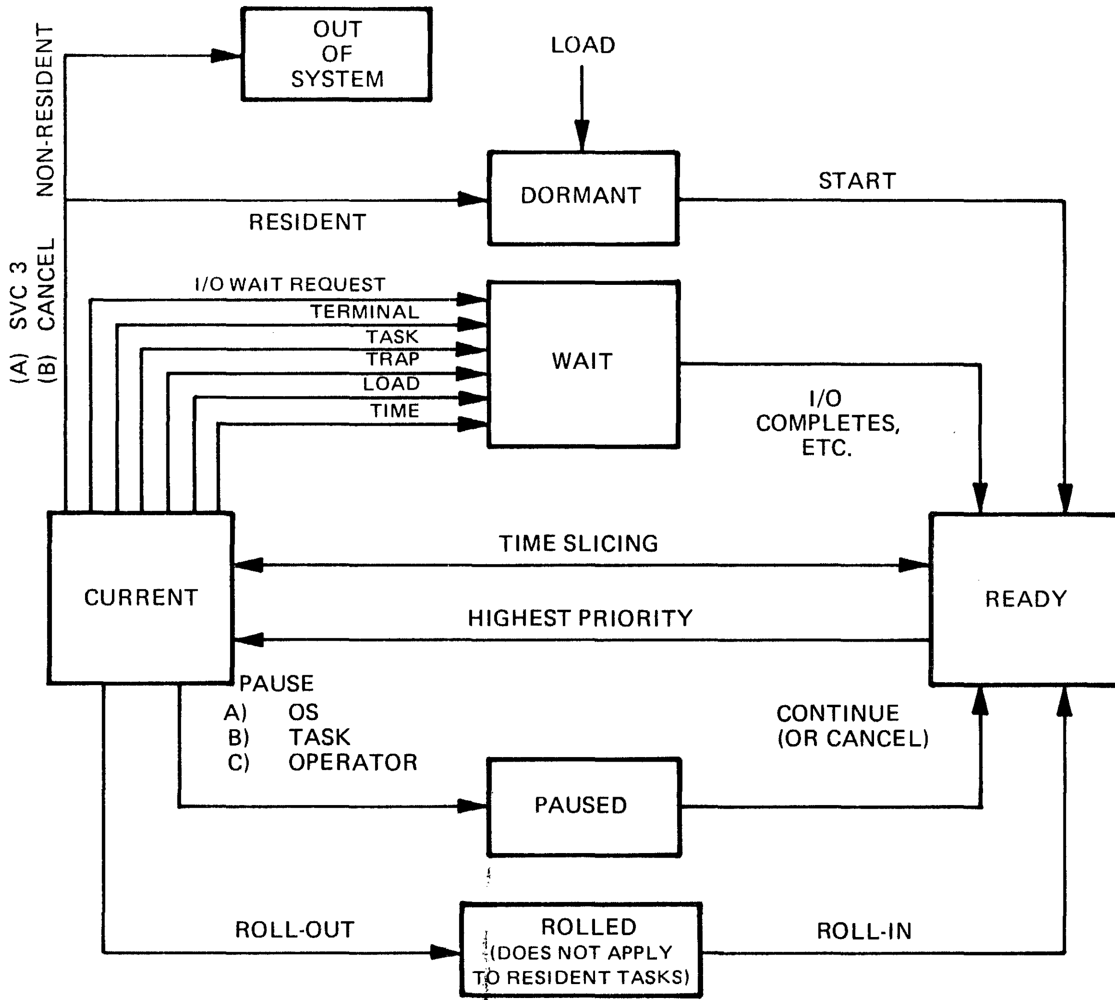


Figure 2-6 Task States

2.5 MONITOR TASKS

In addition to the OS/32 task scheduler, execution of a task can be controlled by another task called a monitor. Tasks that are placed under monitor control are called subtasks. A monitor creates the operating environment within which its subtasks are executed. For example, a monitor can:

- load, start, cancel or suspend any of its subtasks,
- override the task options that had been set when the subtasks were linked, or
- make logical unit (lu) assignments for any of its subtasks.

A task can become a monitor by issuing a call (SVC 6) to an OS/32 executor routine that allows a task to communicate with and control another task. The number of subtasks that can be assigned to a single monitor is unlimited (within the 252 user-written tasks supported).

The OS/32 routine called by SVC 6 causes the operating system to keep the monitor informed of the status of its subtasks; e.g., when the subtasks have been started, suspended, released, rolled out, etc. When a monitor goes to end of task, all of its subtasks are forced to end of task.

2.5.1 The OS/32 Multi-Terminal Monitor (MTM)

All tasks loaded and started in an OS/32 time-sharing environment execute as subtasks of MTM. MTM subtasks run at a maximum priority of at least one less than the priority of MTM.

Both interactive and batch processing are supported by MTM. Up to 64 interactive tasks can be executed concurrently, one from each MTM terminal. The number of batch jobs that can execute concurrently is determined by the operator during MTM system start-up and is a maximum of 64 minus the number of interactive terminals. Any batch jobs submitted above this number are queued by MTM. See the OS/32 Multi-Terminal Monitor (MTM) Reference Manual for more information on MTM.

2.6 RESTRICTIONS ON INTERTASK COMMUNICATION

OS/32 places some restrictions on which tasks can communicate with one another by assigning a group ID to each task. Normally, a task can communicate only with tasks within its assigned group.

Group IDs are assigned according to the operating environment under which a task is loaded. Tasks loaded into an OS/32 real-time environment are divided into two groups: foreground and background. A monitor and its subtasks are assigned to their own group. System tasks (e.g., the console monitor, the command processor, MTM and the Spooler) are in a separate group called the systems group.

To communicate with tasks outside its group, a foreground task should be link-edited with the UNIVERSAL task option enabled. OS/32 defines a background task as nonuniversal to prevent it from communicating with tasks outside its group.

A task monitor determines whether any of its subtasks can communicate outside the monitor's group. For example, all MTM subtasks are loaded with the communication task options specified for their accounts via the authorized user file (AUF), regardless of the task options chosen when the subtasks were built. See the OS/32 Multi-Terminal Monitor (MTM) System Planning and Configuration Guide for information on MTM sysgen options and the specification of options for MTM accounts.

2.7 ACCESSING OS/32 SYSTEM SERVICES

A u-task can access all of the nonprivileged system services that are available through OS/32. Tasks communicate with the operating system through structures that the task builds within its task address space. OS/32 uses the information stored in these structures to perform the services requested by the task.

One structure that is of particular importance in a real-time environment is the UDL. Chapter 3 examines the UDL and its use in handling program interrupts.

CHAPTER 3
INTERRUPT SERVICING IN A REAL-TIME ENVIRONMENT

3.1 INTRODUCTION

Real-time application systems are often designed to interrupt task execution when certain events occur. For example, if program output would be invalidated when execution of an instruction causes a floating point overflow condition, the programmer would want to know if such an event occurred. Otherwise, the programmer could not be certain that the results of the program were valid. The mechanism that informs the task that such an event has occurred is called a task trap.

Traps suspend task execution at the location following the instruction that was executing when a trap-causing event occurred. Execution then continues in the routine that will handle the trap. The location counter (LOC) at the time of the trap is saved in the user-dedicated location (UDL) so that the interrupted execution can be resumed. In the preceding example, the program might continue in a trap handling routine that outputs a message giving the location of the instruction that caused the event. The routine might then suspend task execution until the task is either continued or cancelled by the operator.

Task execution can be continued only if the state of the task (accumulators, carry, current hardware interrupt mask, program counter, etc.) was saved by the operating system after the trap occurred. The task structure that contains the information to be saved by the operating system is the task status word (TSW).

3.2 TASK STATUS WORD (TSW)

The structure of the TSW is shown in Figure 3-1.

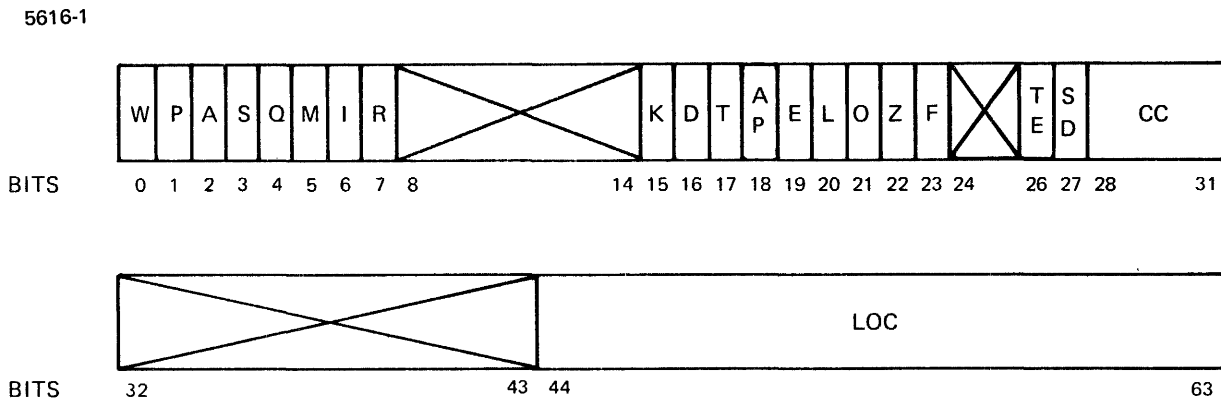


Figure 3-1 Task Status Word

As shown in Figure 3-1, the TSW occupies 8 bytes aligned on fullword boundaries. The first 28 bits of the TSW are used to enable or disable task traps for certain conditions. For example, if bit 2 is set, a task trap will occur when execution of an instruction results in an arithmetic fault.

The current condition code of the task is saved in the area comprised of bits 28 through 33. The program location counter (LOC) is contained in the area following bit 43. All TSW bit settings and their corresponding effects on task execution are listed in Table 3-1.

When a task is link-edited, its TSW is initialized. A Link initialized TSW has all task traps disabled and the task's starting address in the LOC field. The user can change the TSW default values initialized by Link by specifying the desired bit settings in the Link OPTION TSW command. See the OS/32 LINK Reference Manual for more information on this command.

The initialized TSW is placed in the task control block (TCB) when the task is loaded into memory. If all the task trap bits have been disabled, all trap-causing events will be handled by system default trap handlers.

TABLE 3-1 TSW BIT SETTINGS

BIT POSITION	MASK NAME	BIT NAME	EFFECT ON TASK
0(W)	TSW.WTM	Trap Wait	Suspends execution until a trap occurs.
1(P)	TSW.PWRM	Enable Power Restore Trap	Notifies task when power is restored after a power failure.
2(A)	TSW.AFM	Enable Arithmetic Fault Trap	Notifies task when an arithmetic fault occurs.
3(S)	TSW.S14M	Enable SVC 14 Trap	Notifies task when an SVC 14 is issued.
4(Q)	TSW.TSKM	Enable Task Queue Service Trap	Notifies task when an item is placed on the task queue.
5(M)	TSW.MAFM	Enable Memory Access Fault Trap	Notifies task when it attempts to access memory outside its task address space.
6(I)	TSW.IITM	Enable Illegal Instruction Trap	Notifies task when it attempts to execute an illegal instruction.
7(R)	TSW.DFFM	Enable Data/ Alignment Fault Trap	Notifies task when it attempts to execute an instruction that causes a data format or alignment fault.
8-14	-	Reserved	-
15(K)	TSW.SUQM	Queue Entry on Subtask Change	Notifies task of subtask state change by adding a 3-fullword entry to task queue.
16(D)	TSW.DIQM	Queue Entry on Device Interrupt	Notifies task of a device interrupt by adding a fullword entry to the task queue.

TABLE 3-1 TSW BIT SETTINGS (Continued)

BIT POSITION	MASK NAME	BIT NAME	EFFECT ON TASK
17(T)	TSW.TCM	Queue Entry on Task Call	Notifies task of an SVC 6 queue parameter call by adding a fullword entry to the task queue.
18(AP)	TSW.APTM	Queue Entry on Signal from APU	Notifies task of an APU signal to the CPU by adding a fullword entry to the task queue.
19(E)	TSW.PMM	Queue Entry on Task Message	Notifies task that a message has been sent to it by adding a fullword entry to the task queue.
20(L)	TSW.LODM	Queue Entry on Load Proceed	Notifies task that its subtask has been loaded by adding a fullword entry to the task queue.
21(O)	TSW.IOM	Queue Entry on I/O Completion	Notifies task that an SVC 1 I/O operation has completed by adding a fullword entry to the task queue.
22(Z)	TSW.TMCM	Queue Entry on Time-Out Completion	Notifies task that a specified time interval has elapsed by adding a fullword entry to the task queue.
23(F)	TSW.ITM	Queue Entry on Data Communica-tions Functions	Notifies task that an SVC 15 operation has been completed by adding a fullword entry to the task queue.
		Queue Entry on SVC 1 Buffer Transfer Completion	Notifies task that magnetic tape driver has added a buffer to the OUT-QUEUE by adding a fullword entry to the task queue.
24-25	-	Reserved	-

TABLE 3-1 TSW BIT SETTINGS (Continued)

BIT POSITION	MASK NAME	BIT NAME	EFFECT ON TASK
26(TE)	TSW.TESM	Enable Task Queue Event Service	Notifies task of an event through a task event trap. See OS/32 System Level Programmer Reference Manual.
27(SD)	TSW.SDM	Enable Queue Entry on Send Data Call	Notifies task that a message is being sent to it by adding a fullword entry to task queue.
28-31(CC)	-	Condition Code	-
32-40	-	Reserved	-
41-63(LOC)	TSW.LOC	Location counter	Address where task is to begin executing.

3.3 TRAPS HANDLED BY OS/32

Internal trap-causing events detected by the processor hardware are called faults. Five types of faults can be detected by the processor:

- Power restoration after power failure
- Arithmetic faults (see Table 3-2)
- Memory access faults (see Table 3-3)
- Illegal instruction faults
- Data format/alignment faults (see Table 3-4)

When a fault is detected by the processor, the currently executing task is suspended and control is given to the approprate default system trap handling routine. After power restoration, the trap handling routine suspends task execution until the task is continued or cancelled by the operator. When an arithmetic, memory access, illegal instruction or data format/alignment fault occurs, the trap handling routine generates a reason code that identifies the fault. This reason code is output to the console and task execution is suspended. Tables 3-2, 3-3, and 3-4 list the reason codes identifying each type of fault.

TABLE 3-2 ARITHMETIC FAULT TRAP-CAUSING EVENTS

EVENT	REASON CODE
Fixed point zero divide	X'00'
Fixed point quotient overflow	X'01'
Floating point zero divide	X'02'
Floating point exponent underflow	X'03'
Floating point exponent overflow	X'04'

TABLE 3-3 MEMORY ACCESS FAULT TRAP-CAUSING EVENTS

PROCESSOR	EVENT	REASON CODE
3220	SVC address error	X'00'
	Execute protect violation	X'01'
	Write/interrupt protect violation	X'02'
	Reserved	X'03'
	Write protect violation	X'04'
	Reserved	X'05'
	Reserved	X'06'
	Reserved	X'07'
	Segment number not present	X'08'
	Reserved	X'09'
	Program address is greater than segment limit fault (SLF)	X'0A'

TABLE 3-3 MEMORY ACCESS FAULT TRAP-CAUSING EVENTS (Continued)

PROCESSOR	EVENT	REASON CODE
3210	Reserved	X'00'
3230	Execute protect violation	X'01'
3240	Write protect violation	X'02'
3250	Read protect violation	X'03'
3200MPS	Access level fault	X'04'
	Segment limit fault	X'05'
	Nonpresent segment fault	X'06'
	Shared segment table (SST) size exceeded	X'07'
	Private segment table (PST) size exceeded	X'08'

TABLE 3-4 DATA FORMAT/ALIGNMENT FAULT TRAP-CAUSING EVENTS

EVENT	REASON CODE
Reserved	X'00'
Reserved	X'01'
Invalid sign digit, packed data	X'02'
Invalid data digit, packed data	X'03'
Reserved	X'04'
Reserved	X'05'
Fullword alignment fault	X'06'
Halfword alignment fault	X'07'

3.4 TRAPS HANDLED BY USER-WRITTEN TASKS

Before a task can handle a trap, the task must replace the link-initialized TSW in the TCB with a TSW in which the appropriate trap bits are set. Depending on which trap bits are enabled, a task can handle traps caused by the following events:

- arithmetic faults
- data format/alignment faults
- power restoration
- illegal instruction faults
- memory access faults
- events that add entries to the task queue
- user-defined trap-causing events

If the trap bit for one of the internal fault conditions is enabled, execution control is transferred to the user-written trap handling routine when that internal fault condition occurs. If the internal fault trap bit is disabled, execution control is transferred to the OS/32 trap handling routine as explained in Section 3.3. A trap caused by an event that adds items to the task queue is handled by a task queue trap. User-defined trap-causing events are handled by an SVC 14 task trap.

3.4.1 Task Queue Trap-Causing Events

A task queue is in the form of a standard Perkin-Elmer circular list as shown in Figure 3-2.

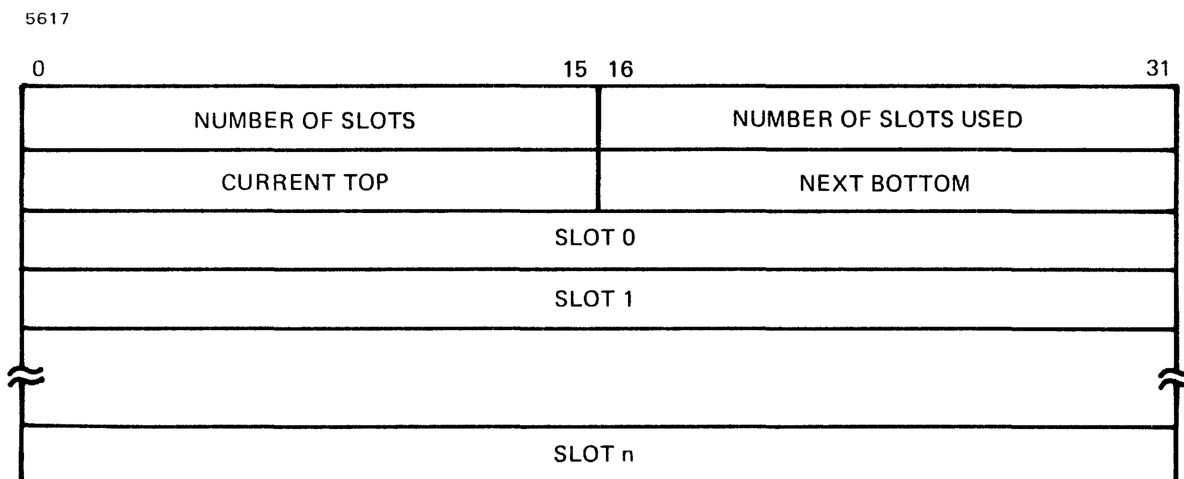


Figure 3-2 Perkin-Elmer Standard Circular List

The first four halfwords of the circular list make up the list header that contains the list parameters. Immediately following the header is the list itself. The first fullword in the list is designated slot 0. The remaining slots are numbered sequentially from 1 up to a maximum of X'FFFE'. Hence, a task queue can contain a maximum of 65,535 fullword slots.

When a task queue trap occurs, entries are added to the bottom of the list, ending with slot 0. The user-supplied task queue trap handling routine should always remove entries from the top of the queue.

Table 3-5 lists the fullword entries added to the list by the task queue trap-causing events.

TABLE 3-5 TASK QUEUE TRAP-CAUSING EVENTS

EVENT	TASK QUEUE FULLWORD ENTRY	
	1-BYTE REASON CODE	3-BYTE PARAMETER
Device interrupt	X'00'	Associated with device
SVC 6 queue parameter	X'01'	Specified by the task that issued the SVC 6
SVC 6 send data	X'04'	A(send data message buffer)
APU signals CPU	X'05'	APU number, status, and error code. See Figure 3-4.
Message received	X'06'	A(message in ring)
Load proceed completion	X'07'	A(SVC 6 parameter block)
I/O proceed completion	X'08'	A(SVC 1 parameter block)
Timer completion	X'09'	Time interval specified by the SVC 2 code 23 parameter block
SVC 15 command completion	X'0A'	A(SVC 15 parameter block)
SVC 15 buffer completion	X'0B'	A(SVC 15 parameter block)
SVC 1 buffer transfer completion	X'0B'	A(SVC 1 parameter block)

TABLE 3-5 TASK QUEUE TRAP-CAUSING EVENTS (Continued)

EVENT	TASK QUEUE FULLWORD ENTRY 1-BYTE REASON CODE	3-BYTE PARAMETER
SVC 15 termination	X'0C'	A(SVC 15 parameter block)
SVC 15 halt I/O	X'0D'	A(SVC 15 parameter block)
ZDLC buffer input	X'0E'	A(UDR list)
ZDLC buffer output	X'0F'	A(UDW list)
ZDLC error condition	X'10'	A(information block)
ZDLC buffer error	X'11'	A(UQR list)
EMT 3270 unsolicited host input	X'18'	-
EMT 3270 unrequested disconnect	X'19'	-
EMT switched line connect timeout	X'1A'	-

NOTE

For more information on the OS/32 supervisor routines that initiate task queue trap-causing events, see the OS/32 Supervisor Call (SVC) Manual.

Task queue entries that are added to a monitor's task queue when its subtask experiences a state change are shown in Figure 3-3. Note that a subtask state change adds 3 fullword entries to the bottom of the queue. The first fullword consists of a 1-byte reason code (X'02'), a 1-byte subtask reason code (see Table 3-6), and other subtask information items. The remaining fullword slots contain the name of the subtask.

5618

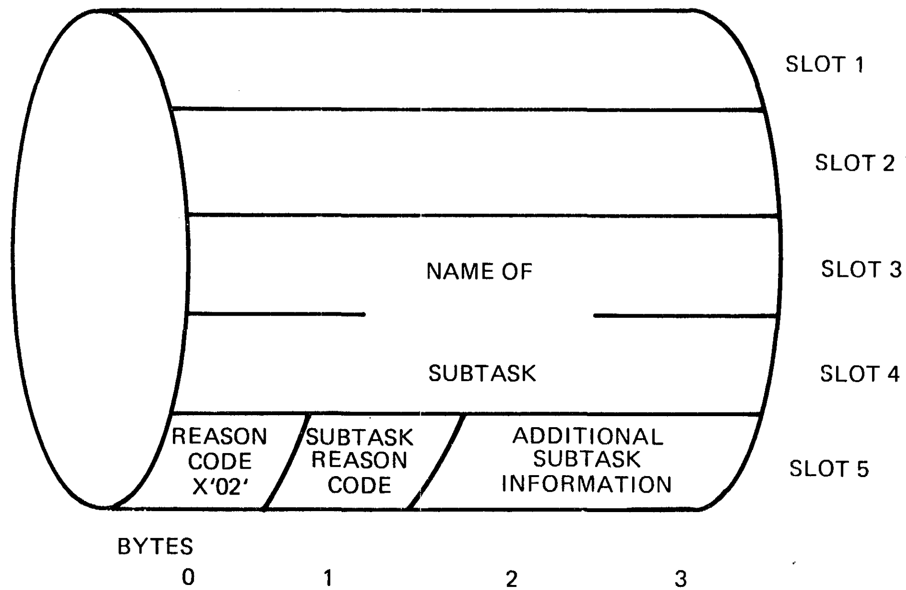


Figure 3-3 Circular List with Task Queue Entries for Subtask State Change

TABLE 3-6 SUBTASK REASON CODES AND CORRESPONDING STATE CHANGES

SUBTASK REASON CODE	SUBTASK STATE CHANGE
0	End of task; bytes 2 and 3 are binary end of task codes
1	Paused
2	Continued
3	Suspended
4	Released
5	Rolled out
6	Rolled in
7	Started by a task other than the monitor
8	Accounting overflow (MTM only)

In a Model 3200MPS System, a task can receive a task queue trap from an auxiliary processing unit (APU). The task must first be connected to the APU through the SVC 6 CONNECT function, and the APU must be enabled for interrupts through the SVC 6 THAW function. See Section 3.6.3. Figure 3-4 shows the format of the task queue entry when an APU in a Model 3200MPS System signals the central processing unit (CPU). Note that the 3-byte parameter consists of three items: APU number, APU status, and APU error status. See the OS/32 System Level Programmer Reference Manual for more information on the values that are returned to the task queue for reason code 5.

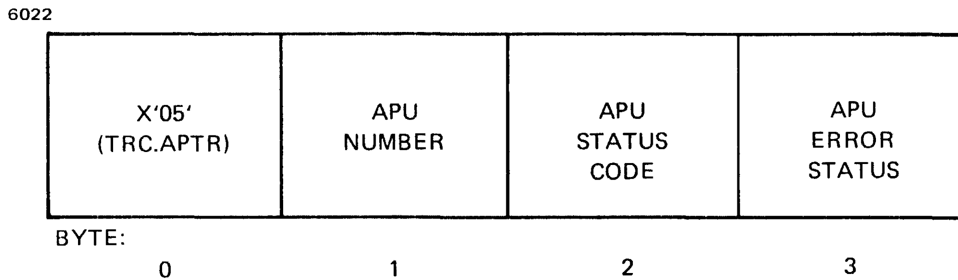


Figure 3-4 Task Queue Entry for APU Signal

3.4.2 User-Defined Trap-Causing Events

The OS/32 supervisor routine called by SVC 14 allows the programmer to define trap-causing events for a task. SVC 14 suspends task execution, saves the current state of the task, and transfers execution to the SVC 14 task trap handling routine.

One argument can be specified when SVC 14 is called. This argument can point to a memory location that contains a reason code for the user-defined trap-causing event. See the OS/32 Supervisor Call (SVC) Reference Manual for more information on using SVC 14.

3.5 THE TASK STATUS WORD (TSW) SWAP

A task that services traps requires a data structure that can be used to save the current TSW after a trap occurs. The OS/32 data structure that is reserved for this purpose is the UDL shown in Figure 3-5.

5921-2

Address	Contents
0 (00)	CTOP (UDL.CTOP)
4 (04)	UTOP (UDL.UTOP)
8 (08)	UBOT (UDL.UBOT)
12 (0C)	DATA MANAGEMENT SYSTEM (UDL.DMS)
16 (10)	A (TASK QUEUE) (UDL.TSKQ)
20 (14)	A (SEND DATA FREE BUFFER QUEUE) (UDL.SDQ)
24 (18)	A (MESSAGE RING) (UDL.MSGR)
28 (1C)	A (SVC 14 ARG) (UDL. SV14)
32 (20)	RESERVED (UDL.EXT1)
36 (24)	RESERVED (UDL.EXT2)
40 (28)	LOAD MULTIPLE STARTING ADDRESS (UDL.LMSA)
44 (2C)	RESERVED
48 (30)	POWER RESTORATION OLD TSW
52 (34)	(UDL.PWRO)
56 (38)	POWER RESTORATION NEW TSW
60 (3C)	(UDL.PWRN)
64 (40)	ARITHMETIC FAULT OLD TSW
68 (44)	(UDL.ARFO)
72 (48)	ARITHMETIC FAULT NEW TSW
76 (4C)	(UDL.ARFN)
80 (50)	RESERVED / 81(51) DATA FORMAT REASON CODE (UDL.DFFR) / 82 (52) MAC MAT FAULT REASON CODE (UDL.MAFR) / 83 (53) ARITH FAULT REASON CODE (UDL.ARFR)
84 (54)	ARITHMETIC FAULT, NEXT INSTRUCTION ADDRESS (UDL.ARFX)
88 (58)	DATA/ALIGNMENT, ACTUAL FAULT ADDRESS (UDL.DFFX)
92 (5C)	MAC/MAT FAULT, ACTUAL FAULT ADDRESS (UDL.MAFL)
96 (60)	SVC 14 OLD TSW
100 (64)	(UDL.S14O)
104 (68)	SVC 14 NEW TSW
108 (6C)	(UDL.S14N)
112 (70)	TASK QUEUE SERVICE OLD TSW
116 (74)	(UDL.TSKO)
120 (78)	TASK QUEUE SERVICE NEW TSW
124 (7C)	(UDL.TSKN)
128 (80)	MEMORY ACCESS FAULT OLD TSW
132 (84)	(UDL.MAFO)
136 (88)	MEMORY ACCESS FAULT NEW TSW
140 (8C)	(UDL.MAFN)
144 (90)	ILLEGAL INSTRUCTION OLD TSW
148 (94)	(UDL.IITO)
152 (98)	ILLEGAL INSTRUCTION NEW TSW
156 (9C)	(UDL.IITN)
160 (A0)	DATA FORMAT FAULT OLD TSW
164 (A4)	(UDL.DFFO)
168 (A8)	DATA FORMAT FAULT NEW TSW
172 (AC)	(UDL.DFFN)
176 (B0)	RESERVED
180 (B4)	
184 (B8)	POINTER TO SYSTEM NETWORK ARCHITECTURE TABLE (UDL.SNA)
188 (BC)	SAVE AREA USED BY SYSTEM NETWORK ARCHITECTURE(UDL.RSAV)
192 (C0)	RESERVED FOR AIDS
196 (C4)	
≈	≈
248 (F8)	
252 (FC)	

Figure 3-5 User-Dedicated Location

Note that the UDL is divided into fields that are used to store the TSW. Each type of trap requires two TSW save areas, the old TSW (OTSW) field and the new TSW (NTSW) field. The operating system uses the OTSW field for saving the TSW that is in the TCB when the trap occurs. The NTSW field is used by the task to store the TSW that contains the address of the user-written trap handling routine and the new TSW bits to be enabled during execution of the routine.

When a task is link-edited, a UDL is initialized in locations 0 through 255 of the task's private image segment. After a task with a link-initialized UDL is loaded, a TSW containing the task's starting address and all trap bits disabled is placed into the TCB for that task. This task cannot handle traps until it loads an NTSW in the UDL and loads a new TSW into the TCB.

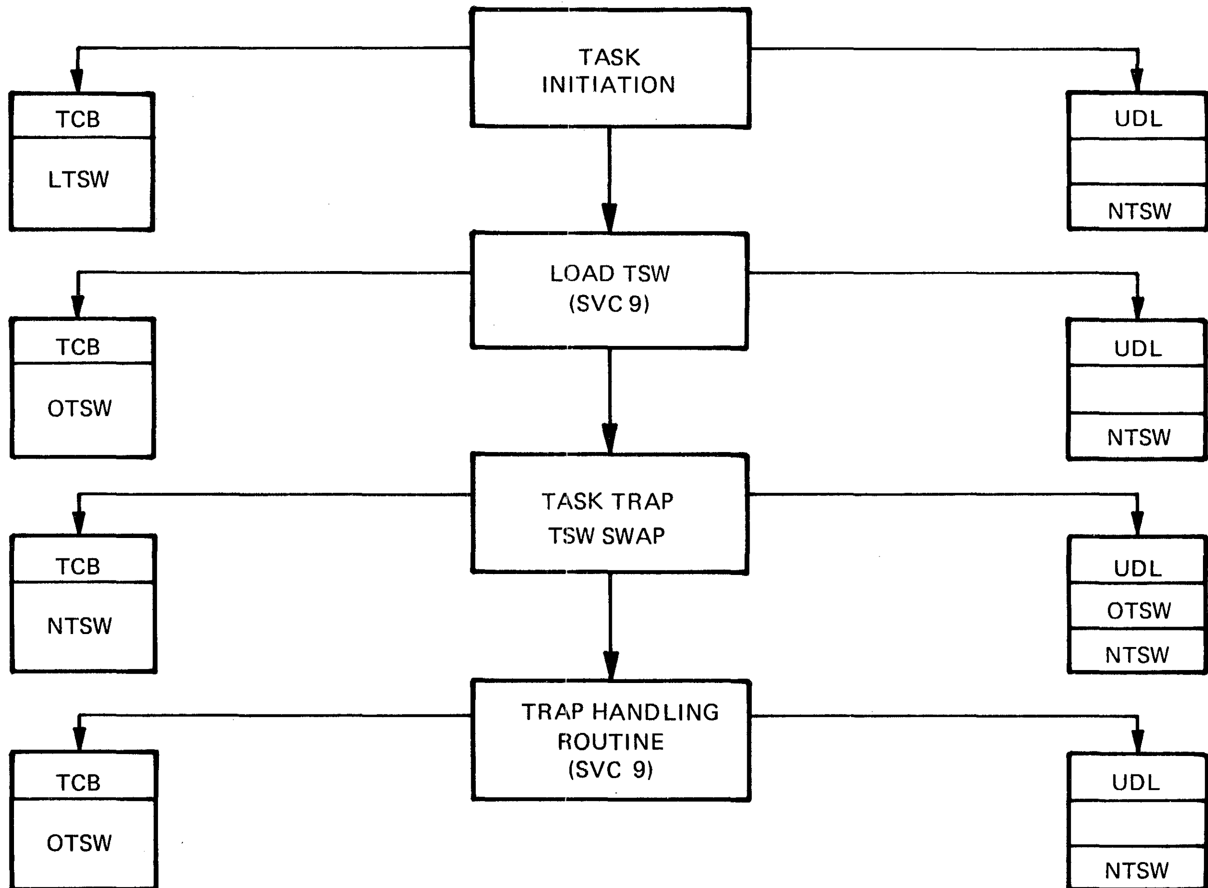
NOTE

> Users can initialize the UDL by assembling the appropriate code and using the Link OPTION ABSOLUTE=0 command before including the UDL in their task.

Figure 3-6 shows the effect of the TSW swap on the TCB. Note that for a task trap to occur, a TSW swap is performed at least three times during task execution:

- after the task is started,
- when the task trap occurs, and
- at the end of a trap handling routine.

5621



LTSW – TSW INITIALIZED BY LINK
OTSW – TSW INITIALIZED BY TASK WITH TASK EXECUTION ADDRESS AND TRAP BIT SET
NTSW – TSW WITH ADDRESS OF TRAP HANDLING ROUTINE

Figure 3-6 Task Status Word Swap

To perform the TSW swap, the task and the trap handling routine issues a call (SVC 9) to an OS/32 supervisor routine that replaces the current TSW in the TCB with the TSW specified as an argument to SVC 9.

As shown in Figure 3-6, the TSW swap performed after the task is started replaces the link-initialized TSW in the TCB with the task-initialized TSW (OTSW). This swap causes task execution to resume at the location specified by OTSW; or, if the LOC of OTSW is zero, execution resumes after the SVC 9 instruction.

The TSW swap performed by the OS/32 trap mechanism replaces the OTSW in the TCB with a copy of the NTSW stored in the UDL. Task execution resumes at the address of the trap handling routine.

The TSW swap performed at the end of the trap handling routine replaces the NTSW in the TCB with the OTSW saved in the UDL. Execution resumes at the instruction that was about to be executed when the trap occurred.

In addition to the TSW swap areas, the UDL structure also reserves fields that are used to store information used by the trap handling routine; e.g., internal fault reason codes, task queue address, address of message buffers, etc.

The UDL fields that are used by a task to handle task traps are summarized in Table 3-7.

TABLE 3-7 UDL FIELDS USED TO HANDLE TASK TRAPS

BYTE LOCATION	FIELD NAME	MASK NAME	CONTENTS OF FIELD
16('10')	A(task queue)	UDL.TSKQ	Address of task queue.
20('14')	A(Send Data Free Buffer Queue)	UDL.SDQ	Address of free buffer queue for SVC 6 send data function.
24('18')	A(Message ring)	UDL.MSGR	Address of ring of 76-byte message buffers aligned on a fullword boundary.
28('1C')	A(SVC 14 Arg)	UDL.SV14	Effective address of the argument to an SVC 14.
40('28')	Load Multiple Starting Address	UDL.LMSA	Second operand of load multiple instruction that caused a memory access fault.
48('30')	Power Restoration Old TSW	UDL.PWRO	TSW saved by the OS/32 after a power failure occurs.
56('38')	Power Restoration New TSW	UDL.PWRN	TSW containing the address of the power restoration trap routine to which execution will branch when power is restored after a power failure occurs.
64('40')	Arithmetic Fault Old TSW	UDL.ARFO	TSW saved by the OS/32 when an arithmetic fault trap occurs. For the Perkin-Elmer 3200 Series processors, the LOC of this TSW points to the faulting instruction. For 7/32 and 8/32 processors, LOC points to the instruction after the faulting instruction.
72('48')	Arithmetic Fault New TSW	UDL.ARFN	TSW containing the address of the arithmetic fault trap handling routine to which execution will branch when an arithmetic fault occurs.
81('51')	Data Format Reason Code	UDL.DFFR	Reason code indicating the event that caused the data format/alignment fault. See Table 3-4.
82('52')	MAC/MAT Fault Reason Code	UDL.MAFR	Reason code indicating the event that caused the memory access fault. See Table 3-3. This reason code is given only for a memory access controller (MAC) or memory address translator (MAT) fault occurring on a Series 3200 processor.

TABLE 3-7 UDL FIELDS USED TO HANDLE TASK TRAPS (Continued)

BYTE LOCATION	FIELD NAME	MASK NAME	CONTENTS OF FIELD
83('53')	Arithmetic Fault Reason Code	UDL.ARFR	Reason code indicating the event that caused the arithmetic fault. See Table 3-2. This reason code is given only for an arithmetic fault occurring on a Series 3200 processor.
84('54')	Arithmetic Fault, Next Instruction Address	UDL.ARFX	Address of the instruction following the instruction that caused an arithmetic fault. This address is given only for an arithmetic fault occurring on a Series 3200 processor.
88('58')	Data/Alignment, Actual Fault Address	UDL.DFFX	Address of the memory location referred to by the instruction that caused the data format or alignment fault. This address is given only for a data/alignment fault occurring on a Series 3200 processor.
92('5C')	MAC/MAT Fault, Actual Fault Address	UDL.MAFL	Address of the data or instruction that caused a memory access fault trap. This address is given only for a MAC/MAT fault occurring on a Series 3200 processor.
96('60')	SVC 14 Old TSW	UDL.S14O	TSW saved by OS/32 when an SVC 14 trap occurs.
104('68')	SVC 14 New TSW	UDL.S14N	TSW containing the address of the SVC 14 trap handling routine to which execution branches when an SVC 14 trap occurs.
112('70')	Task Queue Service New TSW	UDL.TSKN	TSW containing the address of the task queue trap handling routine to which execution branches when a task queue trap occurs.
128('80')	Memory Access Fault, Old TSW	UDL.MAFO	TSW saved by OS/32 when a memory access fault trap occurs.
136('88')	Memory Access Fault, New TSW	UDL.MAFN	TSW containing the address of the memory access fault trap handling routine to which execution branches when a memory access fault trap occurs.
144('90')	Illegal Instruction,Old TSW	UDL.IITO	TSW saved by OS/32 when an illegal instruction fault trap occurs.
152('98')	Illegal Instruction, New TSW	UDL.IITN	TSW containing the address of the illegal instruction trap handling routine to which execution will branch when an illegal instruction fault trap occurs.
160('A0')	Data Format Fault Old TSW	UDL.DFFO	TSW saved by OS/32 when a data format or alignment fault trap occurs. This TSW is saved only for a data format/alignment fault trap occurring on a Series 3200 processor.
168('A8')	Data Format Fault New TSW	UDL.DFFN	TSW containing the address of the data format/alignment fault trap handling routine to which execution branches when a data format or alignment fault trap occurs.

3.6 WRITING TASKS THAT HANDLE TASK TRAPS

A task cannot handle a trap until it has a TSW with the appropriate trap bits enabled in the TCB and a TSW with the address of the trap handling routine in the UDL.

The OS/32 system structure macro library, SYSSTRUC.MLB, provides macro instructions that automatically set up a UDL or TSW within a task's address space. The $UDL instruction defines the UDL structure; the $TSW instruction defines a TSW. To prepare a task to handle a trap, simply execute these instructions and set the appropriate fields or bit masks for the trap the task is to handle.

For example, suppose a program is to output its own message and pause each time it attempts to execute an illegal instruction. First, place the address of the task trap routine in the illegal instruction new TSW field (UDL.IITN) as follows:

Example:

```
         $UDL                    SETS UP UDL AND EQUATES
         ABS   0
MYUDL    DS    UDL
MYUDLE   EQU   *
         ORG   MYUDL+UDL.IITN    SET LOC TO FIELD
         DC    0                 DISABLES TASK TRAPS FOR NTSW
         DC    A(TRAP)           PLACES ADDRESS OF TRAP ROUTINE IN NTSW
         ORG   MYUDLE            SET LOC TO END OF UDL
TRAP     SVC   2,MSG
         SVC   2,PAUSE
```

Next, build a TSW with the TSW.IITM bit mask set.

Example:

```
         $TSW
NTSW     DC     TSW.IITM              SETS ILLEGAL INSTRUC TRAP BIT
         DC     0                     SETS LOC AT INSTRUC FOLLOWING SVC 9
```

After initiation, the task will issue an SVC 9 to load this TSW into the TCB as follows:

```
         SVC    9,NTSW
```

Table 3-8 summarizes the UDL fields and TSW bit masks that pertain to each type of task trap.

CAUTION

TASKS THAT HANDLE ARITHMETIC FAULT TRAPS MUST BE LINK-EDITED WITH THE NAFPAUSE TASK OPTION. WHEN AN ARITHMETIC FAULT TRAP OCCURS, NAFPAUSE PREVENTS OS/32 FROM PAUSING THE TASK SO THAT EXECUTION CONTINUES TO THE USER-WRITTEN TRAP HANDLING ROUTINE. SEE THE OS/32 LINK REFERENCE MANUAL FOR MORE INFORMATION.

TABLE 3-8 SUMMARY OF TASK STRUCTURES USED FOR HANDLING TRAPS

TRAP-CAUSING EVENT	UDL FIELDS	TASK QUEUE	TSW BIT MASK
APU Signals CPU****	UDL.TSKO UDL.TSKN UDL.TSKQ	Yes	TSW.APTM TSW.TSKM
Arithmetic Fault**	UDL.ARFO UDL.ARFN UDL.ARFR* UDL.ARFX*	No	TSW.AFM
Data Communications • SVC 15 • 3270 • ZDLC	UDL.TSKO UDL.TSKN UDL.TSKQ	Yes	TSW.TSKM TSW.ITM
Data Format/ Alignment Fault	UDL.DFFO* UDL.DFFN UDL.DFFX* UDL.DFFR	No	TSW.DFFM
Device Interrupt (SVC 6) • Connect • Thaw • Sint • Freeze • Unconnect	UDL.TSKO UDL.TSKN UDL.TSKQ	Yes	TSW.TSKM TSW.DIQM
Illegal Instruction Fault	UDL.IITO UDL.IITN	No	TSW.IITM
I/O Proceed Completion (SVC 1)	UDL.TSKO UDL.TSKN UDL.TSKQ	Yes	TSW.TSKM TSW.IOM
Load and Proceed Completion (SVC 1)	UDL.TSKO UDL.TSKN UDL.TSKQ	Yes	TSW.TSKM TSW.LODM
Memory Access	UDL.MAFO UDL.MAFN UDL.MAFL* UDL.MAFR* UDL.LMSA	No	TSW.MAFM
Power Restoration	UDL.PWRO UDL.PWRN	No	TSW.PWRM

TABLE 3-8 SUMMARY OF TASK STRUCTURES USED FOR HANDLING TRAPS (Continued)

TRAP-CAUSING EVENT	UDL FIELDS	TASK QUEUE	TSW BIT MASK
Send Data*** (SVC 6)	UDL.TSKO UDL.TSKN UDL.TSKQ UDL.SDQ	Yes	TSW.TSKM TSW.SDM
Send Message*** (SVC 6)	UDL.TSKO UDL.TSKN UDL.TSKQ UDL.MSGR	Yes	TSW.TSKM TSW.PMM
Send Queue Parameter (SVC 6)	UDL.TSKO UDL.TSKN UDL.TSKQ	Yes	TSW.TSKM TSW.TCM
Subtask State Change (SVC 6)	UDL.TSKO UDL.TSKN UDL.TSKQ	Yes	TSW.TSKM TSW.SUQM
SVC 14	UDL.S14O UDL.S14N UDL.SV14	No	TSW.S14M
Timer Termination (SVC 2, Code 23)	UDL.TSKO UDL.TSKN UDL.TSKQ	Yes	TSW.TSKM TSM.TMCM
Trap Wait	-	No	TSM.WTM

* Available on Perkin-Elmer Series 3200 processors only.

** Task must be linked-edited with NAFPAUSE task option enabled.

*** Task must also include a message buffer to receive the message. See OS/32 Supervisor Call Reference Manual.

**** Available with Model 3200MPS System only.

NOTE

A task can be suspended until a trap-causing event occurs by setting the TSW.WTM bit in the OTSW.

3.6.1 Handling Task Queue Traps

In addition to the UDL and TSW, a program that handles task queue traps must have a task queue. The DLIST instruction can be used to build a circular list for the task queue as follows:

```
QUEUE    DLIST 3
```

The following example builds a UDL structure for a task that handles I/O proceed completion traps. Note that the address of the task queue is placed in the UDL.TSKQ field.

Example:

```
         $UDL
         ABS    0
MYUDL    DS     UDL
MYUDLE   EQU    *
         ORG    MYUDL+UDL.TSKQ
         DC     A(QUEUE)             INITS ADDRESS OF TASK QUEUE
         ORG    MYUDL+UDL.TSKN
         DC     0
         DC     A(TRAP)              INITS ADDR OF TASK QUEUE HANDLER
         ORG    MYUDLE
```

The TSW for a task handling an input/output (I/O) proceed completion trap is initialized as follows:

```
         $TSW
NTSW     DC     TSW.TSKM!TSW.IOM    SETS TASK QUEUE & I/O PROCEED BITS
         DC     0
```

If a task queue trap is a send data trap, the task requires a message buffer queue to receive the message sent by the trap. The address of this queue is placed in the UDL.SDQ field.

Send message traps require a message ring whose address is placed in the UDL.MSGR field. For more information on how to set up a message ring or send data buffer queue, see the OS/32 Supervisor Call (SVC) Reference Manual.

The following code sets up a UDL, TSW, task queue and message ring for a task to service a send message trap.

Example:

```
          $UDL
          $TSW
          ABS    0
MYUDL     DS     UDL
MYUDLE    EQU    *
          ORG    MYUDL+UDL.TSKQ
          DC     A(QUEUE)              STORES TASK QUEUE ADDR
          ORG    MYUDL+UDL.MSGR
          DC     A(MESS1)              STORES MESSAGE RING ADDR
          ORG    MYUDL+UDL.TSKN
          DC     0
          DC     A(TRAP)               STORES TRAP HANDLER ADDR IN LOC
          ORG    MYUDLE
QUEUE     DLIST 3
TSW       DC     TSW.TSKM!TSW.PMM      SETS TSK Q AND SEND MESS TRAP BITS
          DC     0
          ALIGN 4
MESS1     DC     A(MESS2)
          DS     72
MESS2     DC     A(MESS1)
          DS     72
               .
               .
               .
```

3.6.2 Tips for Writing Task Trap Handling Routines

The task trap handling routine should contain all the program code necessary to process the trap. Because no registers are saved as part of the TSW swap that causes a trap handling routine to be initiated, the routine should save the contents of any registers required by the task.

Task queue trap handling routines should contain code that will remove items from the task queue. For example, suppose a send message trap placed the following item in slot five of the task queue:

```
06 A(MESS1)
```

The following example demonstrates one method of removing this item from the task queue.

Example:

```
TRAP     EQU    *
         STM    R0,TRAPSAVE           SAVE REGS (OPTIONAL)
         RTL    1,QUEUE               TRANSFER QUEUE ITEM TO R1
         BO     TRAPEXIT              QUEUE EMPTY
         RLL    1,8
         LBR    2,1                   VERIFY REASON CODE
         CLI    2,6
         BNE    ERROR
         SRL    1,8                   SHIFT R1 ONE BYTE TO LEFT
         LI     2,12(1)               (R1)+12 IS MESS START ADR
         ST     2,WRITE+SVC.1.SAD     STORE ADR IN SVC 1 PARBLK
         LI     2,63(2)               (R2)+63 IS MESS END ADR
         ST     2,WRITE+SVC1.EAD      STORE ADR IN SVC 1 PARBLK
                  .
                  .
                  .
         B      TRAP                  SEE IF ANY MORE ON QUEUE
TRAPEXIT EQU    *
         LM     R0,TRAPSAVE           RESTORE REGS (OPTIONAL)
         SVC    9,UDL.TSKO
```

To resume task execution, a trap handling routine must return the old TSW in the UDL to the TCB. In the above example, the routine issued an SVC 9. See the OS/32 Supervisor Call (SVC) Reference Manual for more information.

3.6.3 Handling Traps From Trap-Generating Devices

OS/32 provides intertask control services that allow a task to receive a trap from an external trap-generating device. These services include:

Connect	attaches a trap-generating device to a task
Thaw	enables interrupts from the attached trap-generating device
Sint	simulates an interrupt from a trap-generating device
Freeze	disables interrupts from the attached trap-generating device
Unconnect	detaches a trap-generating device from a task

These services implement the proposed standards established by the Instrument Society of America (ISA) for electronic devices.

An example of a task that receives and handles traps from trap-generating devices is the Perkin-Elmer 8-line interrupt module driver. To handle a trap received from a trap-generating device, the task sets the TSW.TSKM and TSW.DIQM bits in the TSW. The task also builds a task queue to receive an entry from the device when an interrupt occurs. Using the OS/32 intertask control services and data structures for handling task queue traps, the user can write a task that handles traps from external devices.

The OS/32 intertask control services can also be used to attach an APU to a task and enable interrupts from the APU each time it sends a signal to the CPU in a Model 3200MPS System. To handle a trap generated by an APU signal, the task must have the TSW.APTM and TSW.TSKM bits set in the TSW. The task must also build a task queue to receive the interrupt information items from the APU. See Figure 3-4.

See the OS/32 Supervisor Call (SVC) Reference Manual for more information on the intertask control services provided by OS/32.

3.6.4 Sample Task Trap Handling Program

The following assembly program suspends execution of a task (called the directed task) until another task (called the calling task) places a reason code and message buffer address on the task queue. After the send message trap occurs, task execution branches to the task queue trap handling routine that removes the queue entry and stores the beginning and ending address of the message in an SVC 1 parameter block. The routine then outputs the message sent from the calling task. See the OS/32 Supervisor Call (SVC) Reference Manual for more information on SVC 1.

Example:

```
 1           PROG   DIRECTED TASK WITH TRAP HANDLER
 2           MLIBS  8,9,10
 3           NLSTM
 4           FREZE
 5           $UDL                      DEFINE UDL STRUC
 6           $TSW                      DEFINE TSW STRUC
 7           ABS    0
 8  MYUDL    DS     UDL                RESERVE STORAGE FOR UDL
 9  MYUDLE   EQU    *
10           ORG    MYUDL+UDL.TSKQ     INITIALIZE UDL FIELDS
11           DC     A(QUEUE)           DEFINE TASK QUEUE LOC
12           ORG    MYUDL+UDL.MSGR
13           DC     A(MESS1)           DEFINE MESSAGE RING LOC
14           ORG    MYUDL+UDL.TSKN
15           DC     TSW.PMM
16           DC     A(QTRAP)           DEFINE NEW TSW FOR TASK TRAP HANDLER
17           ORG    MYUDLE
18           IMPUR
19  QUEUE    DLIST  3
20  *                                  ENABLE TRAP WAIT,TASK Q TRAPS,MESS Q
21  TSW      DC     TSW.WTM!TSW.TSKM!TSW.PMM
22           DC     0                  RESUME AFTER SVC 9
23           ALIGN  4
24  MESS1    DC     A(MESS2)           RESERVE STORAGE FOR MESSAGE RING
25           DS     72
26  MESS2    DC     A(MESS1)
27           DS     72
28           $SVC1
29           ALIGN  4
30  WRITE    DS     SVC1.              SVC 1 PARAMETER BLOCK
31  ENDBLK   EQU    *
32           ORG    WRITE+SVC1.FUN
33           DB     SV1.WRIT!SV1.WAIT  I/O WRITE AND WAIT FUNC
34           DB     2                  LOGICAL UNIT 2
35           ORG    ENDBLK
36  START    EQU    *
37           SVC    9,TSW              ENABLE Q ENTRIES,TRAP WAIT,ENTER TRAP WAIT
38           SVC    3,0                END TASK
39  QTRAP    EQU    *
40           RTL    1,QUEUE            REMOVE QUEUE ENTRY TO R1
41           RLL    1,8                ROTATE REASON CODE TO LOW BYTE
42           LBR    2,1                MOVE REASON CODE TO R2
43           CLI    2,6                IS IT A MESSAGE (RC=6)
44           BNE    ERROR              NO, ABORT ON ERROR
45           SRL    1,8                SHIFT A(MESS) BACK IN R1
46           LA     2,12(1)            R2 = A(MESSAGE START)
47           ST     2,WRITE+SVC1.SAD   PUT ADR IN SVC 1 PARBLK
48           LA     2,63(2)            R2 = A(MESSAGE END)
49           ST     2,WRITE+SVC1.EAD   PUT ADR IN SVC 1 PARBLK
50           SVC    1,WRITE            ISSUE WRITE
51           LIS    2,0                BIT OFFSET = 0
52           RBT    2,0(1)             RELEASE MESSAGE BUFFER
53           RBT    2,UDL.TSKO         RESET TRAP WAIT IN OLD TSW
54           SVC    9,UDL.TSKO         LOAD OLD TSW
55  ERROR    SVC    3,4                ABORT:  RETURN CODE = 4
56           END    START              SPECIFIES PROGRAM START ADDRESS
```

3.6.5 Using the OS/32 System Macro Library to Handle Traps

The OS/32 system macro library provides macro definitions for setting up the data structures necessary to handle task traps. Another macro, LTSW, performs a TSW load. The following program performs the same functions as the sample program given in Section 3.6.4. Notice, however, the lines of code that have been replaced with one-line macros.

Example:

```
                                                  Lines Replaced by Macros
         MLIBS 8,9,10
         NLSTM
 SETUDL TSKQ=QUEUE,MSGR=MESS1,TSKN=(O,TRAP)                5-17
 LTSW WT,TSKE,TMQ                                          21-22,37
 EOT RC=O                                                   38
QUEUE    DLIST  3
MESS1 MSGRING  3,72                                        24-27
TRAP     EQU    *
         RTL    1,QUEUE
         RLL    1,8
         LBR    2,1
         CLI    2,6
         BNE    ERROR
         SRL    1,8
 WRITE LU=2,ADDR=12(1),RECL=64                             28-35,50
         LIS    2,0
         RBT    2,0(1)
 LTSW PCB=UDL.TSKO                                          54
ERROR    SVC    3,4
         END
```

See the OS/32 System Macro Library Reference Manual for details on how to use the OS/32 macro instructions for writing trap handling programs.

3.6.6 Writing FORTRAN Trap Handling Programs

The Perkin-Elmer FORTRAN VII Run-time Library (RTL) provides subroutines that allow the FORTRAN programmer to write programs that handle task traps. Subroutine INIT initializes the task's TSW, UDL, task queue, and message ring. Subroutine ENABLE sets the appropriate TSW trap bit and stores the address of the task trap handling routine in the UDL.

Example:

```
C THIS PROGRAM SERVICES DEVICE INTERRUPTS
C
      EXTERNAL NAME
           .
           .
           .
      CALL INIT
           .
           .
           .
      CALL ENABLE (1,NAME)
      END


C  THE FOLLOWING SUBROUTINE
C  HANDLES THE TASK TRAP
C
      SUBROUTINE NAME (...)
           .
           .
           .
      RETURN
      END
```

See the FORTRAN VII User Guide for more information on writing FORTRAN programs that handle task traps.

3.6.7 Writing Pascal Trap Handling Programs

Writing a Pascal program that enables and handles task traps is very difficult, but it can be done by using a combination of Pascal features and common assembly language (CAL) routines.

The SMPLSVCS.PAS file supplied with the Perkin-Elmer Pascal compiler provides constants and types for handling SVCs in a Pascal program. The following code represents a sample Pascal program for enabling memory access faults.

Example:

```
PROGRAM SAMPLE;

CONST MAFN = 35;
      TSW_ MEMF_EN = #04000000;

TYPE  UDL_INDEX = 0..63;

VAR   NEWTSW,HADDR: INTEGER;

PROCEDURE TOUDL (I: ULD_INDEX;
                 VAL: UNIV INTEGER); EXTERN;
PROCEDURE GETHADDR (VAR HADDR: INTEGER); EXTERN;
PROCEDURE SETTSW (NEWTSW: INTEGER); EXTERN;
PROCEDURE SVC2PAUS; EXTERN;

BEGIN
   GETHADDR (HADDR);
   TOUDL (MAFN,HADDR);
   NEWTSW := TSW_MEMF_EN;
   SETTSW (NEWTSW);
   SVC2PAUS;
END.
```

```
GETHADDR PROG  GET ADDRESS OF HANDLER
         ENTRY GETHADDR
         EXTRN MEMAFH
STACK    STRUC                   STRUC OF ACTIVATION REC ON STACK
OLDLB    DSF   1                 OLD LOCAL BASE (R2)
RETAD    DSF   1                 RETURN ADDRESS
SLINK    DSF   1                 STATIC LINK
         ENDS
GETHADDR EQU   *
         ST    15,RETAD(2)       SAVE RETURN ADR ON STACK
         LA    8,MEMAFH          MOVE ADR OF HANDLER
         ST    8,0(3)            TO ARGUMENT (R3 = A(HADDR))
         L     15,RETAD(2)       RESTORE RETURN ADDRESS
         L     2,OLDLB(2)        RELOAD LOCAL BASE (RELEASE STACK)
         BR    15                RETURN
         END
```

```
SETTSW    PROG  SET UP NEW TSW IN  TCB
          ENTRY SETTSW
STACK     STRUC
OLDLB     DSF   1
RETAD     DSF   1
SLINK     DSF   1
SVC9BLK   DSF   2
          ENDS
SETTSW    EQU   *
          ST    15,RETAD(2)          SAVE RETURN ADDRESS
          ST    3,SVC9BLK(2)         STORE NEW TSW ON STACK
          LIS   R0,0
          ST    R0,SVC9BLK+4(2)      ZERO LOC OF NEW TSW
          SVC   9,SVC9BLK(2)         LOAD NEW TSW
          L     15,RETAD(2)          RESTORE RETURN ADDRESS
          L     2,OLDLB(2)           RELEASE STACK
          BR    15                   RETURN
          END
MODULE    MEMAFH
BEGIN
          .
          .
          .
END
```

The above example enables and handles memory access faults. The user-written SETTSW procedure issues an SVC 9 to replace the link-initialized TSW in the TCB with a TSW enabled for memory access faults. The user-written procedure GETHADDR sets the variable HADDR to the address of the task trap handling routine (MEMAFH). The procedure TOUDL places the address of this routine into UDL.MAFN. TOUDL is included in the SMPLSVCS.PAS file.

Except for arithmetic fault handlers, all trap-handling programs require a similar user-written procedure to enable the appropriate trap bit. SMPLSVCS.PAS provides information about a procedure that automatically enables the arithmetic fault trap bit in the TCB.

If the trap-handling routines are written in Pascal, the Pascal register set must be set up or preserved. Entry into the trap handling routine would then usually be through a CAL routine that establishes the register set. A separate stack/heap area may need to be set up in such a CAL routine.

For more information on traps and interfaces between Pascal and CAL routines, see the Perkin-Elmer Pascal User Guide, Language Reference, and Run-Time Support Reference Manual.

CHAPTER 4
OS/32 DISK FILE MANAGEMENT SERVICES

4.1 INTRODUCTION TO THE OS/32 FILE MANAGER

Application programs read and write data through the peripheral devices connected to the computer. In addition to such input/output (I/O) operations as logging messages on the system console or reading data from a multi-terminal monitor (MTM) terminal, a task should be able to store any amount of data for future use. All tasks within a system should be able to store, move, and update all information required by the user's application.

The OS/32 file manager stores and retrieves information for a task on secondary storage devices (disks, magnetic tapes, floppy disks, etc.). The file manager partitions this storage into smaller areas, called files, that can be used by tasks for data and program storage. In addition, the file manager provides tasks with the following support services for management of files on disk:

Allocate	initializes a file by allocating space on disk
Delete	removes a file from disk
Rename	changes the name of a file
Open	assigns a file to a task
Close	releases a file (cancel an existing assignment) when a task has completed its use of the file
Fetch Attributes	examines the attributes of a file
Checkpoint	ensures that all data in an output buffer is written to disk

This chapter describes the OS/32 structures that are used by the file manager to provide these services.

4.2 SYSTEM RESOURCE MANAGEMENT

The file manager controls two types of system resources for a task; namely, files and the devices that store and retrieve files.

A file is a named collection of data records on a secondary storage device. Secondary storage devices supported by the file manager include fixed-head disks, moving-head disks, and floppy disks. Fixed-head disk drives have smaller average access times than moving-head disks, while moving-head disks typically have larger storage capacities.

Devices are read from or written to like ordinary disk files; i.e., a task performs a data transfer to a device in the same manner it would perform a data transfer to a file on disk. However, when a task performs an I/O operation to a device (such as a printer, data communication device, magnetic tape, etc.), data must be transferred via an appropriate protocol determined by the device driver. Device drivers are system software modules that make the device look like an ordinary disk file to the file manager. These drivers are included in the OS/32 I/O subsystem.

Every file and device on a Perkin-Elmer system is referenced by a file descriptor (fd). The fd is used by the file manager to find and access a device or file as required by the task.

Format

```
[ { voln: } ] [filename] [.ext] [ / { actno      } ]
[ { dev:  } ]                   [   { file class } ]
```

Parameters:

voln: is a 1- to 4-character alphanumeric string specifying the name of the disk volume on which the file specified by filename resides. The first character must be alphabetic and the remaining alphanumeric. If this parameter is omitted, the default is the system volume in an OS/32 real-time environment or a default volume specified by the user in an MTM environment.

dev: is a 1- to 4-character alphanumeric string specifying the name of a device (e.g., CON:, PR:, NULL:, or MAG1:). The first character must be alphabetic and the remaining alphanumeric.

filename	is a 1- to 8-character alphanumeric string specifying the name of a disk file. The first character must be alphabetic and the remaining alphanumeric. If a filename is specified when a device name is referenced, the filename is ignored.
.ext	is a 1- to 3-character alphanumeric string specifying the extension to a filename.
actno	is a decimal number ranging from 0 through 65,535, specifying the account number associated with the file. Account numbers 1 through 65,535 (excluding 255) are used by MTM for terminal users. Acount number 255 is reserved for the MTM system administrator. Account number 0 is for system files and is the default for all operator commands.
file class	is a 1-character alphabetic string specifying the file class. The file classes are: • P for a private file • G for a group file • S for a system file If the file class is omitted, the default is P for files generated in an MTM environment and S for files generated in an OS/32 real-time environment.

4.3 FILE ORGANIZATION

A data record is a list of information elements that are accessed together. Before the file manager can store a record on a disk, the disk must be initialized by the OS/32 disk initializer program. See the OS/32 Fastchek Reference Manual for more information. Figure 4-1 shows the surface of a formatted disk. Note that the surface of the disk is divided into tracks. A track is the area covered by a stationary read/write head with one revolution of the disk. The amount of information that can be stored on a track is a function of the recording density and size of the track.

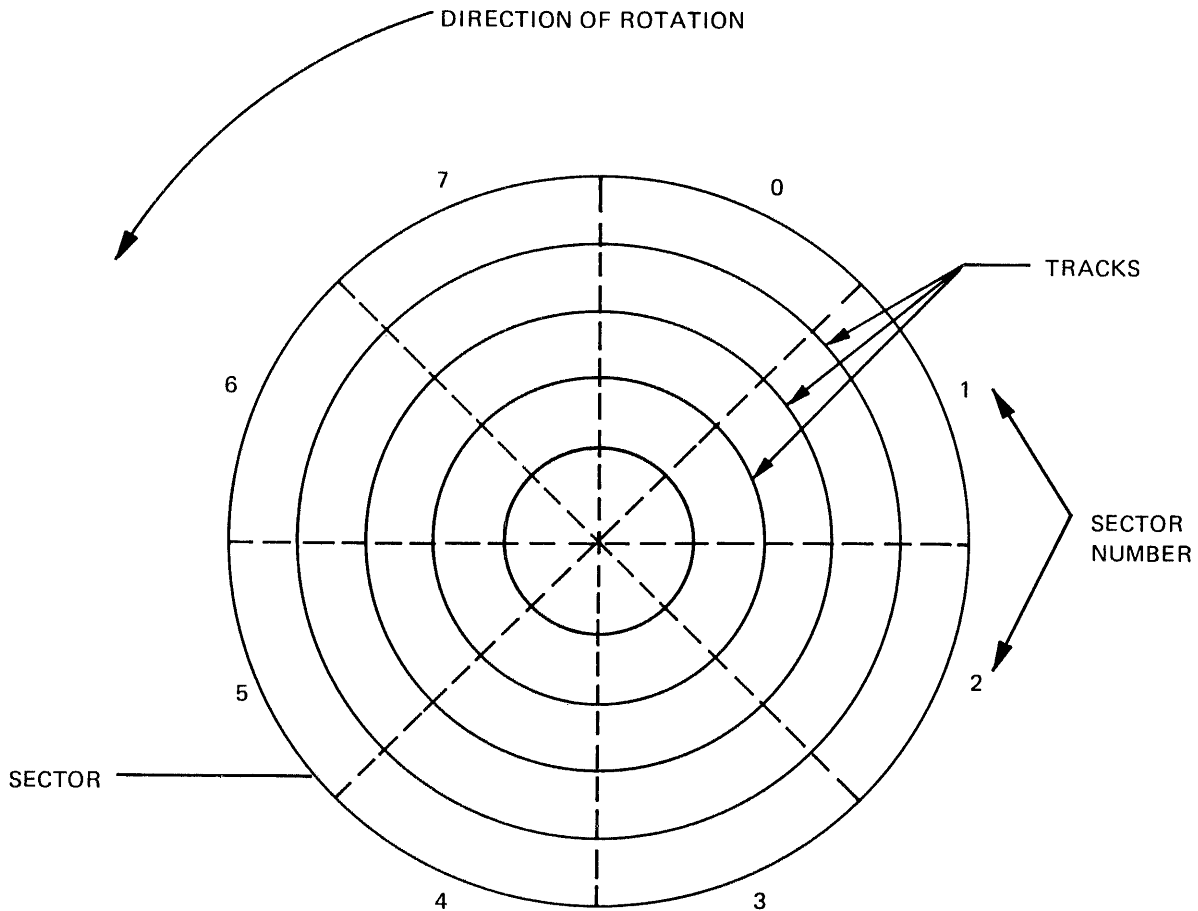


Figure 4-1 Formatted Disk Surface

When a disk is initialized, the tracks are divided into sectors. The disk surface shown in Figure 4-1 is divided into 8 sectors. Each sector of each track holds 256 bytes of data, the smallest addressable storage area on disk.

The file manager uses two methods for organizing data records into files on formatted disks:

- linked-list indexed organization, and
- contiguous organization.

4.3.1 Linked-List Indexed Organization

When the file manager creates a file using the linked-list indexed organization method, it sets aside two types of storage areas or blocks on disk: data blocks and index blocks. These areas are shown in Figure 4-2. The data block is a group of one or more contiguous sectors that are used to store the data records. The index block is a group of contiguous sectors that store pointers to the individual data blocks. Note that index block 1 in Figure 4-2 points to the first sector of index block 2. Because each index block points to its successor, the file is said to contain a linked-list indexed organization. One index block for each indexed file assigned to a task remains in dynamic system space as long as the file is assigned to the task. The remaining index blocks remain on disk. For buffered indexed files, space for two data blocks is reserved in dynamic system space as long as the file remains assigned.

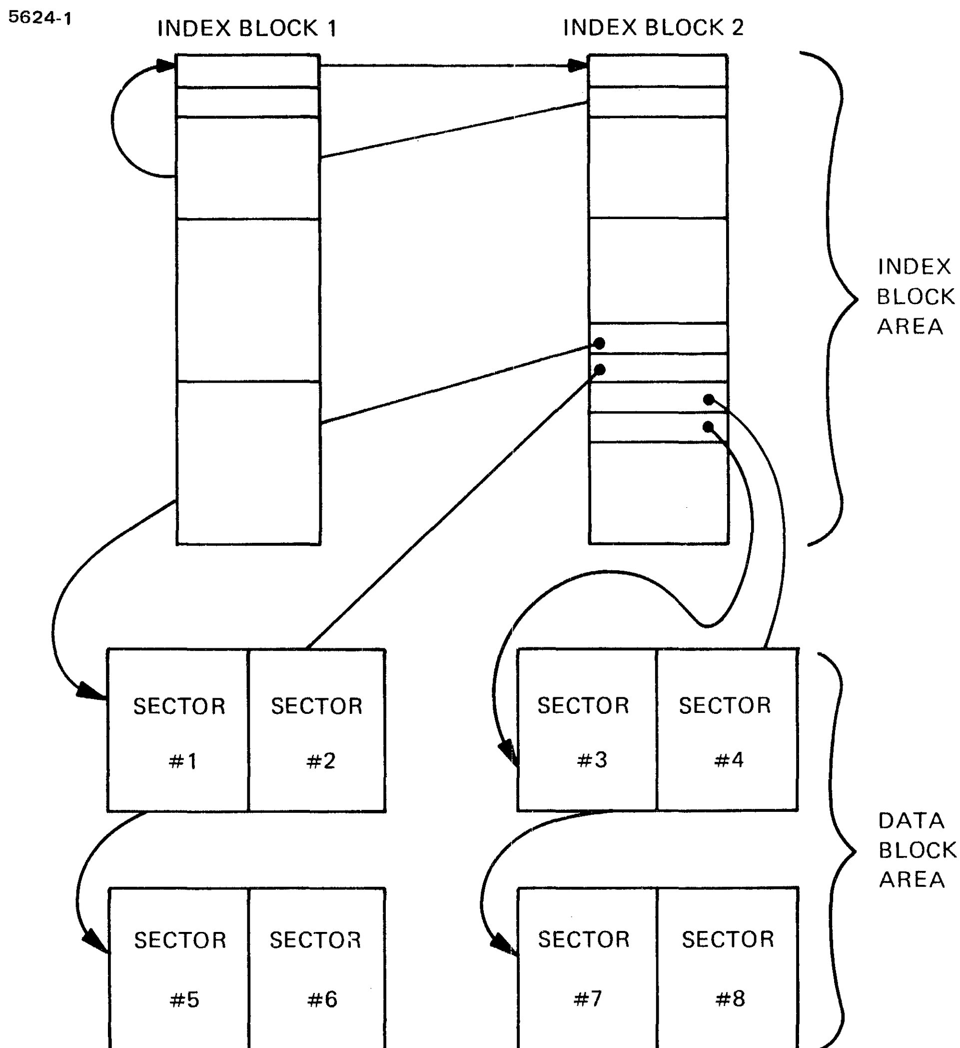


Figure 4-2 Linked-List Indexed File Organization

4.3.2 Contiguous Organization

A contiguously organized disk file consists of a sequence of data blocks stored on tracks with consecutive addresses. Each block of data is one sector (256 bytes) in length. Contiguous files can be accessed randomly (by sector) or sequentially. When accessed sequentially, the contiguous file appears to have a magnetic tape like organization; i.e., to the task the file resembles a magnetic tape with 256-byte blocks. For example, the task can write a filemark (X'1313') to the first 2 bytes of a sector in a contiguous disk file. Using the filemark as a record delimiter, the task can space forward or backward to the filemark as it would to a filemark on magnetic tape.

4.4 SUPPORTED DISK FILE TYPES

How a file is organized determines the size of its data records. Contiguous files have a fixed record length and file length. Indexed organization supports fixed record lengths but the file is extendable. Files can extend the length of the disk volume.

To meet the requirements of applications running in a real-time environment, the OS/32 file manager supports four different types of disk files. These file types offer the user a flexible range of record lengths, file lengths, and access methods. The file types are:

- contiguous
- indexed
- nonbuffered indexed
- extendable contiguous

The following sections define these file types in terms of their record and file lengths. Access methods are discussed in Section 4.7.

4.4.1 Contiguous Files

Contiguous files are organized sequentially on disk. When the file manager has allocated a contiguous file (i.e., reserved a user-specified number of contiguous sectors on disk), the maximum length of the file is fixed. It cannot be changed during data transfer. Records within a contiguous file are 256 bytes (one sector) in length. A read or write to a contiguous file can transfer any amount of data (from a partial sector to the entire length of the file) in a single I/O operation. Nevertheless, the I/O operation must begin on a sector boundary.

Like records stored on magnetic tape, contiguous file records are stored on consecutive adjacent sectors. In addition, a filemark can be written to the first 2 bytes of each record. When using the filemark capability, care should be taken that any data to be transferred to disk does not contain X'1313'.

4.4.2 Indexed and Nonbuffered Indexed Files

Two types of files for which the file manager uses a linked-list organization are the buffered indexed and nonbuffered indexed files. Because these files are open-ended, the maximum length of either of these files is determined during data transfer. Maximum file length is limited only by the free space available on disk; however, records are restricted in size from 1 to 65,535 bytes. In addition, because of hardware restrictions, nonbuffered indexed file records must be an even number of bytes in length. The record size is specified by the user when the file is allocated. Records are stored in data blocks consisting of one or more 256-byte contiguous sectors.

The organization of records in a nonbuffered indexed file differs from that for indexed file records. Each record in a nonbuffered indexed file begins on a physical sector (256-byte) boundary, whether or not the previously transferred record filled up its sector space. Also, nonbuffered files do not have memory reserved in dynamic system space for data block buffers. Indexed file records are transferred so no unused space remains between two records. There are no hardware restrictions for indexed files.

4.4.3 Extendable Contiguous Files

The third type of file that uses a linked-list organization is the extendable contiguous file. Like the indexed and nonbuffered indexed files, the maximum length of an extendable contiguous file can be extended by write operations. However, like contiguous files, the record length is fixed at 256 bytes (1 sector), and any number of sectors can be transferred in a single I/O operation.

4.5 DISK SPACE MANAGEMENT

In servicing task file requests, the file manager must:

- allocate file directory entries and index blocks,
- allocate a sufficient amount of disk space to contain the file (e.g., contiguous files),
- extend the space already allocated to a file (indexed, nonbuffered indexed, or extendable contiguous) by allocating additional index blocks and data blocks, and
- delete a file and return the space to the available space inventory.

To perform these operations effectively, the file manager must keep track of all the files on a disk and the storage space available to the files. A special data structure, called the volume descriptor, is used by the file manager to identify the disk volume and to point to structures that contain the file and disk space information.

The volume descriptor is shown in Figure 4-3. When formatting the disk, the OS/32 disk initializer initializes the volume descriptor in logical block address (LBA) zero (sector 0, head 0, cylinder 0). The data stored in this data structure includes:

- the name of the disk volume,
- the address of the bit map, and
- the address of the first block in the primary file directory.

5625

Offset	Field
0 (0)	VOL (4) VOLUME NAME
4 (4)	ATRB (4) VOLUME ATTRIBUTES
8 (8)	FDP (4) FIRST DIRECTORY BLOCK POINTER
12 (C)	OSP (4) POINTER TO OS IMAGE (UNUSED)
16 (10)	OSS (4) SIZE OF OS IMAGE (UNUSED)
20 (14)	MAP (4) POINTER TO BIT MAP

Figure 4-3 Volume Descriptor

4.5.1 File Directories

The file manager keeps track of all the files on the disk through the primary file directory. This file is organized as a linked-list of one-sector directory blocks. Figure 4-4 shows one block of a primary directory. Note that it contains five directory entries and a pointer to the next primary directory block.

5626

Offset (dec)	Offset (hex)	Field	Length
0	(0)	NEXT DIRECTORY BLOCK POINTER	4
4	(4)	DIRECTORY ENTRY 1	48
52	(34)	DIRECTORY ENTRY 2	48
100	(64)	DIRECTORY ENTRY 3	48
148	(94)	DIRECTORY ENTRY 4	48
196	(C4)	DIRECTORY ENTRY 5	48
244	(F4)	RESERVED	12
256	(100)		

Figure 4-4 Primary Directory Block

5627

Offset	(Hex)	Field		Offset	(Hex)	Field		Offset	(Hex)	Field		Offset	(Hex)	Field	
0	(0)	FNM 8 FILENAME													
8	(8)	EXT 3 EXTENSION										11	(B)	ACT FILE ACCOUNT #	1
12	(C)	FLBA 4 FIRST LOGICAL BLOCK ADDR													
16	(10)	LLBA 4 LAST LOGICAL BLOCK ADDR													
20	(14)	KEYS WKEY	0 1	21	(15)	RKEY READ KEY	1	22	(16)	LRCL LOGICAL RECORD LENGTH	2				
24	(18)	DATE 4 DATE FILE ALLOCATED													
28	(1C)	LUSE 4 LAST DATE FILE ASSIGNED													
32	(20)	WCNT 2 WRITE COUNT						34	(22)	RCNT 2 READ COUNT					
36	(24)	ATTRIBUTES		37	(25)	BKSZ 1 BLOCK SIZE		38	(26)	INBS 1 INDEX BLOCK SIZE		39	(27)	RESERVED	1
40	(28)	CSEC 4 CURRENT SECTOR/#LOGICAL RECORDS													
44	(2C)	RESERVED													
48	(30)														

Figure 4-5 Primary Directory Entry

Each directory entry is a 48-byte data structure that identifies and describes a file on the disk. Every file has an entry in the primary directory. As shown in Figure 4-5, the directory entry tells the file manager:

- the name and extension of the file,
- the low order byte of the file's account number (The eight bits of the high order byte of the account number are distributed across the high-order bits of the eight characters of the filename.),
- the addresses of the first and last sectors if the allocated file is a contiguous file, or the addresses of the first and last index blocks if the allocated file is a nonbuffered indexed, extendable contiguous, or indexed file,

- the file's access privileges,
- the length of the file's records,
- file attributes or set of operations that can be performed on the file,
- date the file was allocated,
- date the file was last read or written by a task,
- the data block size and index block size if the file is an indexed, nonbuffered indexed, or extendable contiguous file, and
- the number of disk records or sectors currently used by the file.

Only one directory block can remain in memory at a time; therefore, only five file entries can be memory resident. The file manager scans these entries to find a file. To search the remaining files on the system, the file manager has to replace the primary directory block in memory with the next block from the disk. Examining five entries at a time to find a file can greatly increase the time it takes to access that file.

To decrease the amount of time required to scan the directory, the file manager uses a secondary file directory. (See Figure 4-6.) The secondary directory is a contiguous file named SYSTEM.DIR that is created when the disk is marked online. The secondary directory points to each block of the primary directory and lists the filenames of the directory entries that are in each block. All or part of the secondary directory is maintained in memory in dynamic system space. After the file manager finds the filename it is searching for in the secondary directory, it can directly access the primary directory block that contains its directory entry. Note that while this method saves access time, the secondary directory does use more memory space than scanning the primary directory.

To use the secondary file directory method, specify this option in the MARKON command when marking a disk online. See the OS/32 Operator's Reference Manual. This command also allows the operator to specify the size of SYSTEM.DIR, including room for expansion. If during processing, the secondary directory cannot accommodate any additional files that are being allocated, the disk must be marked offline and then marked back online to recreate a larger secondary directory. The MARKON command will provide the operator with information to make a decision as to the preferred size of the secondary directory.

5628

Offset	Field	Size
0(0)	PRIMARY DIRECTORY BLOCK POINTER 1	[4]
4(4)	FILENAME 1	[12]
16(10)	FILENAME 2	[12]
28(1C)	FILENAME 3	[12]
40(28)	FILENAME 4	[12]
52(34)	FILENAME 5	[12]
	|	
192(C0)	PRIMARY DIRECTORY BLOCK POINTER 4	[4]
196(C4)	FILENAME 16	[12]
208(D0)	FILENAME 17	[12]
220(DC)	FILENAME 18	[12]
232(E8)	FILENAME 19	[12]
244(F4)	FILENAME 20	[12]
256(100)		

Figure 4-6 Secondary File Directory (SYSTEM.DIR)

4.5.2 Bit Map

The file manager interrogates the bit map to determine which sectors are available for allocation. The bit map maintains an available space inventory for the disk volume. A bit in the bit map is assigned to each sector on disk. When a bit in the bit map is set to 1, its corresponding sector is allocated for a file. When the bit is set to 0, its corresponding sector is free. The disk initializer places the bit map close to the file directory to minimize disk head movement when allocating files.

4.5.3 Permanent and Temporary File Allocation

When a file is allocated, it can be designated as a permanent or temporary file. If permanent, the file remains on disk until an operator command or a task asks the file manager to delete it. Temporary files remain on disk only as long as they are assigned to a task. Once the assignment to a task is closed, the file manager deletes the file.

4.6 ASSIGNING FILES TO A TASK

The OS/32 file manager allows a task to access system resources via a logical unit (lu) number rather than by the name of a device or file. Up to 255 logical units (0 through 254) can be used by a task. Because lu255 is reserved, it is not available for task use.

All disk files that are to be accessed by a task must be assigned to an lu before any I/O operations can be performed to those files. Once a file is assigned to an lu, the OS/32 I/O subsystem ensures that the proper device driver or controller is used when the task requests an I/O transfer to the lu. Such I/O requests are device-independent I/O; i.e., device assignments are made by the operator who started the task, not by the task that made the I/O request.

For example, suppose a FORTRAN program has the following code:

```
READ(2,100) A
WRITE(1,100) B
```

When executed, the task reads a value from the device or file assigned to lu2, stores it into variable A, and writes the value of B to the device assigned to lu1. The operator may assign lu2 to an MTM terminal, disk file or whatever input device is available to the task. Likewise, lu1 can be assigned to a disk file, printer, terminal, or whatever output device is available. Hence, device-independent I/O allows the devices or files that will be used by a task to be changed without changing the actual code within the program.

Sometimes a programmer may wish to perform an operation while suppressing the output from that operation. For example, one may wish to compile a program or build a task image without creating an object or task image file. To do this, the lu should be assigned to the NULL: device. This assignment allows the operation to be performed without generating any output from that operation.

4.7 ACCESS METHODS

The OS/32 system services that interpret and fulfill a task's request for storage and retrieval of data are known as access methods. The OS/32 I/O subsystem supports two access methods: buffered and nonbuffered. Both methods are transparent to the user.

To perform a read or write operation, a task should have two interfaces to these access methods:

- the user code interface that requests a data transfer (e.g., a READ or WRITE statement in FORTRAN), and
- the lu assigned to the file required by the task.

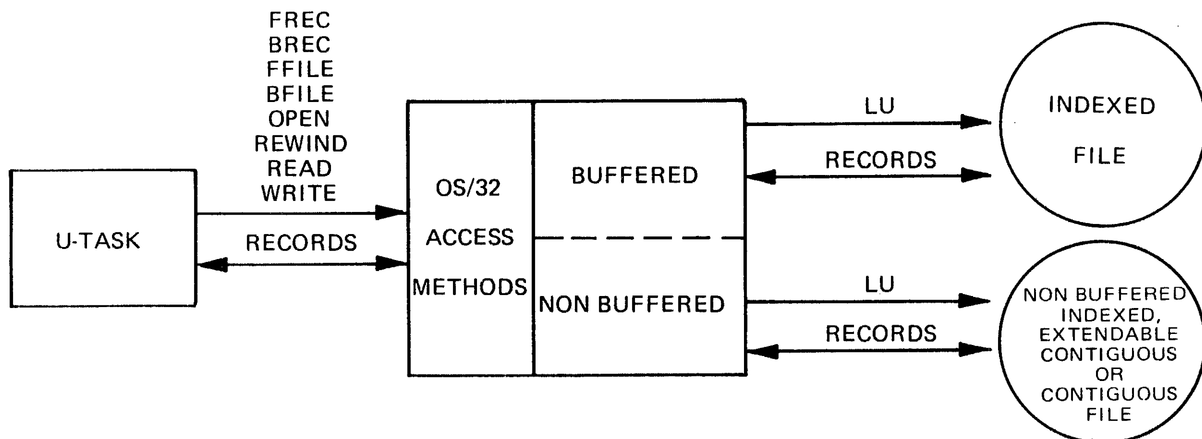


Figure 4-7 Task Interfaces to Access Methods

As shown in Figure 4-7, the access methods fall between these two interfaces.

Each time a read or write operation is performed to a file, the access methods adjust the current record pointer for that file. The value of the current record pointer is the number of a logical record in a file on disk. For contiguous and extendable contiguous files, this number refers to a logical sector address. For nonbuffered indexed and indexed files, this number refers to a logical record. The value of the current record pointer can range from 0 to the current number of records in the file minus one.

All records can be accessed sequentially or randomly. When data records are transferred sequentially (i.e., one record at a time) the record pointer is automatically incremented by 1 to point to the next record after the last record or sector is transferred. After a random read or write operation is completed, the record pointer is set to the number of the record immediately following the last one that was transferred.

In addition to read or write operations, the record pointer is adjusted after the following operations are requested by the task:

- Rewind

  Record pointer is set to 0

- Assign (open)

  Record pointer is set to 0 for all access privileges except write only (SWO/EWO). If write only is in effect, the record pointer is set to the number of the record following the last existing record in an indexed or nonbuffered indexed file and to the last record read or written in a contiguous or extendable contiguous file.

- Backspace Filemark (BFILE)

  If the file is a contiguous file, the record pointer backspaces to the number of the record containing a filemark. Otherwise, the record pointer is set to zero.

- Forward Space Filemark (FFILE)

  If the file is a contiguous file, the record pointer spaces forward to the number of the next record containing a filemark. Otherwise, the record pointer is set to the total number of records in the file.

- Backspace Record (BREC)

  The record pointer is decremented by 1 unless it is already pointing to record number 0.

- Forward Space Record (FREC)

  The record pointer is incremented by 1 unless incrementing by 1 would cause the pointer to exceed end of file (EOF).

Data can be transferred in either binary, image, or ASCII mode. The amount of data that can be accessed is determined by the file type, as listed below.

FILE TYPE	BYTE-LIMIT OF TRANSFER
Contiguous	2 - Capacity of file
Extendable Contiguous	2 - Capacity of file (Read)
	2 - Capacity of disk (Write)
Indexed	1 - Record length
Nonbuffered indexed	2 - Record length

4.7.1 Buffered Input/Output (I/O) (Indexed Files)

Indexed files use buffered I/O. When a data block of an indexed file is read or written, the transfer occurs between the system buffer and the file. Data is moved between the system buffer and the user buffer as requested by the task.

For example, to read data block 1 and data block 60, the data blocks are read into the system buffer before the records in the blocks are transferred to a user buffer. A data transfer is complete when one complete record has been moved into the user buffer or when the user buffer is full, whichever comes first. If a record does not fill the user's receiving buffer, the remaining bytes in the user buffer are unaffected.

When a write operation is performed, data is moved from the user buffer to the system buffer before transfer to disk. The open-ended structure of the indexed file allows the file size to be extended during a write operation up to the available free space on the disk. However, file extension can only be performed sequentially; i.e., each record added to the file must follow the last record written to the file. Random write operations can only be performed to an existing record in the file. For example, if an indexed file consists of 5 records, a request to write record 6 causes the file size to be extended to a 6-record length capacity. However, a request to write record 7 or higher to an indexed file containing 5 records would return EOF status.

If a binary record written to an indexed file is shorter than the file's record length, the remaining bytes of the record are automatically filled with zeros. ASCII records that are shorter than the file's record length are padded with blanks. If a record longer than the files record length is read or written, the data exceeding the record length is not transferred. Hence, the record length of an indexed file should be large enough to hold the largest possible amount of data that will be read or written during one data transfer operation.

4.7.2 Nonbuffered Input/Output (I/O)

Nonbuffered I/O is used for contiguous, extendable contiguous, and the nonbuffered indexed file types. Data is transferred directly between the user buffer and the file on disk. All but contiguous files can be extended during write operations. Both random and sequential I/O are supported by all three nonbuffered file types; however, some restrictions apply.

4.7.2.1 Accessing Contiguous Files

Data records for a contiguous file are transferred in blocks greater or smaller than the file's record length (256 bytes or one sector). All transfers begin on a sector boundary and must be an even number of bytes. If the amount of data written to a file does not equal 256 bytes, the data is left-justified in the sector and the last 2 bytes of the data are propagated to the end of the sector. Because contiguous files cannot be extended during write operations, random writes can only be performed on existing allocated sectors.

To extend the file length of a contiguous file, use OS/32 Copy to copy the file to another file of the desired size. See the OS/32 Copy User Guide for more information.

4.7.2.2 Accessing Nonbuffered Indexed Files

Nonbuffered indexed files provide the flexibility of indexed files without the use of system space for data buffers or the use of processor time for moving data between system space and the task's I/O buffer. For example, suppose a nonbuffered indexed file is made up of a 250-sector data block consisting of 240-byte records. Since each record begins on a sector boundary, there will be 250 records in the block. Because the size of each record is less than 256 bytes, each sector is filled with the last 2 bytes of the data. To read records 15 and 62,015, data blocks 1 and 249 are accessed. If a five-sector index block was specified, the addresses of both of these sectors would be in memory at the time of access when they would be transferred directly to the task's buffer. The time required to perform such a transfer is comparable to that required when using a contiguous file.

Like indexed files, the open-ended structure of a nonbuffered indexed file allows the file to be extended sequentially during write operations. Random write operations can only be performed on existing file records.

4.7.2.3 Accessing Extendable Contiguous Files

A sector in an extendable contiguous file is directly accessed in the same manner as a contiguous file. Multiple data block transfer requests, however, require a separate I/O operation for each block. Contiguous files require one I/O operation for a multiple-sector transfer.

For example, suppose an extendable file has data blocks consisting of 250 sectors per data block. To read sector number 15 and then sector number 62,015, simply access data blocks 1 and 249. If the index block for this file is at least 5 sectors, the addresses of both blocks are in memory at the time of access and they are transferred directly to the task's buffer. The time required to perform such a transfer is comparable to that required by a contiguous file.

Like nonbuffered indexed and indexed files, extendable contiguous files are open-ended. However, extendable contiguous files can be expanded sequentially and randomly. For example, a 10-sector file can be extended to a 20-sector file simply by writing to sector 20. The sectors between 10 and 20 are automatically allocated to the file. Essentially, for write operations no EOF exists. Hence, it is possible to completely fill a disk by writing to a sector with an unusually large random address.

Transfer of data begins and ends on a sector boundary. Partially filled sectors are padded with the last 2 bytes of the transferred data. In addition, an even number of bytes should be transferred; otherwise, the processor hardware will add one additional undefined byte to the buffer.

4.8 FILE SECURITY

As explained in Chapter 2, a task cannot perform its function if the data it acts on has been destroyed. When data is contained in main memory, it is protected by the relocation/protection hardware. However, if task data is stored on disk, access to the files containing the data must be controlled.

File access is controlled by matching a task with a set of permissable operations that they can perform on a given file. These operations are called access privileges. Access privileges are given to a task when a file is assigned to the task's lu. The access privileges are:

- Sharable read only (SRO)
- Exclusive read only (ERO)
- Sharable write only (SWO)
- Exclusive write only (EWO)
- Sharable read/write (SRW)
- Sharable read, exclusive write (SREW)
- Exclusive read, sharable write (ERSW)
- Exclusive read/write (ERW)

When multiple tasks are assigned to the same file, the access privileges for those tasks should be compatible. For example, one task cannot have EWO privileges to a file while another task has SWO privileges. Table 4-1 shows which access privileges are compatible. If a file is assigned to a task with access privileges that are incompatible with those previously assigned for another task, the access privileges for the second assignment will automatically default to the previous assignment.

TABLE 4-1 ACCESS PRIVILEGE COMPATIBILITY

	ERSW	ERO	SRO	SRW	SWO	EWO	SREW	ERW
ERSW	-	-	-	-	*	-	-	-
ERO	-	-	-	-	*	*	-	-
SRO	-	-	*	*	*	*	*	-
SRW	-	-	*	*	*	-	-	-
SWO	*	*	*	*	*	-	-	-
EWO	-	*	*	-	-	-	-	-
SREW	-	-	*	-	-	-	-	-
ERW	-	-	-	-	-	-	-	-

LEGEND

* Compatible
- Incompatible

If a file is assigned to multiple logical units for the same task, the file cannot be assigned for ERO on one lu and SRO on another. If a file is assigned for exclusive read or write access on any given lu, the file cannot be assigned for that access on any other lu.

A task can change its access privileges to a file without closing the file by requesting an access privilege change from the file manager. Allowable access privilege changes are shown in Table 4-2. If the task attempts to change an access privilege to one that is not allowed, the existing access privilege remains in effect.

TABLE 4-2 ALLOWABLE ACCESS PRIVILEGE CHANGES

CHANGE FROM	CHANGE TO							
	SRO	ERO	SWO	EWO	SRW	SREW	ERSW	ERW
SRO	x	x						
ERO	x	x						
SWO			x	x				
EWO			x	x				
SRW	x	x	x	x	x	x	x	x
SREW	x	x	x	x	x	x	x	x
ERSW	x	x	x	x	x	x	x	x
ERW	x	x	x	x	x	x	x	x

LEGEND

x indicates allowable change

A file can also be protected from read or write operations through the read/write keys that are given to the file when it is allocated. See Table 4-3. The read/write keys can protect a file from being read from or written to by any task assigned to it. For example, if a file's read and write keys are X'00' and X'07', respectively, its assigned task can read from that file but it cannot write to the file unless the file is assigned to the task with the same write key.

TABLE 4-3 READ/WRITE KEYS

WRITE KEY	READ KEY	MEANING
00	00	Not protected.
FF	FF	Unconditionally protected (used by executive tasks). See the OS/32 System Level Programmer Reference Manual.
07	00	Unprotected for read, conditionally. Protected for write. Task must match write key of X'07'.
FF	A7	Unconditionally protected for write, conditionally protected for read. Task must match read key of X'A7'.
00	FF	Unprotected for write, unconditionally protected for read
27	32	Conditionally protected for both read and write. Task must match both keys.

A task can change the keys of a file if the file has been assigned to the task with exclusive read or exclusive write privileges. For example, if the file is assigned to the task with the exclusive write only privilege, the write key can be changed. If the file is assigned to the task with exclusive read/write privileges, one or both keys can be changed.

Further protection is available when the disk is marked online. A disk volume can be marked online as write-protected. A write-protected volume will only accept assignments for SRO and SRW. (SRW is immediately changed to SRO). No other access privileges are permitted. If the write-protected feature of the disk hardware is enabled, the volume should also be marked on as a protected volume. See the OS/32 Operator Reference Manual for more information on marking on a disk.

4.9 CHOOSING THE RIGHT FILE TYPE

Not every file type is right for every real-time application. Record length, access method, and file expandability should all be taken into consideration when allocating and assigning files to a task. These and other file type characteristics are summarized in Table 4-4. The following sections describe the advantages and disadvantages of using each of the four file types, as well as some tips on handling disk fragmentation.

TABLE 4-4 FILE TYPE SUMMARY

TYPE	DATA ORGANIZATION	RECORD LENGTH (BYTES)	FILE LENGTH	BUFF-ERED I/O	RECORD POINTER VALUE	BYTE LIMIT/ TRANSFER
Contiguous	Contiguous	256	Fixed at allocation	NO	Sector number	2 - Capacity of file
Indexed	Linked-list indexed	1 to 65,535	Open-ended	YES	Data block number	1 - Record length
Nonbuffered Indexed	Linked-list indexed	2 to 65,535	Open-ended	NO	Data block number	2 - Record length
Extendable Contiguous	Linked-list indexed	256	Open-ended	NO	Sector number	2 - Capacity of file (read) 2 - Available disk space (write)

4.9.1 Using Contiguous Files

The primary advantage of using contiguous files is that all space required for the file is fixed when the file is allocated. Since the maximum file length cannot be changed, the user knows how much data can be input. This advantage should be weighed against the cost of losing file space when a contiguous file that contains a large number of unused sectors exists on disk. Contiguous files also support overlapped I/O and program execution. For all other file types, the task actually waits for an I/O operation to complete, even if a proceed I/O request was made. See the OS/32 Supervisor Call (SVC) Reference Manual for more information on I/O proceed requests.

Another characteristic of contiguous files that proves helpful in some applications (e.g., magnetic tape emulation) is the ability to support filemarks.

Finally, to achieve the fastest possible access time for applications that perform a large number of random read and write operations, use contiguous files.

4.9.2 Using Indexed Files

The advantage of using indexed files is that the user does not have to compute the size of the file before allocation. Hence, indexed files are best suited for applications where file size continues to expand throughout the life of the file.

Many applications, such as compiling or assembling source code, are I/O bound; i.e., the time required to complete the job depends primarily on the speed of the sequential I/O to and from the disk. In these circumstances, indexed files with moderate block sizes provide the best throughput. Due to the additional central processing unit (CPU) overhead caused by buffering operations, the use of data block sizes of 5 to 20 has been found to cause a job to become CPU limited. However, many simple tasks do not become CPU limited until much larger block sizes are used.

It should be remembered that for strictly sequential access applications, nonbuffered indexed, extendable contiguous, and contiguous files all offer the same throughput, and all three are usually lower in performance than indexed files.

The random access performance of indexed files depends on the correct choice of index and data block sizes. The optimal choice of data block size can usually be determined only by experiment for a particular application. In general, the index block size should be such that the file requires only one index block. (In reality, this is often not possible since the data blocks supported by indexed files are not as large as those supported by nonbuffered indexed and extendable contiguous files.)

4.9.3 Using Nonbuffered Indexed Files

The purpose of nonbuffered files is two-fold. They provide the user with excellent random access performance for files of arbitrary logical record length, and they eliminate the CPU overhead and main memory requirements associated with buffered indexed files.

For CPU bound processes, or those processes that perform only random I/O on very large files of arbitrary record size, nonbuffered indexed files are preferred. Because these files have no data buffers in main memory, some users may prefer to use nonbuffered indexed files to conserve memory space, even at the expense of performance in typical sequential access operations.

Like indexed files, the random access performance of nonbuffered indexed files also depends on the correct choice of index and data block sizes. Hence, the maximum possible data block size should always be used, unless disk fragmentation prevents such large block sizes.

Nonbuffered indexed files are also suited for applications whose total file size continues to expand throughout the life of the file.

4.9.4 Using Extendable Contiguous Files

Extendable contiguous files provide all of the random access and performance advantages of contiguous files, without the drawback of fixed file sizes. Care should be taken, however, in choosing data and index block sizes to ensure the best possible performance. For example, suppose that an application requires a contiguous file of 200,000 sectors. Using the largest possible data block size (255 sectors), there would be 785 data blocks. These data blocks could be pointed to from one index block of 13 sectors. Thus, for a cost of 13 sectors (3.25kb) of system space, the entire file index could be contained in memory. This would allow random access to the file to be the same as to a contiguous file.

Extendable contiguous files are also suited for applications whose total file size continues to expand throughout the life of the file.

4.9.5 Disk Fragmentation

The process of repeatedly allocating, expanding, and deleting files of various sizes and record lengths eventually results in disk fragmentation. Here, fragmentation means that the available free space on the disk is divided among a great many relatively small areas. These areas may be as small as one sector or may cover the total available free space on the disk.

On the average, there are fewer places to allocate a large physical block than a small physical block. On badly fragmented disks, the maximum block size that can be allocated may be very small. Also, the time required to allocate any given block will increase with increasing block size or increasing disk fragmentation, since there are generally fewer locations where the block will fit (that is, it takes somewhat longer to locate a large free space than a small free space).

Disk fragmentation and total amount of free space both determine the maximum possible file size for any given physical block size. For example, on a given disk the maximum file size might be 150,000 sectors if the block size were 5 sectors, but only 90,000 sectors if the block size were 64 sectors, and only 40,000 sectors if the block size were 250. On the same disk, the largest contiguous file that could be allocated would be 20,000 sectors.

Once a disk is badly fragmented, the only option available is to compress the disk. To compress a disk, initialize a new disk pack on another drive and copy all files from the fragmented pack to the newly initialized pack. If only a single drive is available (or no additional packs are available), the fragmented pack can be backed up to tape, reinitialized, then restored from the tape.

CHAPTER 5
WRITING PROGRAMS THAT ACCESS OS/32 SYSTEM SERVICES

5.1 INTRODUCTION

The OS/32 supervisor calls (SVCs) provide the task interface to OS/32 system services. These calls activate the appropriate OS/32 executor routines that can handle the user's requests. For example, to request use of a system resource, a task issues an SVC 7. To request transfer of information to the resource given to the task by SVC 7, an SVC 1 is issued.

When a task calls an executor routine through an SVC, the task must pass the information needed by the routine to perform the requested function. For example, to transfer data from a disk file, the operating system requires the address of the user buffer to which the data is to be sent. This information is passed through a special OS/32 data structure called the SVC parameter block. OS/32 provides a separate parameter block structure for each type of SVC that can be issued by the task. The task builds a parameter block in its task address space and stores the information required by the executor routine in that block. When the SVC is issued, the operating system refers to data stored in the parameter block during execution of the routine.

Perkin-Elmer provides a number of methods for writing application programs that access system services. A programmer working in OS/32 Common Assembly Language (CAL) can write a program that directly issues an SVC or executes an OS/32 system macro library routine that issues an SVC. A FORTRAN program can issue an SVC by calling a Perkin-Elmer Run-Time Library (RTL) routine that issues the SVC. If a FORTRAN program is to access the file manager, the FORTRAN VII auxiliary input/output (I/O) statements can be used. Finally, a Pascal program can access system services through procedures contained in the standard Pascal Prefix supplied with the Perkin-Elmer Pascal compiler. These programming methods are outlined in Figure 5-1.

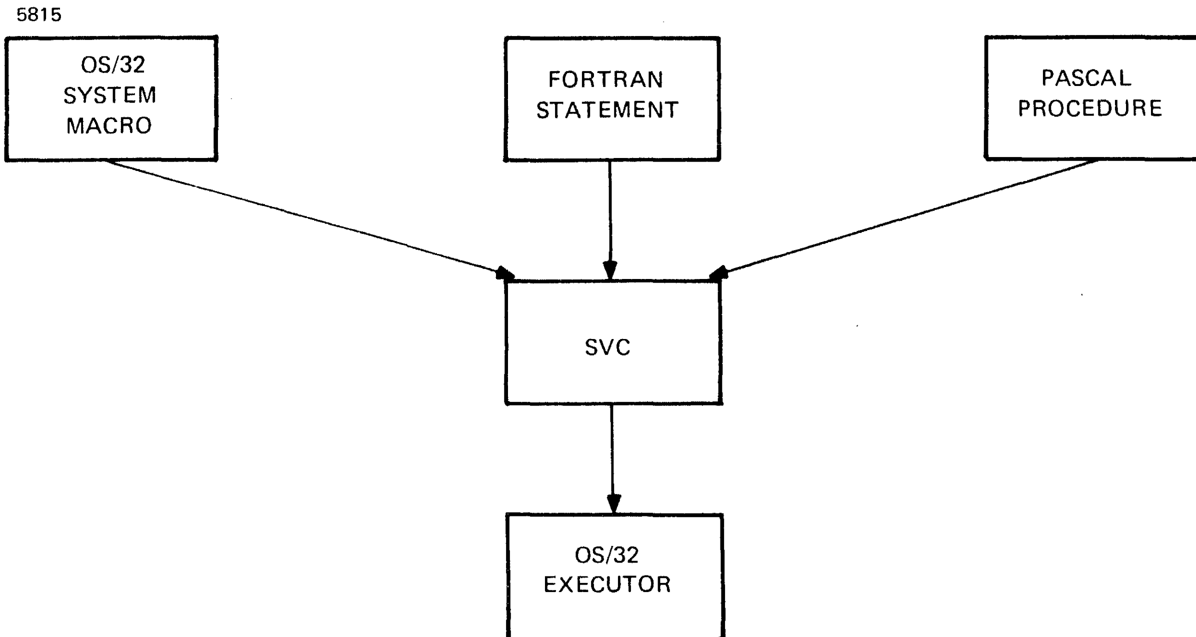


Figure 5-1 Task Interface to OS/32 Executor Routines

This chapter demonstrates how each of these methods are used to write a program that accesses two OS/32 system services; namely, the I/O and file management services. First, the SVC 1 and SVC 7 parameter block structures that pass data to the I/O and file management executor routines are discussed. Refer to the OS/32 Supervisor Call (SVC) Reference Manual for more details on the individual parameters of the SVC parameter blocks discussed in these sections.

5.2 BUILDING A SUPERVISOR CALL (SVC) PARAMETER BLOCK

The OS/32 macro library SYSSTRUC.MLB provides macro routines that define parameter blocks for SVC 1, SVC 5, SVC 6, SVC 7, and SVC 13 in a task's address space. These macro routines are listed in the OS/32 Supervisor Call (SVC) Reference Manual. To build a parameter block structure within a task's address space, expand the appropriate OS/32 system macro routine and store the information required by the OS/32 executor routine in the parameter block.

5.2.1 Accessing Input/Output (I/O) System Services

To request an I/O service, the task first defines an SVC 1 parameter block using the $SVC1 macro. $SVC1 defines the parameter block shown in Figure 5-2.

5816

0(0) FUNCTION CODE (SVC1.FUN)	1(1) LU (SVC1.LU)	2(2) DEVICE INDEPENDENT STATUS (SVC1.STA)	3(3) DEVICE DEPENDENT STATUS (SVC1.DN)
4(4) BUFFER START ADDRESS (SVC1.SAD)			
8(8) BUFFER END ADDRESS (SVC1.EAD)			
12(C) RANDOM ADDRESS (SVC1.RAD)			
16(10) LENTH OF DATA TRANSFER (SVC1.LXF)			
20(14) EXTENDED OPTIONS (SVC1.XOP)			

Figure 5-2 SVC 1 Parameter Block Defined by $SVC1

TABLE 5-1 SVC 1 FUNCTION CODES

EQUATE	FUNCTION CODE	MEANING
SV1.CMDF	X'80'	Command
SV1.READ	X'40'	Read
SV1.WRIT	X'20'	Write
SV1.BIN	X'10'	Binary
SV1.WAIT	X'08'	Wait
SV1.RAND	X'04'	Random
SV1.UPRO	X'02'	Unconditional proceed
SV1.IMG	X'01'	Image mode
SV1.XOP	X'01'	Use extended options
SV1.XIT	X'01'	Use data communications extended option word
SV1.REW	X'C0'	Rewind
SV1.BSR	X'A0'	Backspace record
SV1.FSR	X'90'	Forward-space record
SV1.WFM	X'88'	Write filemark
SV1.FFM	X'84'	Forward-space filemark
SV1.BFM	X'82'	Backspace filemark
SV1.DDF	X'81'	Device dependent function
SV1.HALT	X'80'	Halt I/O
SV1.SET	X'60'	Test and set
SV1.WO	X'08'	Wait only
SV1.TEST	X'02'	Test I/O completion

Notice that the SVC 1 parameter block is divided into specific fields that contain the data required by the OS/32 I/O subsystem to perform the requested operation. The first field contains the function code indicating the particular I/O function to be performed. See Table 5-1. For example, a parameter block for a read request contains the following:

- SV1.READ function code
- logical unit (lu) assigned to the disk file to be read
- starting and ending addresses of the user buffer

The following code builds an SVC 1 parameter block for an SVC 1 that requests a data transfer from a file or device assigned to lu1 to the user buffer at location BUFF.

Example:

```
         $SVC1                         DEFINE THE STRUCTURE
         ALIGN 4
SVC.IN   DS     SVC1.                  ALLOCATE STORAGE FOR PARBLK
ENDPBK   EQU    *
         ORG    SVC1.IN+SVC1.FUN       INITIALIZE FIELDS
         DB     SV1.READ               FUNCTION CODE
         DFB    1                      LOGICAL UNIT=1
         ORG    SVC1.IN+SVC1.SAD
         DC     A(BUFF)                BUFFER START ADDRESS
         DC     A(BUFFE)               BUFFER END ADDRESS
         ORG    ENDPBK
```

To request this operation, the task issues the SVC 1 as follows:

```
         SVC    1,SVC.IN
```

The following program uses SVC 1 to build two parameter blocks, one for an SVC 1 that performs a read operation from a file or device assigned to lu1 and one for an SVC 1 that performs a write operation to a file or device assigned to lu2. Notice that this program uses the function code SV1.WAIT to allow task execution to be suspended until each data transfer operation is complete. Also notice that the program checks the status field of the SVC 1 parameter block to determine if the I/O operation was successful.

Example:

```
SVC1     PROG  SIMPLE SVC 1 EXAMPLE
         MLIBS 8,9,10               DECLARE MACRO LIB TO CAL/MACRO
         NLSTM                      DON'T LIST MACRO EXPANSIONS
         FREZE                      FREEZE LINE NUMBERS
         $SVC1                      DEFINE SVC 1 STRUCTURE
         ALIGN 4                    ALIGN PARBLK ON FULLWORD
SVC1.IN  DS    SVC1.                ALLOCATE INPUT PARBLK
ENDPBK   EQU   *
         ORG   SVC1.IN+SVC1.FUN
         DB    SV1.READ!SV1.WAIT    FUNCTION CODE=READ & WAIT
         DB    1                    LOGICAL UNIT=1
         ORG   SVC1.IN+SVC1.SAD
         DC    A(BUFF)              BUFFER START ADDRESS
         DC    A(BUFFE)             BUFFER END ADDRESS
         ORG   ENDPBK
BUFF     DS    80                   ALLOCATE 80-BYTE BUFFER
BUFFE    EQU   *-1
         ALIGN 4
SVC1.OUT DS    SVC1.                ALLOCATE OUTPUT PARBLK
ENDPBK   EQU   *
         ORG   SVC1.OUT+SVC1.FUN
         DB    SV1.WRIT!SV1.WAIT    FUNC CODE=WRITE & WAIT
         DB    2                    LOGICAL UNIT=2
         ORG   SVC1.OUT+SVC1.SAD
         DC    A(BUFF)              SAME BUFFER AS INPUT
         DC    A(BUFFE)
         ORG   ENDPBK
START    EQU   *
         SVC   1,SVC1.IN            ISSUE SVC 1 TO READ LU1
         LH    0,SVC1.IN+SVC1.STA   CHECK STATUS
         BM    ERROR                <0, ERROR
         SVC   1,SVC1.OUT           ISSUE OUTPUT SVC TO LU2
         LH    0,SVC1.OUT+SVC1.STA  CHECK STATUS
         BM    ERROR                <0, ERROR
         SVC   3,0                  NORMAL EOT=0
ERROR    EQU   *
         SVC   3,1                  EOT=1
         END   START
```

5.2.2 Accessing File Management Services

All file management requests are made through SVC 7. For example, a task requests an lu assignment via the SVC 7 assign function. The operating system assigns the resource (file or device) to the requesting task's lu. The task proceeds to use it as required. When the task no longer needs the lu, the SVC 7 close function is used to cancel the assignment.

The SVC 7 parameter block that passes the necessary information to the file manager is shown in Figure 5-3. To define this structure, use the $SVC7 macro. Notice that the SVC 7 parameter block contains the file descriptor (fd) of the file on which the OS/32 executor is to perform the requested operation.

6021

0(0) FUNCTION CODE (SVC7.OPT)		2(2) ERROR STATUS (SVC7.STA)	3(3) LU (SVC7.LU)
4(4) WRITE KEY (SVC7.WKY)	5(5) READ KEY (SVC7.RKY)	6(6) LOGICAL RECORD LENGTH (SVC7.LRC)	
8(8) VOLUME NAME OR DEVICE MNEMONIC (SVC7.VOL)			
12(C) / 16(10) FILENAME (SVC7.FNM)			
20(14) EXTENSION (SVC7.EXT)			23(17) FILE CLASS/ ACCOUNT NO. (SVC7.ACT)
24(18) INDEX BLOCK SIZE (SVC7.ISZ)		26(1A) DATA BLOCK SIZE (SVC7.DSZ)	

Figure 5-3 SVC 7 Parameter Block Defined by $SVC7

SVC 2 code 16 can be used to pack an fd into the SVC 7 parameter block. This SVC must be used if the fd to be packed into the block specifies an account number. The parameter block for SVC 2 code 16 is shown in Figure 5-4. Note that there is no macro available for defining this structure. See the OS/32 Supervisor Call (SVC) Reference Manual for more information on coding the SVC 2 code 16 parameter block.

5818

0(0) OPTION	1(1) CODE	2(2) USER REGISTER
4(4) ADDRESS OF PACKED FD AREA		

Figure 5-4 SVC 2 Code 16 Parameter Block

The following program issues an SVC 7 and SVC 2 code 16 to assign lu2 to a file named MYFILE.TXT/P. The program uses $SVC7 to define an SVC 7 parameter block. A parameter block is also built for SVC 2 code 16. This block contains the number of the register that holds the fd to be packed and the address of the SVC 7 parameter block field where the fd is to be packed. No error checking is performed by this program.

Example:

```
         MLIBS 8,9
         PROG  ASSIGN
         $SVC7
ASSIGN   DS    SVC7.                BUILD SVC 7 PRBLK
ASSIGNE  EQU   *
         ORG   ASSIGN+SVC7.OPT
         DC    SV7.ASGN!SV7.SRW     SET FUNCTION CODE & ACCESS PRIV
         ORG   ASSIGN+SVC7.LU
         DB    X'2'                 INDICATE LU TO BE ASSIGNED
         ORG   ASSIGNE
         ALIGN 4
PACK     EQU   *                    BUILD SVC 2,16 PRBLK
         DB    X'10'                SET DEFAULT VOLUME OPTION CODE
         DB    16                   SET SVC CODE
         DC    X'1'
         DC    A(ASSIGN+SVC7.VOL)   STORE ADR OF PACKED FD AREA
         ALIGN 4
FD       DB    C'MYFILE.TXT/P'
         ALIGN 4
START    EQU   *
         LA    1,FD                 LOAD FD INTO REG 1
         SVC   2,PACK
         SVC   7,ASSIGN
         SVC   3,0                  END TASK WITH EOT 0
         END   START
```

The above examples represent only three of the I/O and file management services that can be accessed by an application program. For more information on other OS/32 system services, see the OS/32 Supervisor Call (SVC) Reference Manual.

5.3 USING THE OS/32 SYSTEM MACRO LIBRARY TO ACCESS SYSTEM SERVICES

The OS/32 system macro library provides macro routines that not only build an SVC parameter block but also issue the SVC for a task. The programmer simply provides the necessary data for the OS/32 executor as operands to the macro instruction that expands the routine. For example, to output data from the user buffer, BUFF, to the file or device assigned to lu2, execute the WRITE macro instruction as follows:

```
WRITE LU=2,ADDR=BUFF,REGL=80,ENDADDR=BUFFE
```

The WRITE macro routine builds an SVC 1 parameter block with the SV1.WRIT function code set. Using the operands specified in the macro instruction, the WRITE routine stores the values for lu, record length (REGL), and the user buffer's starting and ending address in the appropriate fields of the SVC 1 parameter block. The routine then issues SVC 1.

The following assembly program uses system macros to access the read, write, assign, and allocate OS/32 executor routines.

Example:

```
             PROG   SVC 1 AND SVC 7 MACRO EXAMPLE
             MLIBS  8,9,10
    BUFF     DS     80
    BUFFE    EQU    *-1
    START    EQU    *
     ALAS FD='TEST2.DTA',LU=2,AP=SRW,RECL=80,FT=IN,BLKSIZE=1,NDXSIZE=1
     ASSIGN LU=1,FD='CARDIN.FMU/S'
     READ LU=1,ADDR=BUFF,RECL=80,ENDADDR=BUFFE
     WRITE LU=2,ADDR=BUFF,RECL=80,ENDADDR=BUFFE
     EOT RC=0
             END    START
```

In the above example, the ALAS macro routine builds an SVC 7 parameter block and then issues an SVC 7 to allocate the file TEST2.DTA and assign the file to lu2. The ASSIGN macro routine builds another SVC 7 parameter block and issues an SVC 7 to assign CARDIN.FMU/S to lu1.

The READ macro routine builds an SVC 1 parameter block and issues an SVC 1 to read data from CARDIN.FMU/S into the user buffer at location BUFF. The WRITE macro routine builds an SVC 1 parameter block and issues an SVC 1 to write data from BUFF to the indexed file TEST2.DTA.

See the OS/32 System Macro Library Reference Manual for details on how to use the OS/32 macro routines for writing assembler language programs that access OS/32 system services.

5.4 WRITING A FORTRAN PROGRAM THAT ACCESSES SYSTEM SERVICES

The Perkin-Elmer FORTRAN VII RTL provides subroutines that allow access to system services through a FORTRAN application program. Like the OS/32 system macro library routines, these subroutines build the SVC parameter block and issue the SVC for the program. The programmer simply calls the RTL routine specifying the required SVC parameters as arguments to the call. For example, the SYSIO RTL routine is used to access OS/32 I/O services. SYSIO builds an SVC 1 parameter block using the arguments in the CALL SYSIO statement. After the block is built, SYSIO issues an SVC 1 to perform the requested I/O operation.

The following example uses SYSIO to transfer data from the disk file assigned to lu2 to the user buffer at location BUFF. A second RTL subroutine, IOERR, is called to interpret the status of the I/O request after the operation is completed. IOERR places the status code contained in the error status field of the SVC 1 parameter block into the argument ISTATUS and outputs a message to the device assigned to lu6.

Example:

```
     INTEGER  LU,ISTATUS,NBYTES,RANADD,FC
     INTEGER  PBLK(5), BUFF(20)
     LU=2
     NBYTES=80
     RANADD=0
     ISTATUS=0
     FC=Y'28' ; SVC1 READ WAIT FUNCTION CODE
     CALL SYSIO(PBLK,FC,LU,BUFF,NBYTES,RANADD)
     CALL IOERR(PBLK,ISTATUS)
     IF (ISTATUS.NE.0) GO TO 10
         .
         .
         .
```

In the above example, the function code Y'28' tells the OS/32 executor to perform the write operation and suspend task execution until the data transfer is completed. PBLK specifies the array in which the parameter block will be built. LU is the logical unit assigned to the output device. BUFF is the buffer array that is to be output. NBYTES specifies the number of bytes that will be output.

See the FORTRAN VII User Guide for more information on using the RTL routines to access system services.

To access file management services, the Perkin-Elmer FORTRAN VII compilers provide auxilliary I/O statements; e.g., OPEN and CLOSE. These statements are similar to the system macro library and RTL routines in that they build the necessary parameter block and call the SVC. These statements include parameters required to perform the specific file management function.

Example:

```
OPEN (UNIT=1,FILE=CARDIN.FMU/S,STATUS=OLD,ERR=100)
```

The above example assigns the file CARDIN.FMU/S to lu1 by building an SVC 7 parameter block and issuing an SVC 7 to assign the file. STATUS=OLD tells the operating system that the file has already been allocated. If the file assignment operation ends in error, the program branches to statement label 100.

See the FORTRAN VII Reference Manual for more information on the use of auxiliary I/O statements.

5.5 WRITING A PASCAL PROGRAM THAT ACCESSES SYSTEM SERVICES

The standard Pascal Prefix supplied with the Perkin-Elmer Pascal package contains CONST, TYPE and PROCEDURE declarations that can be used to access system services. The Prefix also supplies procedures that provide access to system or SVC services. Like the FORTRAN VII standard RTL routines, these Pascal procedures both build the SVC parameter block and issue the SVC. The data required by the OS/32 executor is passed via the procedure parameters as shown in the following example.

Example:

```
(*$INCLUDE(PREFIX.PAS/S)*)
PROGRAM SAMPLEPAS(OUTPUT)

VAR STATUS:BYTE;

BEGIN
  OPEN(1,'M300:CMDȸȸȸȸȸ.FIL/P',SRW,0,STATUS);
  IF STATUS< >0 THEN
     WRITELN ('ERROR STATUS=',STATUS);
END
```

The above procedure uses the OPEN Prefix procedure to assign the file M300:CMD.FIL/P to lu1. With the OPEN procedure, an SVC 7 parameter block is built and an SVC 7 is issued to assign the lu. This program checks the status field returned by the OPEN procedure. This status is the SVC 7 parameter block status after the I/O operation is completed. If an error has occurred, this program outputs an error message.

See the OS/32 Pascal User Guide, Language Reference, and Run-Time Support Reference Manual for details on how to use the Pascal Prefix to write a Pascal program that accesses system services.

INDEX

A

Absolute code 2-4
Access privileges 4-18, 4-20
   compatibility of 4-19
   write only 4-15
APU. See auxiliary processing unit.
Arithmetic faults 3-5, 3-8
   reason codes 3-6
   traps 3-18
Assign 4-13, 4-15
AUF. See authorized user file.
Authorized user file 2-14
Authorized user utility 1-5
Auxiliary I/O statements 5-1, 5-9
Auxiliary processing unit 3-12, 3-25

B

Backspace filemark 4-15
Backspace record 4-15
BFILE. See backspace filemark.
Bit map 4-8, 4-12
BREC. See backspace record.

C

Central processing unit 3-12
Circular list 3-8, 3-22
Common assembly language 5-1
Condition code 3-2
Contiguous files 4-6
   advantages of 4-22, 4-23
   byte-limit of transfer 4-16
   file length 4-17
   I/O operations 4-7, 4-22
   maximum file length 4-7
   record length 4-17
   record size 4-7
   write operations 4-16, 4-17
Contiguous organization 4-4, 4-6
CPU. See central processing unit.
CTOP 2-9

D

D-task. See diagnostic task.
Data blocks 4-5, 4-6, 4-7, 4-11, 4-16, 4-17, 4-18, 4-23
Data format/alignment faults 3-5, 3-8
   reason codes 3-7
Data record 4-3, 4-6
DEBUG/32 2-4
Debugger 2-4
Device drivers 3-25, 4-2, 4-13
Device-independent I/O 4-13
Diagnostic task 2-2
Disk
   compression of 4-25
   fixed-head 4-2
   formatted 4-3
   fragmentation of 4-24, 4-25
   moving-head 4-2
   sector 4-4, 4-7, 4-10, 4-12
   track 4-3
   volume 4-21
Disk initializer 4-3, 4-8, 4-12
Disk volume
   write-protected 4-21
Dormant state 2-11

E

E-task. See executive task.
Executive task 2-2
Extendable contiguous files 4-6, 4-17
   advantages of 4-24
   byte-limit of transfer 4-16
   I/O operations 4-7
   maximum file length 4-7
   record size 4-7
   write operations 4-16, 4-18

F,G

Faults
   reason code 3-6
   types of 3-5
fd. See file descriptor.
FFILE. See forward space filemark.
File access methods
   buffered 4-14, 4-16
   interfaces to 4-14
   nonbuffered 4-14, 4-16, 4-17
File descriptor 4-2, 5-5
File directories
   primary 4-8, 4-9, 4-10, 4-11
   secondary 4-11
File management services 5-2, 5-5
   allocate 4-1
   checkpoint 4-1
   close 4-1
   delete 4-1
   fetch attributes 4-1
   open. See also assign. 4-1
   rename 4-1
File manager 4-1
   operations 4-8
   organization of 4-3
   supported devices 4-2
File organization
   contiguous 4-4, 4-6
   linked-list indexed 4-4, 4-5
File types 4-6
   choosing 4-22
   contiguous 1-4
   extendable contiguous 1-4
   indexed 1-4
   nonbuffered indexed 1-4
   summary of 4-22
Filemarks 4-6, 4-7, 4-23
Files
   access methods 4-14
   access privileges 4-11, 4-18, 4-19, 4-20
   account numbers 4-3, 4-10
   attributes 4-11
   buffered 4-5, 4-14, 4-16
   byte-limit of transfer 4-16
   classes 4-3
Files (Continued)
   definition of 4-2
   naming of 4-10
   nonbuffered 4-14, 4-16, 4-17
   permanent 4-13
   random access 4-15
   read/write keys 4-20, 4-21
   record pointer 4-14
   records 4-11, 4-15, 4-16
   security 4-18
   sequential access 4-15
   size of 4-25
   temporary 4-13
   types of 4-6
FORTRAN VII
   auxiliary I/O statements 5-1, 5-9
   run-time library 5-9
FORTRAN VII run-time library 3-27
Forward space filemark 4-15
Forward space record 4-15
FREC. See forward space record.
Function code 5-4

H

History records 2-2

I,J,K

I/O subsystem 4-13, 4-14, 5-4
I/O system services 5-2
Illegal instruction faults 3-5, 3-8
Image file 2-1
   format 2-2
Impure code 2-2
Index blocks 4-5, 4-10, 4-11, 4-18, 4-23, 4-24
Indexed files
   advantages of 4-23
   buffered 4-5, 4-7, 4-16
   byte-limit of transfer 4-16
   I/O operations 4-7
   maximum file length 4-7
   nonbuffered 4-6, 4-16, 4-17
   read operations 4-16
   record length 4-16

Indexed files (Continued)
– record size 4-7
– size of 4-16
– write operations 4-16
Instrument Society of America 3-25
Intertask communication
– restrictions on 2-14
Intertask control services 3-24
ISA. See Instrument Society of America.

L

LIB. See loader information block.
Link 1-3, 2-1
Linked-list indexed organization 4-4, 4-5, 4-7
Loader information block 2-2
LOC. See location counter.
Location counter 3-1, 3-2
Logical unit 2-13, 4-13, 4-14, 4-18, 4-19
lu. See logical unit.

M

MAC. See memory access controller.
MAT. See memory address translator.
Memory access controller 2-5
Memory access faults 3-5, 3-8
– reason codes 3-6
Memory address translator 2-5
Message buffer queue 3-22
Message ring 3-22
Model 3200MPS System 3-12, 3-25
Monitors 1-1, 1-5, 2-13, 2-14
MTM. See OS/32 Multi-Terminal Monitor.

N

Nonbuffered indexed files 4-6
– advantages of 4-23, 4-24
– byte-limit of transfer 4-16
– hardware restrictions 4-7
– I/O operations 4-7
– maximum file length 4-7
Nonbuffered indexed files (Continued)
– record size 4-7
– write operations 4-16, 4-17
Nonresident task 2-11
NULL: device 4-13

O

Object libraries 2-4
Object modules 2-4
ODT. See overlay descriptor table.
OPEN. See also assign.
OS/32 5-4
– common assembly language 5-1
– disk initializer 4-3, 4-8, 4-12
– features of 1-1, 1-2
– file management services 4-1, 5-2, 5-5
– file manager 4-1
– I/O subsystem 4-2, 5-4
– I/O system services 5-2
– intertask control services 3-24
– linkage editor 1-3, 2-1
– loader 2-2
– monitors 1-1
– operating environments 1-1
– parameter block 5-8
– supervisor call 5-1
– system macro library 3-27, 5-1, 5-8
OS/32 Copy 4-17
OS/32 Multi-Terminal Monitor 1-1, 1-4
– account privileges 1-5
– batch jobs 2-13
– subtasks 2-13
OS/32 PATCH 2-2
OS/32 real-time environment
– fault conditions 1-3
– file management services 1-4
– I/O services 1-3
– interrupt servicing 3-1
– memory addressing 1-3
– message service 1-4
– priority levels 1-3
– spooling utilities 1-4
– timer facility 1-3
OS/32 Spooler 1-4
OS/32 system macro library 3-27
OS/32 time-sharing environment 1-4
Overlay descriptor table 2-4
Overlays 2-4, 2-9

P,Q

Page 2-5
Parameter block 5-1, 5-2, 5-5, 5-6, 5-8, 5-9, 5-10
Partial images 2-5, 2-7
Pascal Prefix 5-1
  example 5-10
  use of 5-10
Permanent files 4-13
Power restoration 3-5, 3-8
Prefix, Pascal 5-1
  example 5-10
  use of 5-10
Primary directory block 4-11
Primary directory entry 4-10
Primary file directory 4-8, 4-9
Private image segment 2-2, 3-14
Program status word 2-4
PSW. See program status word.
Pure code 2-2, 2-4

R

Random access 4-15
Random I/O 4-16
Read/write keys 4-20
  changing of 4-21
Record
  ASCII 4-16
  binary 4-16
Record pointer 4-14, 4-15
RELIANCE 1-1
Relocation/protection hardware 2-2, 2-5
Resident task 2-11
Rewind 4-15
Rolled state 2-11
Root node 2-9
Root segment 2-4, 2-5
RTL. See run-time library.
Run-time library 5-1
  use of 5-9

S

Secondary file directory 4-11
  size of 4-11
Secondary storage devices 4-1
Sector 4-4, 4-7, 4-10, 4-12
Send data trap 3-22
Send message trap 3-22
  example of 3-25
Send message trap handler
  example of 3-23
Sequential access 4-15
Sequential I/O 4-16
Shared image segment 2-4
Shared segment 2-9
SPL/32 1-4
Spooling utilities 1-4
Subtasks 2-13, 2-14
  state change 3-10
Supervisor calls
  example 5-4, 5-5, 5-7
  parameter block 5-1
  use of 5-2, 5-4, 5-5, 5-7, 5-2
SVC 1 5-1, 5-5
  example 5-9
  parameter block 5-3, 5-8
SVC 14 task trap 3-8
  reason code 3-12
SVC 2 code 16
  parameter block 5-6
SVC 6 2-13, 3-12
SVC 7 5-1
  parameter block 5-5, 5-6, 5-8
SVC 9 3-15
SVC. See supervisor call.
System macro library 5-1
  example 5-4, 5-8
  use of 5-2, 5-8
System structure macro library 3-18
System tasks 2-14

T

Task 2-1
  background 2-14
  foreground 2-14
  logical address space 2-9
  nonresident 2-11
  nonsegmented 2-9
  object modules 2-4
  overlays 2-4

Task (Continued)
   priorities 2-9
   private area 2-7
   resident 2-11
   segmented 2-4, 2-7, 2-9
   states 2-9
   system 2-14
   wait states 2-10
   workspace 2-4, 2-9
Task address space 2-6, 2-7
Task control block 2-4, 3-2, 3-15, 3-16, 3-18
Task image 2-1
Task queue 3-22, 3-25
Task queue trap handler
   example of 3-22
Task queue trap handling routines
   examples 3-23, 3-24, 3-25
Task queue traps 3-8, 3-9, 3-12, 3-22
   reason codes 3-9, 3-10, 3-11
Task scheduler 2-9
Task states 2-9
   current 2-9
   dormant 2-11
   ready 2-9
   rolled 2-11
   wait 2-9
Task status word 2-4, 3-1, 3-8, 3-14, 3-18, 3-25
   bit definitions 3-2
Task status word swap 3-12, 3-14, 3-15, 3-16
Task trap 3-2, 3-18
Task trap handler
   example of 3-18, 3-25, 3-27, 3-28, 3-29
Task trap handling routine 3-23
Task wait states 2-10
TCB. See task control block.
Temporary files 4-13
Track 4-3
Transaction processing 1-1
Trap-causing events
   arithmetic fault 3-6
   data format/alignment fault 3-7
   memory access fault 3-6, 3-7
   task queue 3-9
   user-defined 3-12
Trap-generating devices 3-24, 3-25
Trap handlers, system default 3-2
TSW. See task status word.

U

U-task. See user-task.
UBOT 2-9
UDL. See user-dedicated location.
User-dedicated location 2-4, 3-1, 3-12, 3-13, 3-14, 3-16, 3-18
User-task 2-2
   logical address space 2-5
   nonprivileged 2-2
   privileged 2-2
UTOP 2-9

V

Virtual task manager 1-3
Volume descriptor 4-8
VTM. See virtual task manager.

W-Z

Workspace 2-9

PUBLICATION COMMENT FORM

Please use this postage-paid form to make any comments, suggestions, criticisms, etc. concerning this publication.

From ______________________ Date ______________________

Title ______________________ Publication Title ______________________

Company ______________________ Publication Number ______________________

Address ______________________

____

____

FOLD FOLD

Check the appropriate item.

☐ Error Page No. ________ Drawing No. ______________

☐ Addition Page No. ________ Drawing No. ______________

☐ Other Page No. ________ Drawing No. ______________

Explanation:

FOLD FOLD

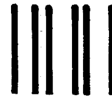
CUT ALONG LINE

Fold and Staple
No postage necessary if mailed in U.S.A.

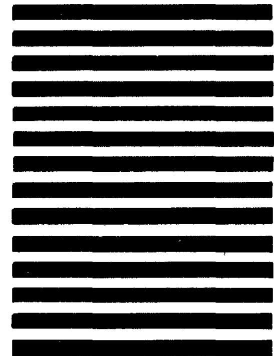
6434

STAPLE STAPLE

FOLD FOLD



NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES



BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 22 OCEANPORT, N.J.

POSTAGE WILL BE PAID BY ADDRESSEE

PERKIN-ELMER
Computer Systems Division
2 Crescent Place
Oceanport, NJ 07757

TECH PUBLICATIONS DEPT. MS 322A

FOLD FOLD

STAPLE STAPLE