

# **OS/32 APPLICATION LEVEL PROGRAMMER**

Reference Manual

OS/32 Version 8.2 or higher

48-039 F00 R03



The information contained in this document is subject to change without notice. Concurrent Computer Corporation has taken efforts to remove errors from this document, however, Concurrent Computer Corporation's only liability regarding errors that may still exist is to correct said errors upon their being made known to Concurrent Computer Corporation.

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 all copyright notices, trademarks, or other legends or credits of Concurrent Computer Corporation and/or its suppliers. Title to and ownership of the described software and any copies thereof shall remain in Concurrent Computer Corporation and/or its suppliers.

The licensed program described herein may contain certain encryptions or other devices which may prevent or detect unauthorized use of the Licensed Software. Temporary use permitted by the terms of the License Agreement may require assistance from Concurrent Computer Corporation.

Concurrent Computer Corporation assumes no responsibility for the use or reliability of this software if used on equipment that is not supplied by Concurrent Computer Corporation.

© 1988 Concurrent Computer Corporation — All Rights Reserved

Concurrent Computer Corporation, 106 Apple Street

Tinton Falls, New Jersey 07724

Printed in the United States of America

## NOTICE

THIS MANUAL CONTAINS REFERENCES TO THE EXTENDED MEMORY AND THE MULTIPROCESSOR VERSION OF THE 3280 SYSTEM WHICH WERE NOT AVAILABLE FOR THE OS/32 8.2 SOFTWARE RELEASE.

FOR FURTHER INFORMATION OR ASSISTANCE CONCERNING THESE PRODUCTS, PLEASE CONTACT YOUR LOCAL CONCURRENT COMPUTER CORPORATION SALES REPRESENTATIVE.



## TABLE OF CONTENTS

<b>PREFACE</b>		v	
<b>CHAPTERS</b>			
<b>1</b>	<b>PROGRAMMING IN AN OS/32 ENVIRONMENT</b>		
1.1	OS/32 OPERATIONAL OVERVIEW	1-1	
1.2	OS/32 REAL-TIME ENVIRONMENT	1-3	
1.3	MULTI-TERMINAL MONITOR (MTM) TIME-SHARING ENVIRONMENT	1-4	
1.4	THE OS/32 APPLICATION PROGRAMMER	1-5	
<b>2</b>	<b>TASK STRUCTURE AND EXECUTION CONTROL</b>		
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	2-10	
2.5	MONITOR TASKS AND SUBTASKS	2-12	
2.5.1	The Multi-Terminal Monitor (MTM)	2-13	
2.6	RESTRICTIONS ON INTERTASK COMMUNICATION	2-13	
2.7	ACCESSING OS/32 SYSTEM SERVICES	2-14	
<b>3</b>	<b>INTERRUPT SERVICING IN A REAL-TIME ENVIRONMENT</b>		
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	USER-DEDICATED LOCATION (UDL) AND TASK STATUS WORD (TSW) SWAP	3-9
	3.5	TRAPS HANDLED BY USER-WRITTEN TASKS	3-16
	3.5.1	Arithmetic Fault Trap	3-17
	3.5.2	Data Format/Alignment Faults	3-18
	3.5.3	Power Restoration	3-19
	3.5.4	Illegal Instruction Faults	3-19
	3.5.5	Memory Access Faults	3-19
	3.5.6	Task Queue Trap-Causing Events and Task Queue Service	3-20
	3.5.7	User-Defined Trap-Causing Events	3-27
	3.6	WRITING TASKS THAT HANDLE TASK TRAPS	3-27
	3.6.1	Handling Task Queue Traps	3-31
	3.6.2	Tips for Writing Task Trap-Handling Routines	3-32
	3.6.3	Handling Traps from Trap-Generating Devices	3-33
	3.6.4	Sample Task Trap-Handling Program	3-34
	3.6.5	Using the OS/32 System Macro Library to Handle Traps	3-36
	3.6.6	Writing FORTRAN Trap-Handling Programs	3-36
	3.6.7	Writing Pascal Trap-Handling Programs	3-37
	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	DISK ORGANIZATION	4-6
	4.5	SUPPORTED DISK FILE TYPES	4-7
	4.5.1	Contiguous Files	4-7
	4.5.2	Indexed and Nonbuffered Indexed Files	4-8
	4.5.3	Extendable Contiguous Files	4-8
	4.5.4	Long Record Files	4-9
	4.6	MIRROR DISK ENVIRONMENT	4-9
	4.6.1	Disk Failure	4-10
	4.6.2	Normal Input/Output (I/O) Performance	4-10
	4.7	DISK SPACE MANAGEMENT	4-10
	4.8	VOLUME DESCRIPTOR	4-11
	4.8.1	File Directories	4-12
	4.8.2	Bit Map	4-16
	4.8.3	Permanent and Temporary File Allocation	4-16

## CHAPTERS (Continued)

4.9	ASSIGNING FILES TO A TASK	4-16
4.10	ACCESS METHODS	4-17
4.10.1	Buffered Input/Output (I/O) (Indexed Files)	4-20
4.10.2	Nonbuffered Input/Output (I/O)	4-21
4.10.2.1	Accessing Contiguous Files	4-21
4.10.2.2	Accessing Nonbuffered Indexed Files	4-21
4.10.2.3	Accessing Extendable Contiguous Files	4-22
4.10.2.4	Accessing Long Record Files	4-22
4.11	FILE SECURITY	4-23
4.12	CHOOSING THE RIGHT FILE TYPE	4-26
4.12.1	Using Contiguous Files	4-27
4.12.2	Using Indexed Files	4-27
4.12.3	Using Nonbuffered Indexed Files	4-28
4.12.4	Using Extendable Contiguous Files	4-28
4.12.5	Using Long Record Files	4-29
4.12.6	Disk Fragmentation	4-29
5	WRITING PROGRAMS THAT ACCESS OS/32 SYSTEM SERVICES	
5.1	INTRODUCTION	5-1
5.2	BUILDING A SUPERVISOR CALL (SVC) PARAMETER BLOCK	5-3
5.2.1	Accessing Input/Output (I/O) System Services	5-3
5.2.2	Accessing File Management Services	5-7
5.3	USING THE OS/32 SYSTEM MACRO LIBRARY TO ACCESS SYSTEM SERVICES	5-9
5.4	WRITING A FORTRAN PROGRAM THAT ACCESSES SYSTEM SERVICES	5-10
5.5	WRITING A PASCAL PROGRAM THAT ACCESSES SYSTEM SERVICES	5-11

## 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	2-6
2-5	Task Address Space on a VAT Machine	2-7
2-6	Segmented Task Loaded into Memory	2-9
2-7	Task States	2-12

## FIGURES (Continued)

3-1	Task Status Word	3-2
3-2	User-Dedicated Location	3-10
3-3	TSW Swap	3-12
3-4	Standard Circular List	3-21
3-5	Fullword Task Queue Entry (TQE)	3-21
3-6	Circular List with Task Queue Entries for Subtask State Change	3-24
3-7	TQE for APU Signal	3-25
4-1	Formatted Disk Surface	4-4
4-2	Linked-List Indexed File Organization	4-5
4-3	Volume Descriptor	4-11
4-4	Primary Directory Block	4-13
4-5	Primary Directory Entry	4-13
4-6	Secondary File Directory (SYSTEM.DIR)	4-15
4-7	Task Interfaces to Access Methods	4-18
5-1	Task Interface to OS/32 Executor Routines	5-2
5-2	SVC1 Parameter Block Defined by \$SVC1	5-3
5-3	SVC7 Parameter Block Defined by \$SVC7	5-7
5-4	SVC2 Code 16 Parameter Block	5-8

## TABLES

2-1	TASK WAIT STATES	2-11
3-1	TSW BIT SETTINGS	3-3
3-2	ARITHMETIC FAULT HANDLING	3-7
3-3	UDL FIELDS USED TO HANDLE TASK TRAPS	3-13
3-4	ARITHMETIC FAULT TRAP-CAUSING EVENTS (REASON CODE IN UDL (83('53')))	3-17
3-5	DATA FORMAT/ALIGNMENT FAULT TRAP-CAUSING EVENTS (REASON CODE IN UDL 81('51'))	3-18
3-6	MEMORY ACCESS FAULT TRAP-CAUSING EVENTS	3-20
3-7	TASK QUEUE TRAP-CAUSING EVENTS	3-22
3-8	SUBTASK REASON CODES AND CORRESPONDING STATE CHANGES	3-24
3-9	SUMMARY OF TASK STRUCTURES USED FOR HANDLING TRAPS	3-28
4-1	ACCESS PRIVILEGE COMPATIBILITY	4-24
4-2	READ/WRITE KEYS	4-25
4-3	FILE TYPE SUMMARY	4-26
5-1	SVCL FUNCTION CODES	5-4

## INDEX

IND-1



## PREFACE

This manual describes the facilities available to a programmer implementing an application program in an OS/32 real-time or 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.

The F00 R03 version includes new trap reason codes for the OS/32 PROCOM driver for PENnet and their definitions. |

This revision is intended for use with the OS/32 R08.2 software release or higher. |



# CHAPTER 1 PROGRAMMING IN AN OS/32 ENVIRONMENT

## 1.1 OS/32 OPERATIONAL OVERVIEW

The 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 32-bit processor.

The OS/32 real-time operating environment is an event-driven, priority-based, multitasking environment in 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 uses higher level scheduling algorithms. OS/32 is augmented for time-sharing by the 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.

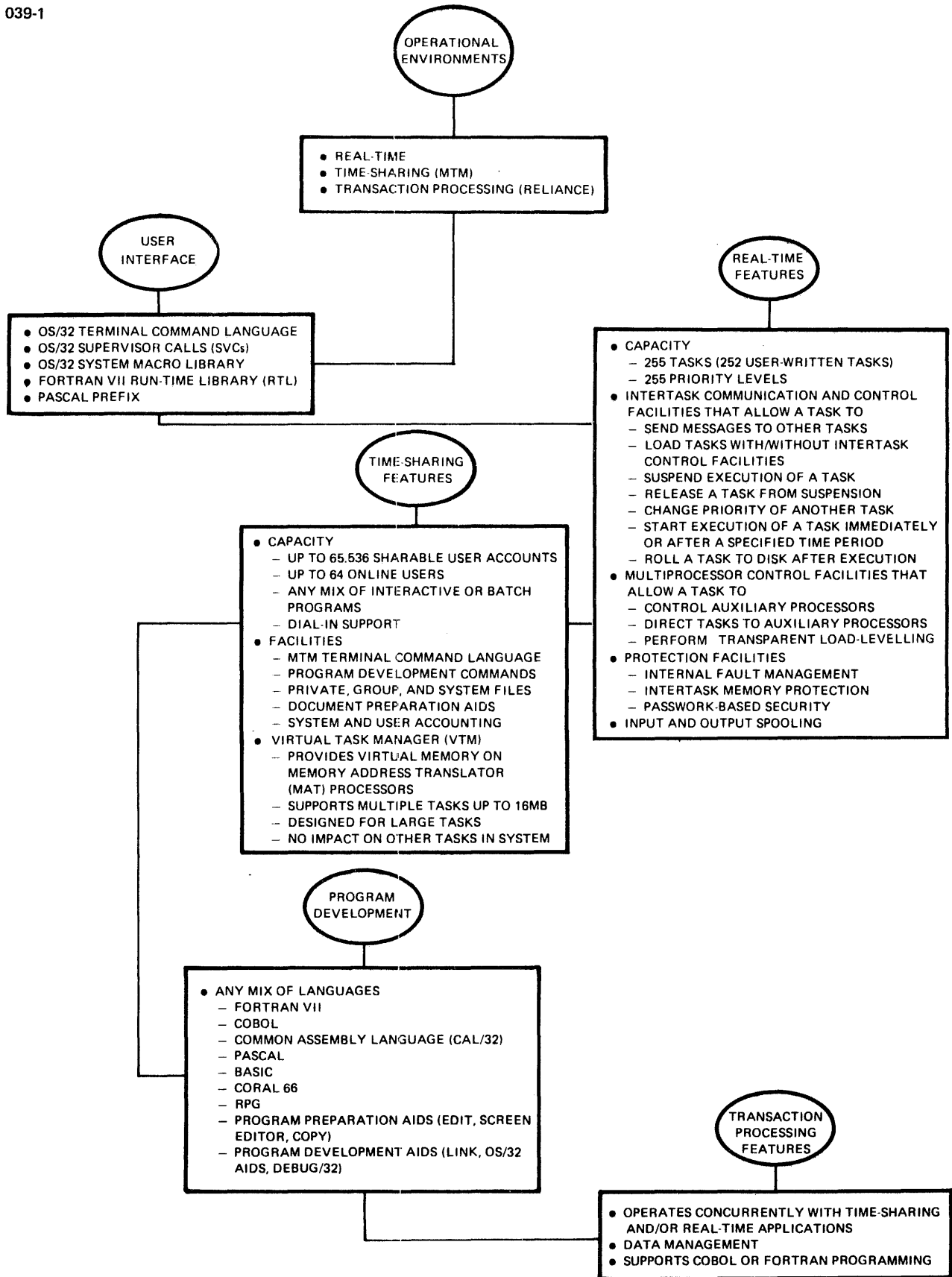


Figure 1-1 Summary of OS/32 Features

## 1.2 OS/32 REAL-TIME ENVIRONMENT

Hardware facilities are enhanced by OS/32 real-time operations. 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 so 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 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 memory addressing capability of a system will have a greater effect 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.

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 by 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 Multi-Terminal Monitor (MTM) Reference Manual for more information on spooling in an OS/32 environment.

OS/32 file management services provide five different file types: indexed, nonbuffered indexed, contiguous, extendable contiguous and long record files. 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 that require a fixed-file length and no buffering overhead can use the contiguous file type. Long record files are designed for applications that require large record lengths. 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 MULTI-TERMINAL MONITOR (MTM) TIME-SHARING ENVIRONMENT

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 Concurrent Computer Corporation 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, and I/O intensive jobs are given higher priority levels and shorter time-slices.

Link, the OS/32 linkage editor, supports the virtual task manager (VTM). VTM provides a 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.

The priority scheduling mechanism (PSM) dynamically alters MTM task priority. PSM increases system throughput by adjusting the priorities of MTM subtasks according to their run-time behavior. PSM performs a periodic check on the run-time behavior of all MTM subtasks. Based on this behavioral information, PSM adjusts the priorities of subtasks whose initial load priorities have been assigned.

For further information on PSM, see the Multi-Terminal Monitor (MTM) System Planning and Operator Reference Manual.

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, RPG, or 'C'. The MTM terminal command language allows users to develop their own command files for compiling, assembling, linking and running a program. See the Multi-Terminal Monitor (MTM) Reference Manual for more information on developing programs in an MTM environment.

To support a diverse community of users, system operators and/or system administrators are provided with Authorized User Utility. This utility allows certain privileges 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 receive those privileges. For information on what privileges can be assigned, see the 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.





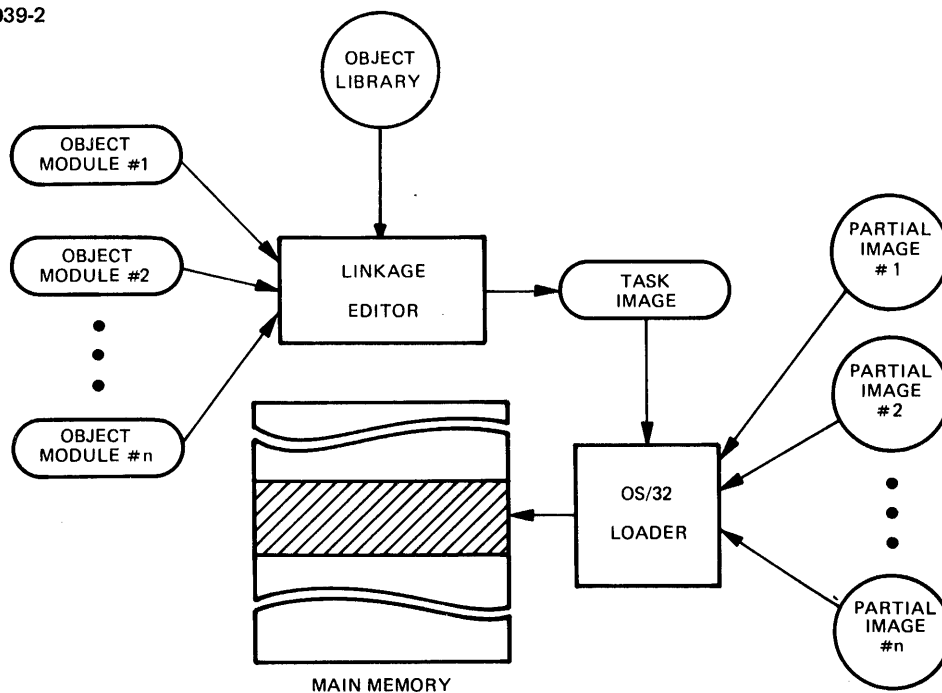
**CHAPTER 2**  
**TASK STRUCTURE AND EXECUTION CONTROL**

**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, in an image file on disk.

039-2



**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 user task (u-task). E-tasks run with memory relocation/protection hardware turned off and are allowed to execute all instructions provided by the 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 System 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 section of 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.

Following the LIB and the history records, if they exist, is the task image that is actually loaded into memory. This task image consists of at least one private image segment. The linkage editor creates the private image with read, write and execute privileges. 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.

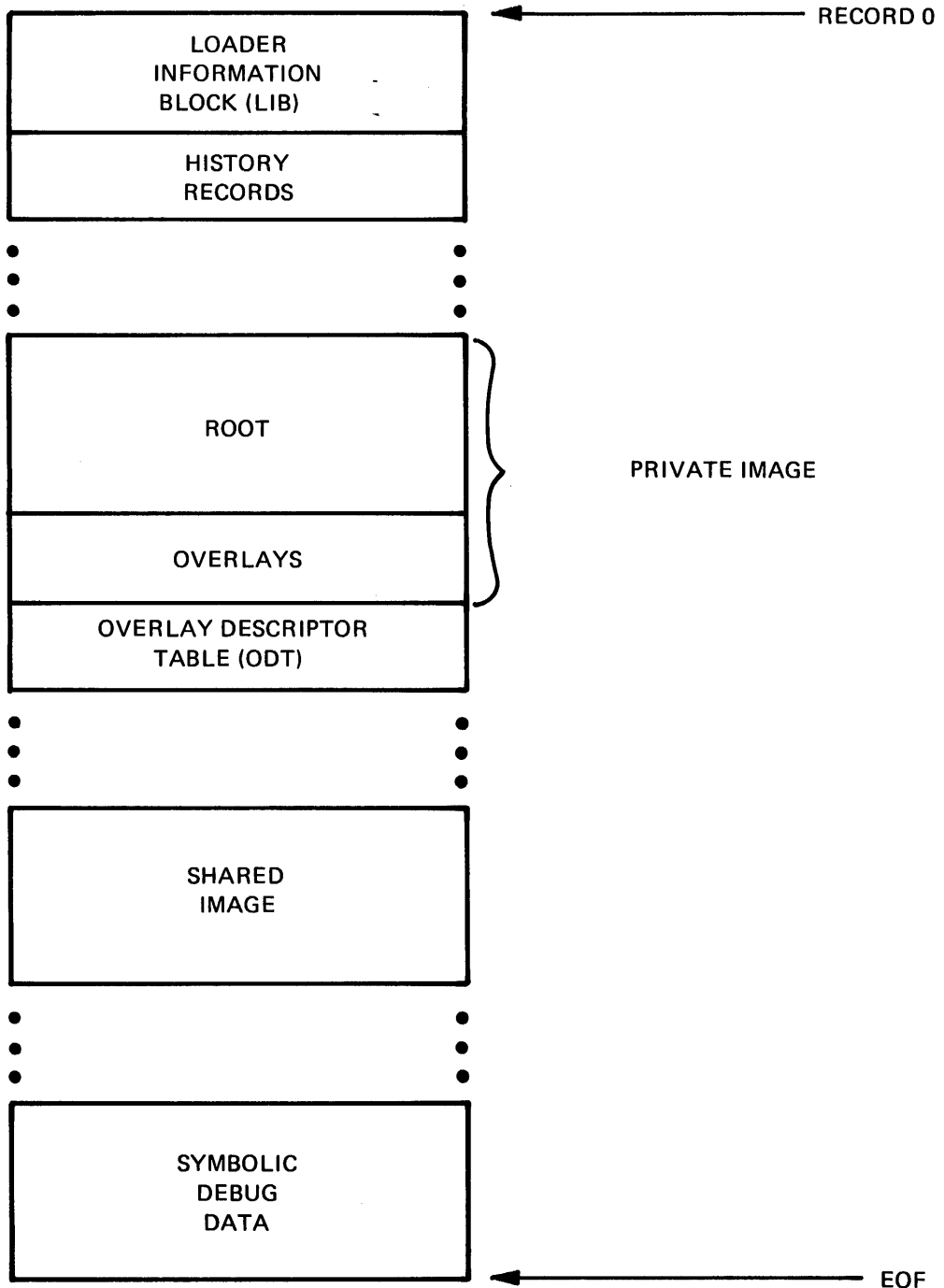


Figure 2-2 Task Image File Format for a Segmented Task

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.

The UDL is used to communicate between the user task and the operating system. Within the UDL is the task status word (TSW). Each task has an active TSW that 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, or 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 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.

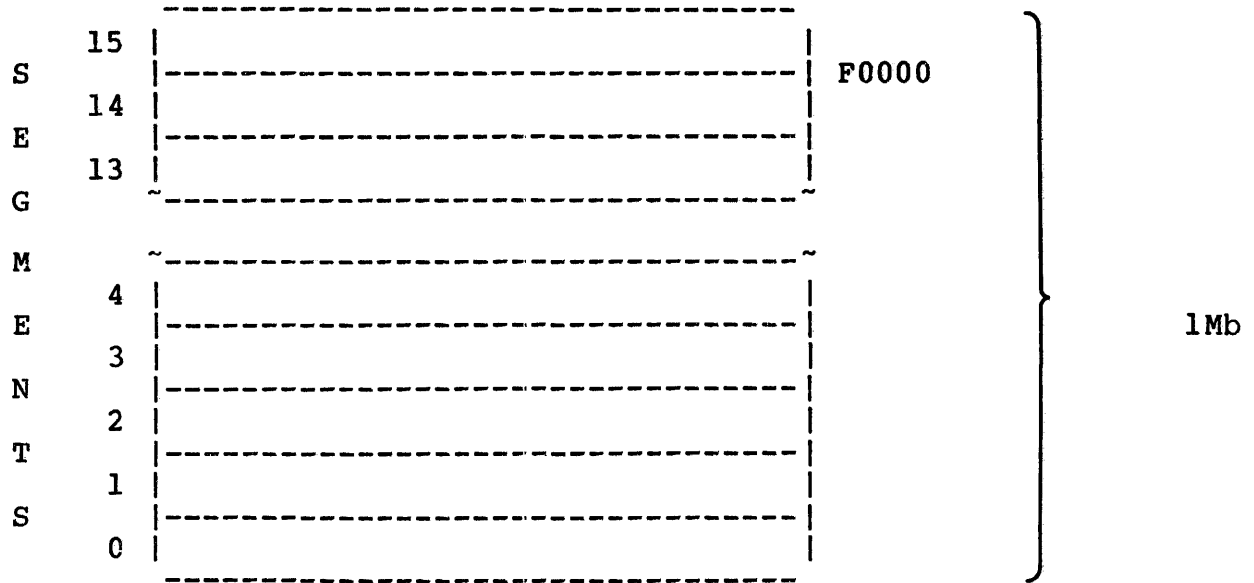
### 2.3 LOADING A USER TASK (U-TASK) INTO MEMORY

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 that might attempt to interfere with its execution. The relocation/protection hardware provides this protection.

In a multitasking system, processors use one of three types of relocation/protection hardware. Models 7/32, 8/32 and 3220 Processors use the memory access controller (MAC); Models 3203, 3205, 3210, 3230(XP), 3240, 3250, 3230MPS, and 3260MPS use the memory address translator (MAT) and the 3280XP/MPS processor uses the virtual address translator (VAT).

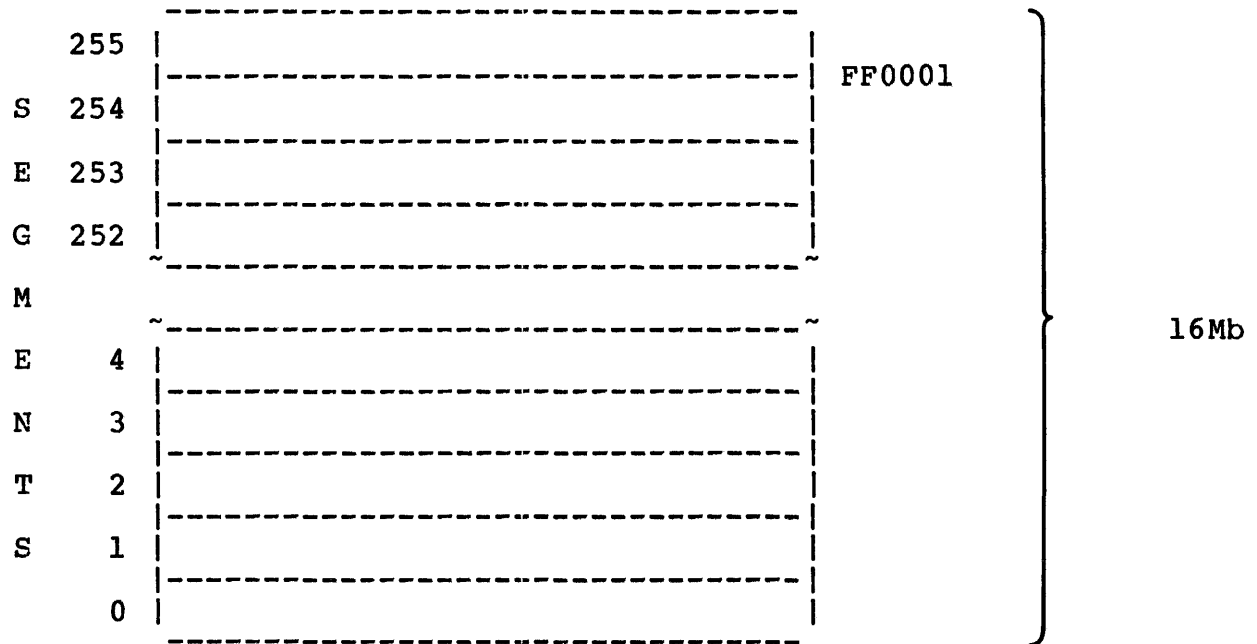
When a u-task is loaded into memory, the relocation/protection hardware automatically allocates the first relative address in the task's root segment to the task's first physical address in memory. To the programmer, the task appears to be loaded at location 0 in memory. Actually, the MAC, MAT or VAT 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 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 1-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.



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

Figure 2-3 Task Address Space on a MAC Machine

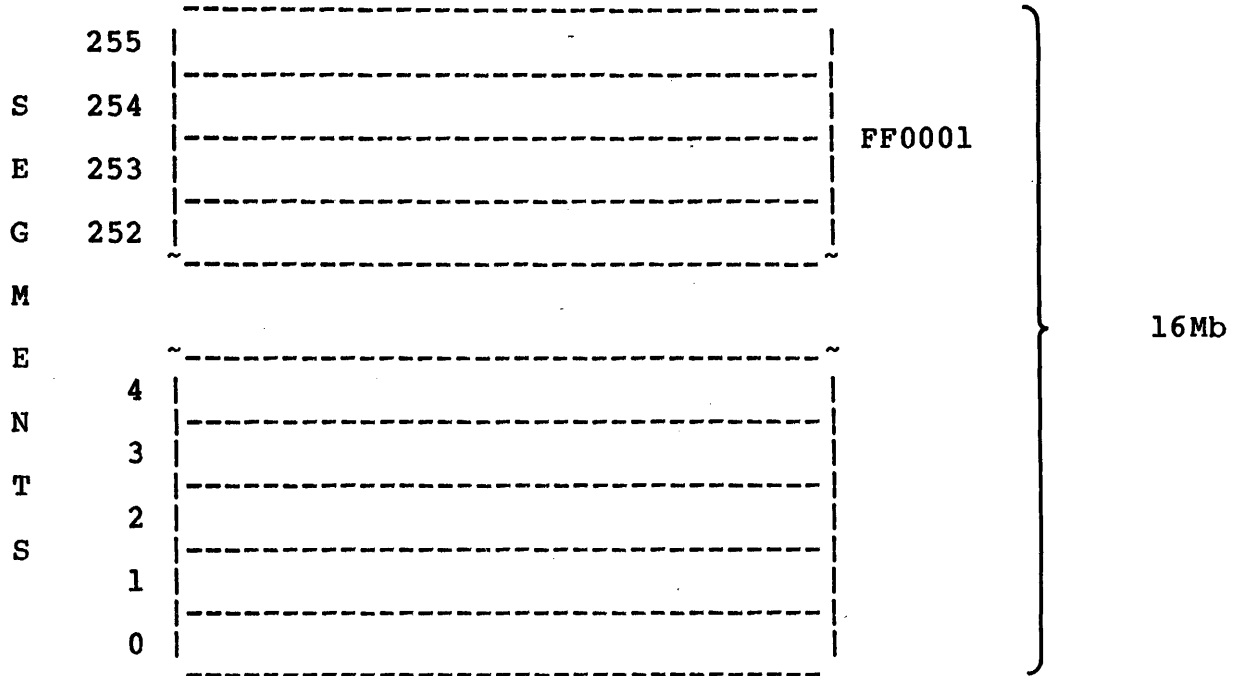


Each MAT segment consists of 32 2,048-byte pages or 64kb.

Figure 2-4 Task Address Space on a MAT Machine

NOTE

Each MAT segment for the Model 3203 and 3205 processors have 16 4,096-byte pages or 64kb.



Each VAT segment consists of 16 4,096-byte pages or 64kb.

Figure 2-5 Task Address Space on a VAT 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. On all MAT machines, except Model 3203 and 3205 Processors, each segment is divided into 32 2,048-byte pages. On Models 3203 and 3205, each segment is divided into 16 4,096-byte pages. MAT locates the first address of each segment of the task on a 2,048-byte or 4,096-byte boundary.

Figure 2-5 illustrates how the VAT describes the u-task address space. Each VAT segment consists of 16 4,096-byte pages or 64kb. The VAT locates the first address of each segment of the task on a 4,096-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 discontinuous) areas, private and shared, as shown in Figure 2-6. Every task has a private area. This area contains the private image code (UDL, root, 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.

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 the nominal space and the maximum work space. UTOP is a pointer to the address of the first byte of the task workspace. UTOP plus the nominal workspace is equivalent to CTOP, the pointer to the address of the last addressable halfword in the task workspace. CTOP must be equal to or less than the maximum workspace parameter of the OPTION WORK command.

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.



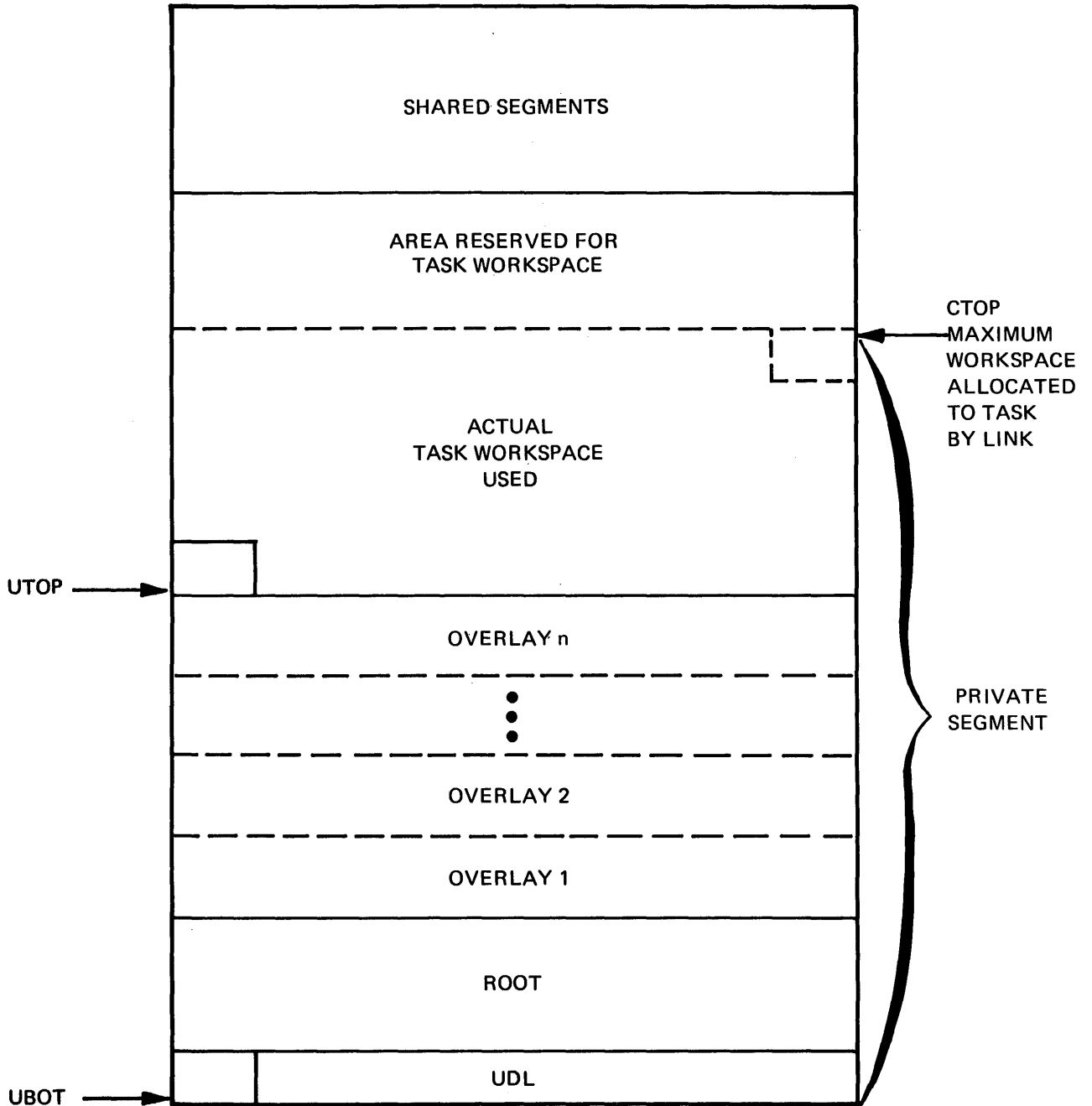


Figure 2-6 Segmented Task Loaded into Memory

## 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 to 255, with priority 1 being the most favored. 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 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. 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.

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 by 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., if 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 2-7.

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 intercepted 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.

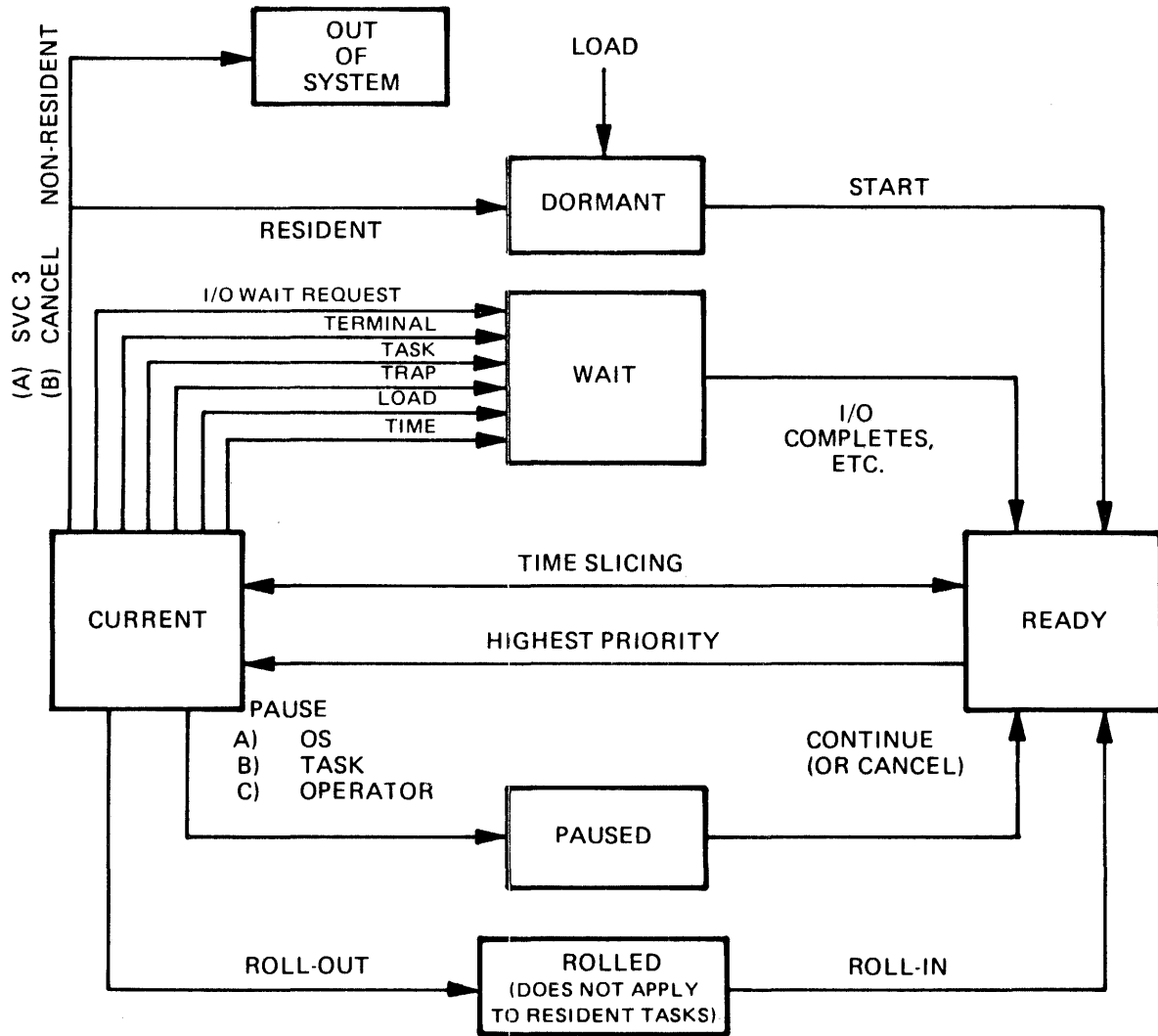


Figure 2-7 Task States

## 2.5 MONITOR TASKS AND SUBTASKS

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 specifying the subtask reporting option when loading another task. The number of subtasks that can be assigned to a single monitor is unlimited (within the 252 user-written tasks supported).

The subtask reporting option causes the operating system to keep the monitor informed of the status of its subtasks; i.e., 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 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; i.e., if MTM's priority is 140, the task's highest priority could not exceed 141.

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 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 (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 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 was invalidated when execution of an instruction caused a floating point overflow condition, the programmer would want to know when such an event occurred. Otherwise, the programmer could not be certain that the results of a program were valid. The mechanism that informs the task when such an event occurs 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. For example, the user may create a trap-handling routine so that when an event such as a floating point overflow occurs, the trap-handling routine places asterisks in the output of a variable and outputs some meaningful message. The trap routine then causes the original task to resume execution at the instruction following the one that caused the suspended trap execution.

Task execution can be resumed only if the state of the task was saved by the operating system prior to executing the trap service routine. 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.

039-6

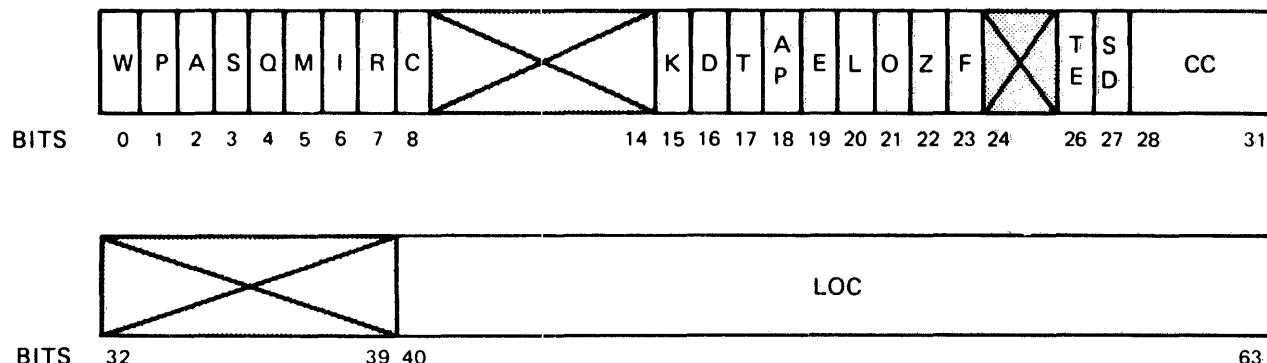


Figure 3-1 Task Status Word

As shown in Figure 3-1, the TSW occupies eight bytes (64 bits) aligned on doubleword 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 31. The program LOC is contained in the area following bit 39. All TSW bit settings and their corresponding effects on task execution are listed in Table 3-1.

When a task is link-edited, using OS/32 Link, its TSW is initialized. The default TSW has all task traps disabled and the task's starting address in the LOC field. The user can change the TSW trap and LOC 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 an operating system data structure called the task control block (TCB). When the task is loaded into memory. If a task trap bit is disabled, that trap-causing event will be handled by the default OS/32 trap-handling routine. See Section 3.3.

If the trap bit for one of the internal fault conditions is enabled, execution control is transferred to the user-written trap-handling routine, as described in Section 3.5.



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 SVC14 Trap	Notifies task when an SVC14 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 (C)	TSW.CPOM	Central Processing Unit Only	Prevents scheduling task to auxiliary processing unit execution queue. Applies to the 3200MPS Family of Processors only.
9-14	-	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.

TABLE 3-1 TSW BIT SETTINGS (Continued)

BIT POSITION	MASK NAME	BIT NAME	EFFECT ON TASK
16 (D)	TSW.DIQM	Queue Entry on Device Interrupt	Notifies task of a device interrupt by adding a fullword entry to the task queue.
17 (T)	TSW.TCM	Queue Entry on Task Call	Notifies task of an SVC6 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 Input/Output (I/O) Completion	Notifies task that an SVC1 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 Communications Functions	Notifies task that an SVC15 operation has been completed by adding a fullword entry to the task queue.
		Queue Entry on SVC1 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.

TABLE 3-1 TSW BIT SETTINGS (Continued)

BIT POSITION	MASK NAME	BIT NAME	EFFECT ON TASK
24-25	-	Reserved	-
26 (TE)	TSW.TESM	Enable Task Queue Event Service	Notifies task of an event through a task event trap. See the 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	-
32-39	-	Reserved	-
40-63 (LOC)	TSW.LOC	LOC	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 3-4)
- Memory access faults (see Table 3-6)
- Illegal instruction faults
- Data format/alignment faults (see Table 3-5)

When a fault is detected by the processor, the currently executing task is checked for trap fault-handling facilities. If none exist (TSW trap bits are zero), the task is suspended and control is given to the appropriate default system trap-handling routine.

In the case of power restoration, the default OS/32 trap-handling routine causes suspension of 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 default OS/32 trap-handling routine identifies the fault and outputs a message to the console. Task execution is then suspended.

All fault-caused traps are handled in the above manner, except arithmetic faults. Arithmetic faults are handled based on the following three possible conditions:

- OS/32 task arithmetic fault pause option (AFPAUSE)
- TSW arithmetic fault bit (bit 2)
- Hardware floating point underflow interrupt bit (bit 19) of the program status word (PSW) (the AF bit on Model 7/32 and 8/32 Systems)

The task option for arithmetic fault handling is set at Link time via the Link command:

OPTION	AFPAUSE
	NAFPAUSE

or at run-time via the operating system command:

OPTION	AFPAUSE
	AFCONT

The TSW fault bit is set as described in Section 3.6, and a PSW bit is set or reset by SVC2 code 4. See the OS/32 Supervisor Call (SVC) Reference Manual. The results are described in Table 3-2.

TABLE 3-2 ARITHMETIC FAULT HANDLING

HARDWARE PSW BIT 19	LINK/OS TASK OPTION	TSW BIT 2	ACTION
0	X	X	For Model 7/32 and 8/32 processors, all AF fault recognition disabled; task continues; no operating system message.  For the 3200MPS Family of Processors, floating-point underflow recognition disabled; task continues; no operating system message.
1	Continue	0	Operating system message sent to console; task continues.
1	Pause	0	Operating system message sent to console; task is paused.
1	Continue	1	No operating system message sent to console; task trapped to AF handler.
1	Pause	1	Operating system message sent to console; task is paused.

1 = Interrupt enabled  
 0 = Interrupt disabled  
 X = Don't-care

The following are the default trap-handling messages sent to the system console or multi-terminal monitor (MTM) terminals.

**Error Messages:**

taskid: ACCESS PRIVILEGE ADDRESS ERROR AT RRxxxx (yyyyyy)

The user program tried to perform a function (execute, store or load) that is prevented by the access privileges requested at Link time for the segment. The most common cause of this error is an attempt to store data into a pure, nonwritable segment. Program address is RRxxxx; segmentation number register is RR; physical address is YYYYYY.

taskid: ALIGNMENT FAULT INSTRUCTION AT xxxxxx (yyyyyy)  
MEMORY FAULT ADDRESS=xxxxxx (yyyyyy)

Data instruction is not properly aligned to specific fields for fullword or halfword alignment. The memory fault address is the memory location that is not properly aligned. The memory fault address is given only on the 3200MPS Family of Processors. Program address is xxxxxx; physical address is yyyyyy.

taskid: ARITHMETIC FAULT AT xxxxx (yyyyyy)

Arithmetic fault is detected at location xxxxx in the taskid address space; physical address is yyyyyy.

taskid: ILLEGAL INSTRUCTION AT xxxxx (yyyyyy).

Illegal instruction fault detected at location xxxxx in the taskid address space; physical address is yyyyyy.

taskid: INVALID SEGMENT ADDRESS ERROR AT xxxxx (yyyyy)

The task tried to address a segment outside the address space of the program. Program address is xxxxx; physical address is yyyyyy.

taskid: MEMORY PARITY ERROR AT xxxxx (yyyyyy)

Parity or an error check and correction (ECC) machine malfunction (MMF) is detected at location xxxxx; physical address is yyyyyy.

taskid: SEGMENT LIMIT ADDRESS ERROR AT RRxxxx (yyyyyy)

The task attempted to access an address outside allowable limits for one of its segments. Program address is RRxxxx; segmentation register is RR; physical address is YYYYYY.

taskid: TASK PAUSED

Task taskid paused; results from a SVC2 code 1, operator PAUSE command or power fail recovery via the OS/32 default.

POWER RESTORE - RESET PERIPHERALS

Power fail restore sequence; no operator response required.

## POWER RESTORE - RESET PERIPHERALS AND ENTER GO

Power fail restore sequence; perform any manual intervention required at the peripheral device(s), then type GO (CR) to complete power recovery.

### NOTE

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 with the user-written trap-handling routine. See Section 3.5 or the OS/32 Link Reference Manual for more information.

### 3.4 USER-DEDICATED LOCATION (UDL) AND 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. The UDL is an area occupying locations 0 through 255 (X'FF') in each task impure memory space preceding the task code. Expansion of the \$UDL macro generates the structures and equates for the UDL data structure. Figure 3-2 illustrates the UDL structure. The UDL fields used to handle traps are described in Table 3-3.

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)	RESERVED			
184 (B8)	POINTER TO SYSTEM NETWORK ARCHITECTURE TABLE (UDL.SNA)			
188 (BC)	SAVE AREA USED BY SYSTEM NETWORK ARCHITECTURE(UDL.RSAV)			
192 (C0)				
196 (C4)				
200 (C8)	RESERVED FOR AIDS			
204 (CC)	OR DEBUG/32			
208 (D0)				
212 (D4)				
216 (D8)				
220 (DC)				
224 (E0)				
228 (E4)				
232 (E8)				
236 (EC)				
240 (F0)				
244 (F4)				
248 (F8)				
252 (FC)				

Figure 3-2 User-Dedicated Location



Note that several fields in the UDL 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.

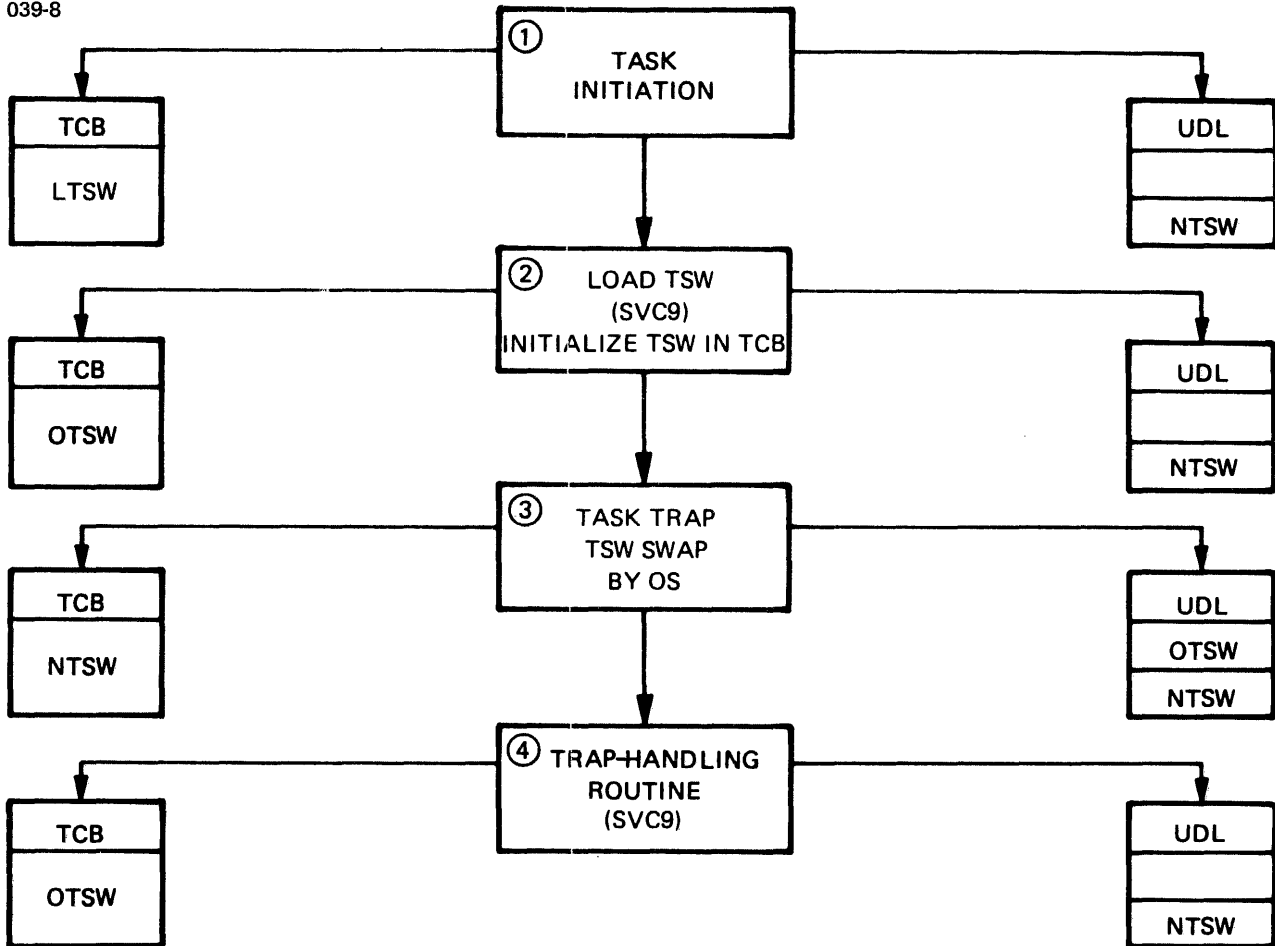
Unless the user specifically initializes the UDL, all locations in the UDL and the initial TSW are set to zero by OS/32 Link. Thus, for all tasks, the default is operating system handled traps.

The users, however, can initialize the UDL by assembling the appropriate code and using the Link OPTION ABSOLUTE=0 command before including the UDL in their task or by initializing the fields at run-time (e.g., by calling the appropriate FORTRAN routines as described in Section 3.6).

Figure 3-3 shows the effective TSW swaps to and from the TCB and the UDL. The following notes correspond to the numbers in Figure 3-3.

#### NOTES

1. In Figure 3-3, the original TSW was allowed to default to zero at Link time.
2. After the task is started, the TSW can be initialized or modified via an SVC9.
3. TSW swap is performed when the task trap occurs.
4. At the end of a trap-handling routine, the user can return to the original interrupted code sequence.



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

Figure 3-3 TSW Swap

To perform the TSW swap, the task and the trap-handling routine issues a call (SVC9) to an OS/32 supervisor routine that replaces the current TSW in the TCB with the TSW specified as an argument to SVC9 (see number 2 of Figure 3-3).

As shown in Figure 3-3, 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 SVC9 instruction.

The TSW swap performed by the OS/32 trap mechanism, in response to an event, replaces the OTSW in the TCB with a copy of the NTSW stored in the UDL (see number 3 of Figure 3-3). 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. See number 4 of Figure 3-3.

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; i.e., 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 3-3.

TABLE 3-3 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 SVC6 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(SVC14 Arg)	UDL.SV14	Effective address of the argument to an SVC14
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 3200MPS Family of Processors, the LOC of this TSW points to the faulting instruction. For Model 7/32 and 8/32 Processors, LOCs point to the instruction after the faulting instruction.

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

BYTE LOCATION	FIELD NAME	MASK NAME	CONTENTS OF FIELD
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 3-7)
82('52')	Relocation/Protection Hardware Fault Reason Code	UDL.MAFR	Reason code indicating the event that caused the memory access fault (see Table 3-6)  This reason code is given only for a memory access controller (MAC) memory address translator (MAT) or virtual address translator (VAT) fault occurring on the 3200MPS Family of Processors.
83('53')	Arithmetic Fault Reason Code	UDL.ARFR	Reason code indicating the event that caused the arithmetic fault (see Table 3-5)  This reason code is given only for an arithmetic fault occurring on the 3200MPS Family of Processors.
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 the 3200MPS Family of Processors.
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 the 3200MPS Family of Processors.
92('5C')	Relocation/Protection Hardware 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 or VAT (for the 3280XP/MPS only) fault occurring on the 3200MPS Family of Processors.

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

BYTE LOCATION	FIELD NAME	MASK NAME	CONTENTS OF FIELD
96('60')	SVC14 Old TSW	UDL.SI40	TSW saved by OS/32 when an SVC14 trap occurs
104('68')	SVC14 New TSW	UDL.SI4N	TSW containing the address of the SVC14 trap-handling routine to which execution branches when an SVC14 trap occurs
112('70')	Task Queue Service Old TSW	UDL.TSKO	TSW saved by OS/32 when a service trap occurs
120('78')	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 the 3200MPS Family of Processors.
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

### 3.5 TRAPS HANDLED BY USER-WRITTEN TASKS

The purpose of this section is to describe and summarize the following task-handled traps:

- Arithmetic faults
- Data format/alignment faults
- Power restoration
- Illegal instruction faults
- Memory access faults
- Task queue service events
- User-defined, trap-causing events

The traps described below are divided into three categories based upon the causing event: hardware-based events in category 1, asynchronous software events in category 2, and specifically called (synchronous) software events in category 3.

Before a task can handle a trap, the task must have a TSW in which the appropriate trap bits are set. If the trap bit in the TSW is enabled, execution control is transferred to the user-written trap-handling routine when that fault condition occurs. (If the trap bit is disabled, execution control is transferred to the default OS/32 trap-handling routine as previously explained in Section 3.3.)

Category 1 events (hardware) include arithmetic, data format/alignment, power restoration, illegal instruction and memory access faults. These events are recognized by the operating system, which then passes control by performing a TSW swap. The TSW is removed from the TCB at the time of the fault and is saved in the UDL at an old TSW save area. The location portion of the old TSW is updated to point to the next executable instruction.

After successfully saving the old TSW, the operating system fetches a new TSW from the user's UDL. The location specified by the new TSW is the entry point to the appropriate trap service routine. The effect of this is a forced branch into a subroutine that handles the special conditions represented by the fault. These traps are more fully described in Sections 3.5.1 through 3.5.5.

The second category consists of asynchronous software events, known as task queue service events. Unlike the specific events described above, software-based traps are handled by a single trap service routine, referred to as the task queue service routine, and a data structure referred to as a task queue.

Task-handled traps use the data structures and procedures previously discussed. The task queue is added as an information transfer mechanism. Task queue service is fully described in Section 3.5.6.

The third category of task-handled traps are user-defined traps. User-defined traps allow the assembly programmer to insert in the program coding specific SVC14 calls to a trap routine. When the SVC14 is encountered, the user is trapped to the SVC14 trap handler exactly as described in the category 1 traps. This mechanism is frequently used as a debugging aid in debugging packages. User-defined traps are more fully described in Section 3.5.7.

Section 3.6 provides examples of programming techniques and tips for effective use of task-handled traps in various programming languages.

### 3.5.1 Arithmetic Fault Trap

An arithmetic fault trap can result from any one of the events listed in Table 3-4.

TABLE 3-4 ARITHMETIC FAULT TRAP-CAUSING EVENTS  
(REASON CODE IN UDL 83('53'))

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'
Decimal underflow error	X'05'
Decimal overflow error	X'06'
Floating point function range error (for the 3280XP/MPS only)	X'07'

When an arithmetic fault occurs with the TSW.AFM bit set in the TSW, the current TSW is stored in the UDL.ARFO field, and the new TSW in the UDL.ARFN field is loaded and becomes the current TSW. The reason code is stored in the UDL.ARFF field. The LOC of the new TSW contains the address of the arithmetic fault trap service routine. The action taken when an arithmetic fault trap occurs depends on the options specified by Link and the traps enabled in both the TSW and PSW.

### 3.5.2 Data Format/Alignment Faults

A data format or alignment fault trap results when one of the events listed in Table 3-5 occurs.

TABLE 3-5 DATA FORMAT/ALIGNMENT FAULT TRAP-CAUSING EVENTS (REASON CODE IN UDL 81('51'))

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'

When a data format or alignment fault trap occurs with the TSW.DFFM bit set, the current TSW is stored in the UDL.DFFO field; the NTSW in the UDL.DFFN field is loaded and becomes the current TSW; the address of the location in memory referenced by the faulting instruction is stored in the UDL.DFFX field; and the reason code is stored in the UDL.DFFR field. The new TSW LOC contains the address of the data format or alignment fault trap service routine. This trap service routine exits by issuing an LTSW macro or SVC9 call to load the TSW stored in the UDL.DFFO field as the current TSW.



### 3.5.3 Power Restoration

A power restoration trap occurs after power is restored following a power failure and the TSW.PWRM bit in the TSW is set. The current TSW is stored in the UDL.PWRO field, and the new TSW in the UDL.PWRN field is loaded and becomes the current TSW. The LOC of the new TSW should contain the address of the power restoration trap service routine. This trap service routine exits by issuing an LTSW macro or SVC9 call to load the TSW stored in the UDL.PWRO field as the current TSW.

Upon restoration of power, OS/32 checks each TCB to determine the state of each task at the time of the power failure. If a task was dormant or in the process of cancellation, its state is not changed. For active tasks with power fail traps enabled, the trap is taken. The action taken for active tasks with power fail traps disabled depends upon the option (SGN.PWF) chosen at sysgen. If automatic power fail recovery was specified (SGN.PWF=0), the task state is unchanged. If operator intervention upon recovery was specified (SGN.PWF=1), the task is paused.

OS/32 also checks each device control block (DCB). If a device has a power fail recovery routine, the routine is scheduled. If a device was connected when the failure occurred, the I/O is timed out.

Prior to servicing of any tasks or driver scheduled events, the power restore leaf is added to the system queue. This leaf triggers the power restore event service routine (ESR) which resets the trap wait bit for each task. All timers are cancelled; if a task was in an interval or time-of-day wait state, the wait state is removed. Removing the wait state allows tasks to be replaced on the appropriate queue (ready or roll in) and also allows any tasks that may have been in a timer wait state to be dispatched so that any power restore trap or pause pending can be serviced.

### 3.5.4 Illegal Instruction Faults

An illegal instruction trap occurs after a user task (u-task) executes an illegal instruction with the TSW.IITM bit set. The current TSW is stored in the UDL.IITO field, and the new TSW in the UDL.IITN field is loaded and becomes the current TSW. The new TSW LOC should contain the address of the illegal instruction trap service routine. This trap service routine exits by issuing an LTSW macro call to load the TSW stored in the UDL.IITO field as the current TSW.

### 3.5.5 Memory Access Faults

A memory access fault trap occurs when one of the events listed in Table 3-6 occurs.

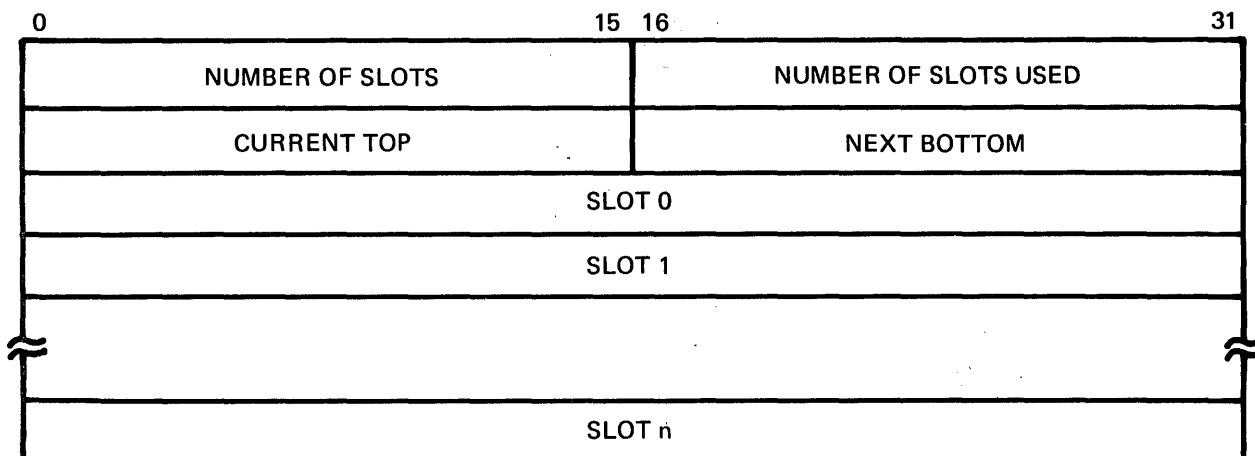
TABLE 3-6 MEMORY ACCESS FAULT TRAP-CAUSING EVENTS

EVENT	REASON CODES	
	3220	MAT
SVC address error	X'00'	N/A
Execute protect violation	X'01'	X'01'
Write/interrupt protect violation	X'02'	X'02'
Read protect violation	N/A	X'03'
Access level fault	N/A	X'04'
Segment limit fault	X'10'	X'05'
Nonpresent segment fault	N/A	X'06'
Shared segment table (SST) size exceeded	N/A	X'07'
Private segment table (PST) size exceeded	X'08'	'X08'

When a memory access fault occurs with the TSW.MAFM bit set, the current TSW is stored in the UDL.MAFO field; the new TSW in the UDL.MAFN field is loaded and becomes the current TSW; and the faulting instruction address is stored in the UDL.MAFL field. The new TSW LOC should contain the address of the memory access fault trap service routine. This trap service routine exits by issuing an LTSW macro or SVC9 call to load the TSW stored in the UDL.MAFO field as the current TSW.

### 3.5.6 Task Queue Trap-Causing Events and Task Queue Service

A task queue is in the form of a standard circular list, as shown in Figure 3-4.

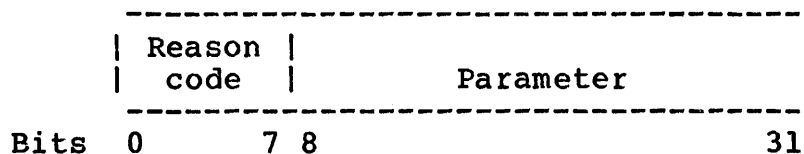


**Figure 3-4 Standard Circular List**

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. For further information on list processing, see the Instruction Set Reference Manual for any 32-bit processor.

When a software fault trap occurs, entries are added to the bottom of the list. The user-supplied task queue service routine should always remove entries from the top of the queue.

Except for the subtask change event and the APU signaling the CPU, all items added to the task queue, by OS/32, are four bytes long and have the following format.



**Figure 3-5 Fullword Task Queue Entry (TQE)**

Reason codes and parameter information can be found in Table 3-7. Table 3-7 lists the fullword entries added to the list by the task queue trap-causing events.

TABLE 3-7 TASK QUEUE TRAP-CAUSING EVENTS

EVENT	TASK QUEUE FULLWORD ENTRY	
	1-BYTE REASON CODE	CONTENTS OF 3-BYTE PARAMETER
Device interrupt (SVC6)	X'00'	Associated with trap-generating device
SVC6 queue parameter	X'01'	Specified by the task that issued the SVC6
Subtask state change	X'02'	See Figure 3-6.
System Performance Monitor (SPM)	X'03'	Reserved
SVC6 send data	X'04'	A(send data message buffer)
APU signals CPU	X'05'	APU number, status, and error code (see Figure 3-7)
Message received (SVC6)	X'06'	A(message buffer in ring)
Load proceed completion (SVC6)	X'07'	A(SVC6 parameter block)
I/O proceed completion (SVC6)	X'08'	A(SVC1 parameter block)
Timer completion (SVC2, 23)	X'09'	Time interval specified by the SVC2 code 23 parameter block
SVC15 command completion	X'0A'	A(SVC15 parameter block)
SVC15 buffer completion	X'0B'	A(SVC15 parameter block)
SVC1 gapless buffer completion	X'0B'	A(SVC1 parameter block)
SVC1 buffer transfer completion	X'0B'	A(SVC1 parameter block)

TABLE 3-7 TASK QUEUE TRAP-CAUSING EVENTS (Continued)

EVENT	TASK QUEUE FULLWORD ENTRY	
	1-BYTE REASON CODE	CONTENTS OF 3-BYTE PARAMETER
SVC15 termination	X'0C'	A(SVC15 parameter block)
SVC15 halt I/O	X'0D'	A(SVC15 parameter block)
ZBID data link control (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 time-out	X'1A'	-
OS/32 PROCOM driver for PENnet	X'1C'	Transmit of a PENnet message/function complete
OS/32 PROCOM driver for PENnet	X'1D'	Received PENnet function from PROCOM
OS/32 PROCOM driver for PENnet	X'1E'	Received PENnet message from PROCOM
OS/32 PROCOM driver for PENnet	X'1F'	Received PENnet message with one associated data buffer from PROCOM
OS/32 PROCOM driver for PENnet	X'20'	Received PENnet message with two associated data buffers from PROCOM

TABLE 3-7 TASK QUEUE TRAP-CAUSING EVENTS (Continued)

EVENT	TASK QUEUE FULLWORD ENTRY	
	1-BYTE REASON CODE	CONTENTS OF 3-BYTE PARAMETER
OS/32 PROCOM driver for PENnet	X'21'	No resources available to receive PENnet function or message transmitted from PROCOM
OS/32 PROCOM driver for PENnet	X'22'	Protocol error detected
OS/32 PROCOM driver for PENnet	X'23'	Reserved

\* The letter A indicates that the 3-byte Q entry is an address.

NOTE

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

Task queue entries that are added to a monitor's task queue when its subtask experiences a state change are shown in Figure 3-6. Note that a subtask state change adds three 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 3-8) and other subtask information items. The remaining fullword slots contain the name of the subtask.

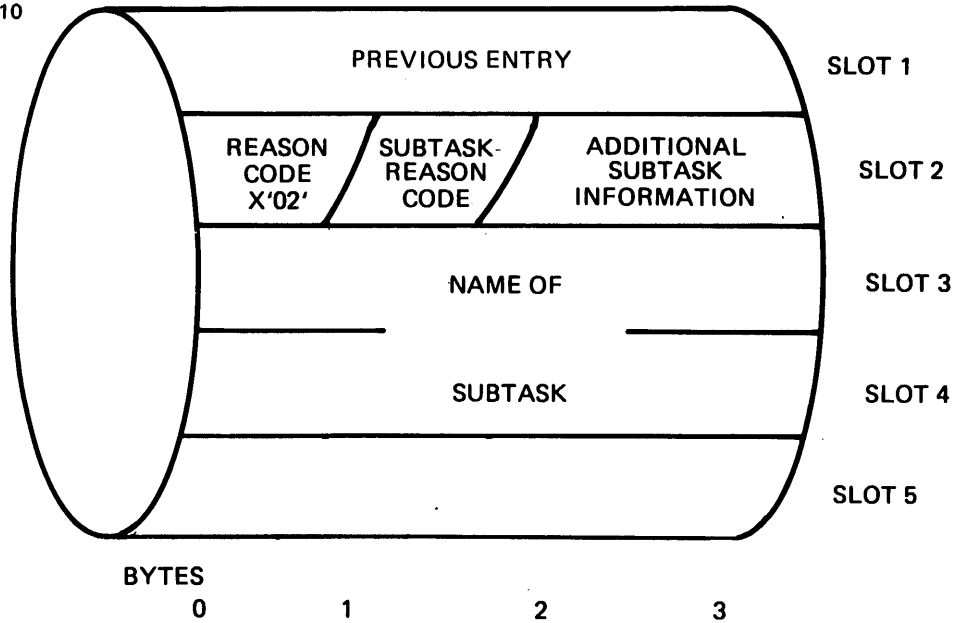


Figure 3-6 Circular List with Task Queue Entries for Subtask State Change

TABLE 3-8 SUBTASK REASON CODES AND CORRESPONDING STATE CHANGES

SUBTASK REASON CODE	SUBTASK STATE CHANGE
X'00'	End of task; bytes 2 and 3 are binary end of task codes
X'01'	Paused
X'02'	Continued
X'03'	Suspended
X'04'	Released
X'05'	Rolled out
X'06'	Rolled in
X'07'	Started by a task other than the monitor
X'08'	Accounting overflow (MTM or AFDCB only)

In the 3200MPS Family of Processors, a task can receive a trap in response to the APU signals that indicate APU state changes or errors. Section 3.6.3 describes the SVC6 CONNECT and THAW functions that associate APUs with the task. Figure 3-7 shows the format of the task queue entry (TQE) for an APU signal.

039-11

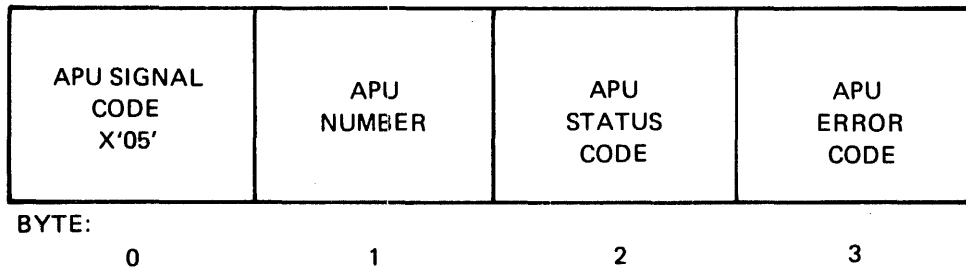


Figure 3-7 Task Queue Entry (TQE) for APU Signal

**Fields:**

- APU Signal Code is X'05'.
- APU Number is a 1-byte field specifying a decimal number (1 through 9) indicating the number of the APU from which the signal was received.
- APU Status Code is a 1-byte field whose contents correspond to the response byte of the SVC13 APU hardware status field.
- APU Error Code is a 1-byte field whose contents correspond to the error byte of the SVC13 APU hardware status field. An X'80' code indicates that no errors exist.

**NOTE**

For information concerning the SVC13 APU hardware status field, consult the OS/32 Supervisor Call (SVC) Reference Manual.



An APU signal entry can be made to the task queue in order to:

- Indicate that the APU is entering the queue wait state. For example, the APU is waiting for the CPU to place a TCB on the empty APU queue; no errors exist. In this instance, the APU status code is set to X'C1' and the APU error code is set to X'80'.
- Indicate that the APU has returned a TCB to the APU queue; no errors exist. In this instance, the APU status code is set to X'C2' and the APU error code is set to X'80'.
- Indicate that the APU has placed a TCB on the CPU receive queue; no errors exist. In this instance, the APU status code is set to X'43' and the error code is set to X'80'.
- Indicate that the APU has detected an error in system data structures at an arbitrary moment. In this instance, the APU status code is set to X'C4' and the error code indicates the particular APU error condition.
- Indicate that the APU has detected an error in system data structures during the queue wait state. In this instance, the APU status code is set to X'45' and the error code indicates the particular APU error condition.
- Indicate that the APU has detected an error in system data structures during the queue lock state. In this instance, the APU status code is set to X'46' and the error code indicates the particular APU error condition.

### 3.5.7 User-Defined Trap-Causing Events

The OS/32 supervisor routine called by SVC14 allows the programmer to define trap-causing events for a task. SVC14 suspends task execution, saves the current state of the task, and transfers execution to the SVC14 task trap-handling routine.

One argument can be specified when SVC14 is called. This argument can point to a memory location that contains a reason code for the user-defined trap-causing events. See the OS/32 Supervisor Call (SVC) Reference Manual for more information on using SVC14.

### 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 structure. To prepare a task to handle a trap, simply execute these instructions and set the appropriate fields or bit masks for the trap that 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:**

```

                MLIBS 8                FETCHES SYSSTRUC.MLB
                NLSTM                   SUPPRESSES MACRO LISTING
                $UDL                     MACRO TO DEFINE UDL STRUC
MYUDL           ABS 0
MYUDLE          DS  UDL
                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

```

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

**Example:**

```

                NLSTM                   SUPPRESSES MACRO LISTING
                $TSW                     MACRO TO EXPAND TSW EQUATES
NTSW           DC   TSW.IITM            SETS ILLEGAL INSTRUC TRAP BIT
                DC   0                   SETS LOC AT INSTRUC FOLLOWING SVC9

```

After initiation is completed, the task will execute the "TRAP" label and will then issue an SVC9 to load this TSW into the TCB as follows:

```

TRAP          SVC  2,MSG
              SVC  2,PAUSE
              SVC  9,NTSW

```

Table 3-9 summarizes the UDL fields and TSW bit masks that pertain to each type of task trap.

TABLE 3-9 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.ARFN* UDL.ARFN*	No	TSW.AFM
Data Communications  SVC15 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 (SVC6)  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 (SVC1)	UDL.TSKO UDL.TSKN UDL.TSKQ	Yes	TSW.TSKM TSW.IOM
Load and Proceed Completion (SVC1)	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 3-9 SUMMARY OF TASK STRUCTURES USED FOR HANDLING TRAPS (Continued)

TRAP-CAUSING EVENT	UDL FIELDS	TASK QUEUE	TSW BIT MASK
Send Data*** (SVC6)	UDL.TSKO UDL.TSKN UDL.TSKQ UDL.SDQ	Yes	TSW.TSKM TSW.SDM
Send Message*** (SVC6)	UDL.TSKO UDL.TSKN UDL.TSKQ UDL.MSGR	Yes	TSW.TSKM TSW.PMM
Send Queue Parameter (SVC6)	UDL.TSKO UDL.TSKN UDL.TSKQ	Yes	TSW.TSKM TSW.TCM
Subtask State Change (SVC6)	UDL.TSKO UDL.TSKN UDL.TSKQ	Yes	TSW.TSKM TSW.SUQM
SVC14	UDL.S14O UDL.S14N UDL.SV14	No	TSW.S14M
Timer Termination (SVC2 code 23)	UDL.TSKO UDL.TSKN UDL.TSKQ	Yes	TSW.TSKM TSM.TMCM
Trap Wait	-	No	TSM.WTM

\* Available on the 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 the OS/32 Supervisor Call (SVC) Reference Manual.

\*\*\*\* Available with the 3200MPS Family of Processors only.

NOTE

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

### 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:

```

                MLIBS 8                FETCHES SYSSTRUC.MLB
                NLSTM                SUPPRESSES MACRO LISTING
                $UDL                MACRO TO DEFINE UDL STRUC
MYUDL          ABS 0
MYUDLE         DS  UDL
                EQU  *
                ORG  MYUDL+UDL.TSKQ
                DC   A(Queue)         INITS ADDRESS OF TASK QUEUE
                ORG  MYUDL+UDL.TSKN
                DC   TSW.IOM         QUEUE ENTRY - I/O COMPLETION
                DC   A(TRAP)        INITS ADDR OF TASK QUEUE HANDLER
                ORG  MYUDLE
```

The TSW for a task handling an 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 Chapter 6 (SVC6) in 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:**

```

                MLIBS 8                                FETCHES SYSSTRUC.MLB
                NLSTM                                  SUPPRESSES MACRO LISTING
                $UDL                                  MACRO TO DEFINE UDL STRUC
                $TSW                                  MACRO TO EXPAND TSW EQUATES
                ABS 0
MYUDL           DS  UDL
MYUDLE         EQU  *
                ORG  MYUDL+UDL.TSKQ                 STORES TASK QUEUE ADDR
                DC   A(Queue)
                ORG  MYUDL+UDL.MSGR                 STORES MESSAGE RING ADDR
                DC   A(MESS1)
                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 RIGHT
          LA       2,12(1)            (R1)+12 IS MESS START ADR
          ST       2,WRITE+SVC.1.SAD   STORE ADR IN SVCL PARBLK
          LI       2,63(2)            (R2)+63 IS MESS END ADR
          ST       2,WRITE+SVCL.EAD   STORE ADR IN SVCL PARBLK
          .
          .
          .
          LIS      2,0                 RESET BUFFER FALL BIT IN
          RBT      2,0(1)              MESSAGE BUFFER
          B        TRAP                SEE IF ANY MORE ON QUEUE
TRAPEXIT  EQU      *
          LM       R0,TRAPSAVE          RESTORE REGS (OPTIONAL)
          SVC      9,UDL.TSKO          LOAD OLD TSW

```

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 SVC9. 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 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 the 3200MPS Family of Processors. 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 (called the directed task) establishes an environment for handling task queue entry (TQE) traps on the reception of a message from another task (called the calling task), and then places itself in a trap wait state. 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 SVCL 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 SVCL.



Example:

```

1      PROG DIRECTED TASK WITH TRAP HANDLER
2      MLIBS 8          FETCH SYSSTRUC.MLB
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     *
21     TSW DC TSW.WTM!TSW.TSKM!TSW.PMM    ENABLE TRAP WAIT,TASK Q TRAPS,MESS Q
22     DC 0          RESUME AFTER SVC9
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     $SVCL
29     ALIGN 4
30     WRITE DS SVCL.          SVCL PARAMETER BLOCK
31     ENDBLK EQU *
32     ORG WRITE+SVCL.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+SVCL.SAD    PUT ADR IN SVCL PARBLK
48     LA 2,63(2)          R2 = A(MESSAGE END)
49     ST 2,WRITE+SVCL.EAD    PUT ADR IN SVCL 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          SPECIFY 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, L`TSW`, performs a TSW load. The following program performs the same type of 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
	NLSTM	
QUEUE	DLIST 3	
MESS1	MSGRING 15,R	24-27
START	SETUDL TSKQ=QUEUE,MSGR=MESS1,TSKN=(TMQ,TRAP)	5-17
	LTSW WT,TSKE,TMQ	21-22,37
	EOT RC=0	38
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)	
	RBT 2,UDL.TSKO	
	LTSW PCB=UDL.TSKO	54
ERROR	SVC 3,4	
	END START	

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. See the CAL Macro/32 Processor And Macro Library Utility Reference Manual for more details for expanding macro instructions during the assembly process.

### 3.6.6 Writing FORTRAN Trap-Handling Programs

The 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 OS/32 Common Assembly Language (CAL) routines.

The SMPLSVCS.PAS file supplied with the 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: UDL_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  
OLDLB    DSF   1          STRUC OF ACTIVATION REC ON STACK  
RETAD    DSF   1          OLD LOCAL BASE (R2)  
SLINK    DSF   1          RETURN ADDRESS  
          ENDS          STATIC LINK  
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 SVC9 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 Pascal User Guide, Language Reference and Run-Time Support Reference Manuals.



## 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 (cancels 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: 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 communications 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 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:

$$\left[ \left\{ \begin{array}{l} \text{voln:} \\ \text{dev:} \end{array} \right\} \right] [\text{filename}] [.\text{ext}] \left[ / \left\{ \begin{array}{l} \text{actno} \\ \text{file class} \end{array} \right\} \right]$$

### 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, 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, 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, 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 to 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. Account number 255 is reserved for the MTM system administrator. Account number 0 is for system files and is the default for all operator commands. Under MTM, only privileged users may specify an account number rather than a file class.

**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.

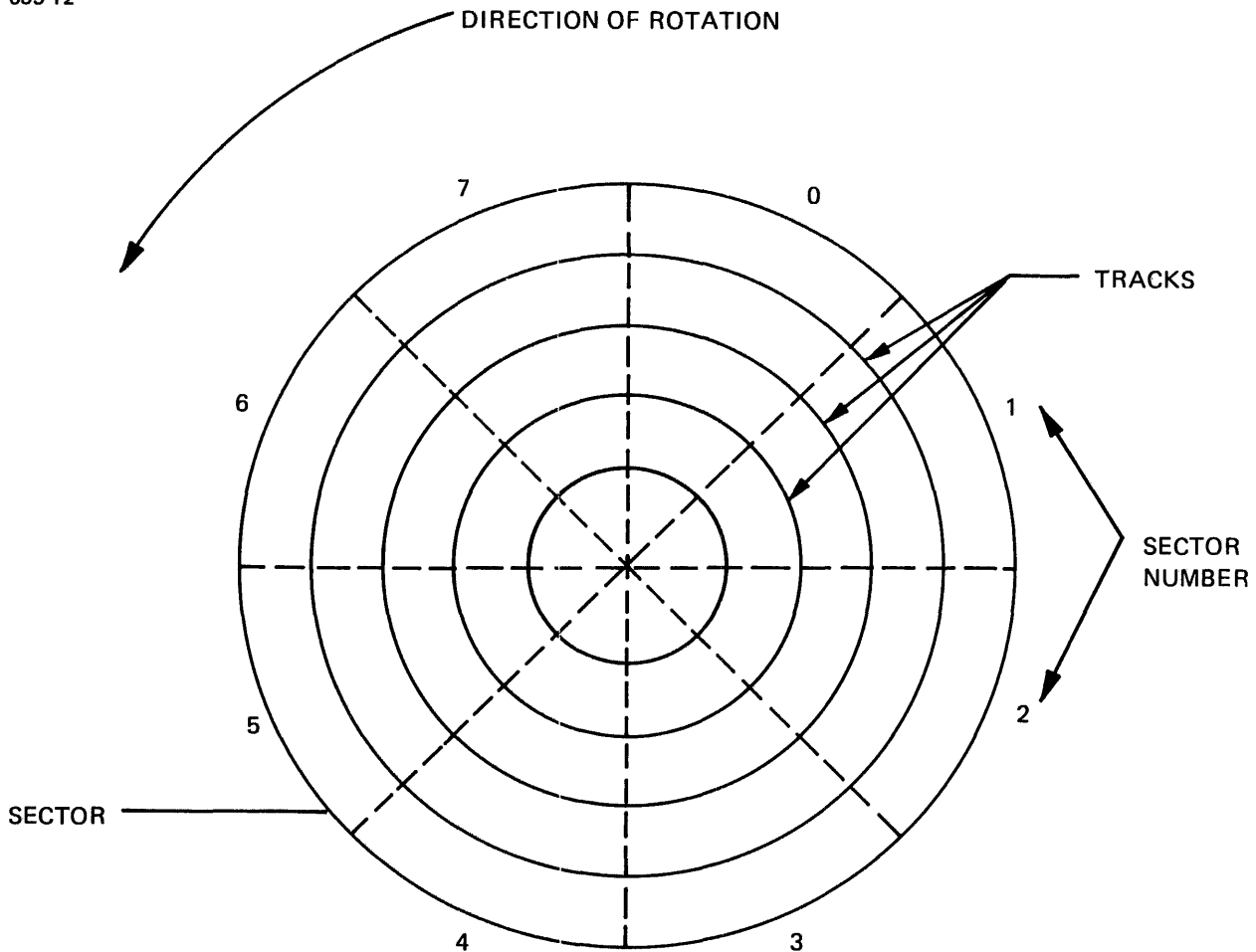


Figure 4-1 Formatted Disk Surface

When a disk is initialized, the tracks are divided into sectors. The disk surface shown in Figure 4-1 is divided into eight 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 unused 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.

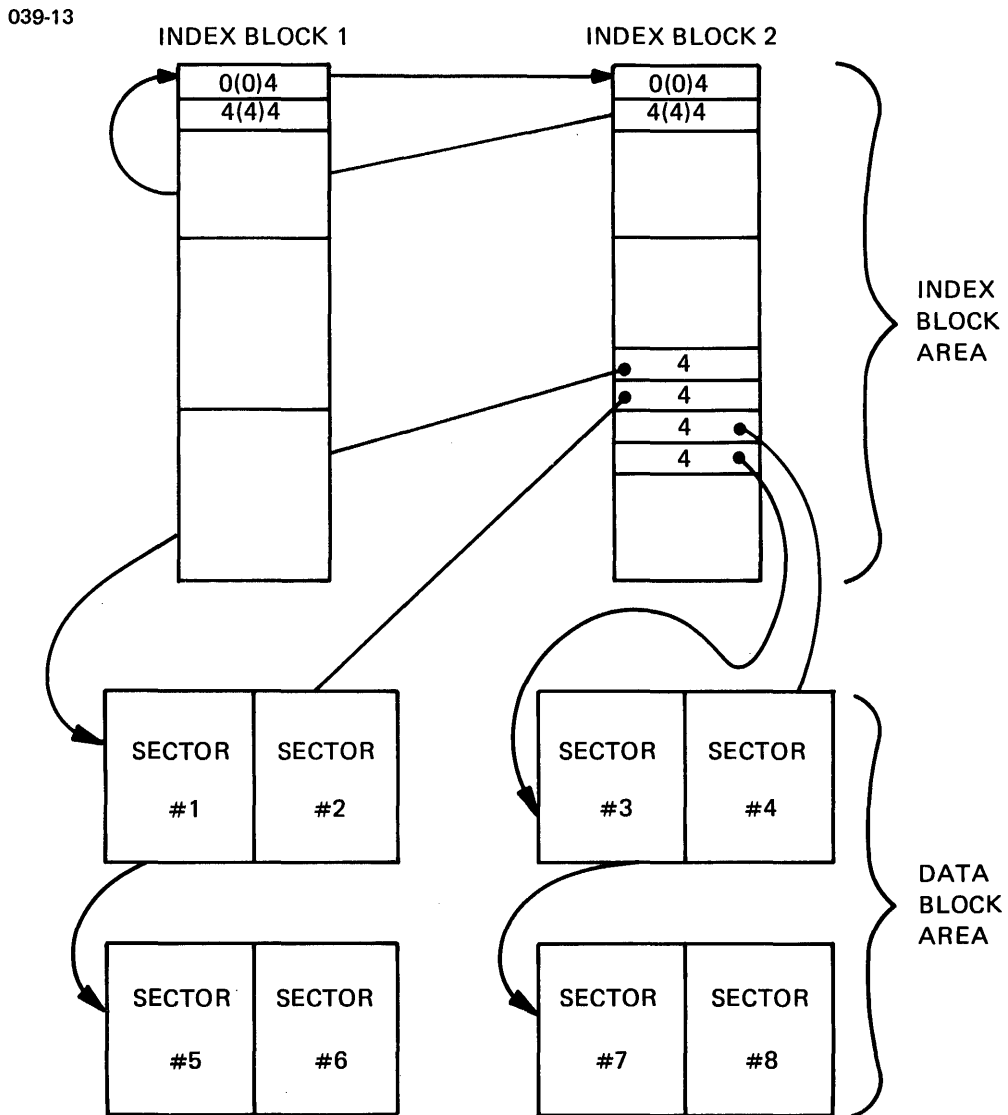


Figure 4-2 Linked-List Indexed File Organization

### 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 two 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 DISK ORGANIZATION

Disk packs organized under OS/32 contain two distinct categories of information:

- Control information
- User-defined data

Control information includes the volume descriptor, bit map, file directory and pack administration file, which perform the following functions:

- The volume descriptor is a data structure used by the file manager to identify the disk volume and to point to structures that contain the file and disk space information.
- The bit map, also called a sector allocation map, is used to indicate which sectors are in use on a disk. Also, in the case of defective sectors, the bit map indicates which sectors are unusable.
- The file directory is a file used by the file manager to keep track of all files on the disk. This file is organized as a linked-list, where each entry of the list identifies and describes a file on the disk.
- The pack administration file contains a list of defective sectors and a record of the administrative history of the disk pack.

User-defined data includes all the contiguous and indexed application files.

The Mirror Disk facility maintains two identical copies of the user-defined files on separate disk packs. These files are physically synchronized by keeping them identical bit for bit and sector for sector. The control information is also mirrored in normal write and read operations in the mirrored environment. This control information is important in establishing whether two disks are compatible for mirroring.

#### 4.5 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. OS/32 does not support multi-volume files.

To meet the requirements of applications running in a real-time environment, the OS/32 file manager supports five 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
- Long record

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

##### 4.5.1 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 two 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'.

#### 4.5.2 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.

Data block size describes the physical block size to be used for buffering and debuffering operations on indexed files or data communications devices. Data block size represents the block size in sectors of the physical data blocks containing the file.

Index block size represents the size of index blocks in sectors. Index blocks are actual pointers to physical data blocks.

#### 4.5.3 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. A random write to a record past the current end of file will cause that record to become the new end of file, and empty records will be created between the previous last record and the new last record. However, like contiguous files, the record length is fixed at 256 bytes (one sector), and any number of sectors can be transferred in a single I/O operation.

#### 4.5.4 Long Record Files

OS/32 also supports the long record file type which is used to perform I/O operations with very large logical record lengths. This file type is similar to an extendable contiguous file but differs in that the logical record length of the long record file is specified by its data block size.

For long record files the maximum size of a data block is 65,535 (64K) sectors, which is equivalent to an absolute maximum logical record length of 16,776,960 (16M) bytes. With the use of a long record file, a maximum of 16Mb of data can be read from or written to memory with a single I/O command. In practice, however, the actual maximum logical record length for any given system is limited by the amount of memory available for I/O buffers in the application task.

#### 4.6 MIRROR DISK ENVIRONMENT

The Mirror Disks facility provides a service such that in the event of a disk failure a system can continue normal operation using a duplicate copy of the failed disk. This is made possible by maintaining pairs of disks in physical synchronization. Disks are physically synchronized if the user-defined data files are identical bit for bit. One disk is designated as the primary disk and the other is designated as the secondary disk.

The only necessary hardware to support the facility is the extra disks required. These must be the same size as those which are to be mirrored.

During normal operation, any write to a mirrored disk will result in writes to both disks. This not only includes writes to the files themselves, but also file allocation, file renaming, etc. Control is returned once both writes are complete. If one disk fails, the system continues normal operation using the remaining good disk. Reads are scheduled from one of the pair of mirrored disks, the primary disk. The remaining disk is called the secondary disk. In the event that an attempt to read from the primary disk fails, data is read from the secondary disk instead. The programmer interface is essentially unchanged. It is possible, however, to detect when a primary read has failed. This requires an extended option as described in the OS/32 Supervisor Call (SVC) Reference Manual.

For more detailed information on the mirror disk facility, see the OS/32 Operator Reference Manual.

#### 4.6.1 Disk Failure

In the case of a failure of one of a mirrored pair of disks, the following occurs:

- Repeated messages are output to the system console, at regular intervals, indicating the disks are no longer synchronized.
- The secondary disk immediately takes over if the primary disk failed, with no interruption to the system software using the disk.

#### 4.6.2 Normal Input/Output (I/O) Performance

The disks of a mirrored pair should be on separate selector channels (SELCHs) to ensure the efficiency of write operations to the mirrored disks. These disks are not required to be on separate SELCHs, but if they are, the performance degradation is likely to be minimal.

Degradation varies according to the volume of data being written for mirror disks that are on the same SELCH. It is estimated that it takes 22% longer to write a sector and 45% longer to write a 16kb buffer in a mirrored environment. Read operations are not adversely affected.

The Mirror Disk facility is designed to be user-transparent. However, extended SVCL options are available (see the OS/32 Supervisor Call (SVC) Reference Manual for more information).

#### 4.7 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, extendable contiguous or long record) by allocating additional index blocks and data blocks.
- Delete a file and return the space to the available space inventory.



## 4.8 VOLUME DESCRIPTOR

The file manager must keep track of all the files on a disk and the storage space available to the files to perform its operations effectively. 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 Fastchek Utility initializes the volume descriptor in logical block address (LBA) 0 (sector 0, head 0, cylinder 0). See the OS/32 Fastchek Reference Manual.

039-14

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
24(18)	ILA (4) RESERVED FOR OS/16
28(1C)	SDP (4) SECOND DIRECTORY BLOCK POINTER
32(20)	SYNC (4) SYNCHRONIZATION STAMP

Figure 4-3 Volume Descriptor

The volume descriptor is used as the basis for establishing whether a pair of disks are synchronized and mirrored. The following fields of the volume descriptor are used:

- Disk volume name
- Volume on-line bit of the attribute words
- First directory block pointer
- Pointer to the bit map
- Synchronization stamp

After the initial synchronization of the two disks, the synchronization stamp is the main factor used to control the mirror disk operation.

The stamp is initialized to zero whenever a disk pack is initialized or checked by the Fastchek Utility. A zero setting indicates that the Synchronization Utility (DISCSYNC) must be run. See the OS/32 System Support Utilities Reference Manual for information. It is important that disk packs used for mirroring are initialized by the Fastchek Utility.

The synchronization stamp is only calculated and placed in the volume descriptor when a pair of mirrored disks are marked off together. The stamp is cleared again when the disks are marked on. In all circumstances, the synchronization stamp remains with a zero setting and must be resynchronized using the DISCSYNC Utility when used as a mirrored pair again.

#### 4.8.1 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.

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

0(0)			
FNM (8) FILE NAME			
8(8)		EXT (3) EXTENSION	11(8) ACT (1) ACCT NUMBER
12(C) FLBA (4) FIRST LOGICAL BLOCK ADDRESS			
16(10) LLBA (4) LAST LOGICAL BLOCK ADDRESS			
20(14) WKEY (1) WRITE KEY	21(15) RKEY (1) READ KEY	22(16) LRCL (2) LOGICAL RECORD LENGTH	
24(18) DATE (4) DATE FILE ALLOCATED			
28(1C) LUSE (4) DATE FILE LAST WRITTEN			
32(20) WCNT (2) WRITE COUNT		34(22) RCNT (2) READ COUNT	
36(24) ATRB (1) ATTRIBUTES	37(25) BLSZ (1) BLOCK SIZE	38(26) INBS (1) INDEX BLOCK SIZE	39(27) SHDS (1) SHARED DISK SUPPORT (BOONSEWERE)
40(28) CSEC (4) # CURRENT SECTOR/# LOGICAL RECORDS			
44(2C) LASN (4) DATE FILE LAST ASSIGNED			

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
- 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.)
- 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, indexed or long record file
- File's access privileges
- 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
- Data block size and index block size if the file is an indexed, nonbuffered indexed, extendable contiguous or long record file
- 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 on-line with secondary directory support. 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 on-line (see the OS/32 Operator 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 off-line and then marked back on-line 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.

039-17

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)

Within a mirror disk environment, attention should be given to the file directories of the mirrored disks. Since the file directories are mirrored bit for bit on the secondary disk, the primary file directories must begin at the same position on both disks. If the secondary file directory SYSTEM.DIR is present in memory, it may need to be deleted and recreated for the primary disk if there are any defective sectors on the secondary disk.

#### 4.8.2 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.

When marking on mirror disks, the bit map is used to synchronize the disks. If the disks are compatible, the primary disk's bit map is updated for all sectors that are defective on the secondary disk and for any extra sectors taken up by the secondary disk's pack administration file. The pack administration file (PACKINFO.DIR) is a file created and maintained by the Fastchek Utility containing a list of defective sectors. Defective sectors are an important factor in establishing whether two disks can be mirrored. For further information regarding the pack administration file see the OS/32 Fastchek Reference Manual.

#### 4.8.3 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, MTM command or 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. Temporary files are named as &nnnnnnn; where nnnnnnn is a unique, internally assigned identifier.

#### 4.9 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 or file 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 pertinent lu should be assigned to the NULL: device. This assignment allows the operation to be performed without generating any output from that operation.

#### 4.10 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.

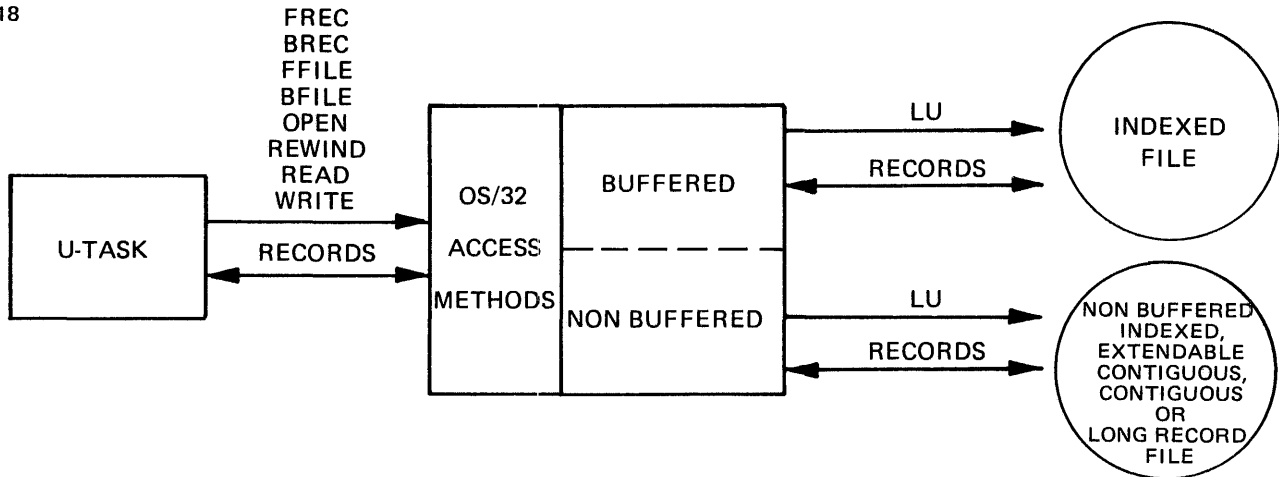


Figure 4-7 Task Interfaces to Access Methods

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, indexed files and long record 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 whenever 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 is positioned to the number of the record containing a filemark. Otherwise, the record pointer is set to zero.

The intelligent peripheral controller (IPC) tape driver does not support backspace filemark.

- Forward Space Filemark (FFILE)

If the file is a contiguous file, the record pointer backspaces 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.

The IPC tape driver does not support backspace record.

- 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 PER TRANSFER
Contiguous	2 to capacity of file
Extendable contiguous	2 to capacity of file (Read) 2 to capacity of disk (Write)
Indexed	1 to record length
Nonbuffered indexed	2 to record length
Long Record	2 to record length

#### 4.10.1 Buffered Input/Output (I/O) (Indexed Files)

An SVC1 I/O proceed request to an indexed file executes in a different manner than an I/O proceed to other file types or devices. 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 user buffer rather than between the user buffer and the file on disk. An I/O proceed for an indexed file, therefore, functions like a wait I/O request with one exception: an entry is placed on the task queue as with normal I/O proceed. When data is transferred, however, the task execution stops and waits until the completion of I/O to continue.

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 five 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 five 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.

#### 4.10.2 Nonbuffered Input/Output (I/O)

Nonbuffered I/O is used for contiguous, extendable contiguous, nonbuffered indexed and the long record 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 four nonbuffered file types; however, some restrictions apply.

With all nonbuffered I/O file types, transfer of data begins and ends on a sector boundary. Partially filled sectors are padded with the last two 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.

##### 4.10.2.1 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 two 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.

The EOF status for a contiguous file is returned as an end of medium (EOM), X'90', marker. Pseudo filemarks, X'1313', are supported by contiguous files; the standard EOF, X'88', status is returned when a pseudo filemark is encountered.

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.

##### 4.10.2.2 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 two bytes of the data. To read records 15 and 62,015, data blocks 1 and 249 are accessed. If a 5 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 be performed on existing file records or on the next record after EOF.

#### 4.10.2.3 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. As in contiguous files, the EOF status is returned as an EOM, X'90', marker.

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 five 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 fill a disk completely by writing to a sector with an unusually large random address.

#### 4.10.2.4 Accessing Long Record Files

In long record files, as in other nonbuffered I/O file types, data is transferred directly between the user's buffer in task space and the file on disk without incurring the overhead required by buffering in system space. The main difference between this file and other nonbuffered I/O types is that in a long record file, the data block size can be greater than 255 sectors. In fact, the maximum size of a data block for a long record file can be up to 64K sectors. In accessing this file type, it must be remembered that the logical record length of the long record file is specified by its data block size. For example, if a long record file exists with data blocks of 4000 sectors each, to read sector 6000, the user must access the

second data block. Sector 6000 is in the second record of the file because the record length of the long record file is specified by the data block size.

#### 4.11 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 permissible 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

	E R S W	E R O	S R O	S R W	S W O	E W O	S R E W	E R E W
ERSW	-	-	-	-	*	-	-	-
ERO	-	-	-	-	*	*	-	-
SRO	-	-	*	*	*	*	*	-
SRW	-	-	*	*	*	-	-	-
SWO	*	*	*	*	*	-	-	-
EWO	-	*	*	-	-	-	-	-
SREW	-	-	*	-	-	-	-	-
ERW	-	-	-	-	-	-	-	-

\* 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. The new access privileges must be compatible with existing ones.

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 4-2. The read/write keys can protect a file from being read from or written to by any task to which this file is assigned. For example, if a file's read and write keys are X'00' and X'07', respectively, the task to which this file is assigned can read from the file but it cannot write to it unless the file is assigned to the task with the same write key.

TABLE 4-2 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 the 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 EWO 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 on-line. A disk volume can be marked on-line 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.

#### 4.12 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 4-3. The following sections describe the advantages and disadvantages of using each of the five file types, as well as some tips on handling disk fragmentation.

TABLE 4--3 FILE TYPE SUMMARY

TYPE	DATA ORGANIZATION	RECORD LENGTH (BYTES)	FILE LENGTH	BUFF-ERED (I/O)	RECORD POINTER VALUE	BYTE LIMIT PER TRANSFER	
						MINIMUM	MAXIMUM
Contiguous	Contiguous	256	Fixed at allocation	No	Sector number	2	Capacity of file
Indexed	Linked-list indexed	1 to 65,535	Open-ended	Yes	Logical record number	1	Record length
Non-buffered indexed	Linked-list indexed	2 to 65,535	Open-ended	No	Logical record number	2	Record length
Extendable contiguous	Linked-list indexed	256	Open-ended	No	Sector number	2	Capacity of file (read) available disk space (write)
Long record	Linked-list indexed	256 to 16,776,960	Open-ended	No	Logical record number	2	Record length



#### 4.12.1 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 an I/O proceed 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.

#### 4.12.2 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, using data block sizes of 5 to 20 sometimes causes a job to become CPU-bound. However, many simple tasks do not become CPU-bound 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.)

### 4.12.3 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.

### 4.12.4 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.

#### 4.12.5 Using Long Record Files

Long record files provide the user with the unusual ability to read or write very long logical records to or from memory in a single I/O operation. This file type permits the user to transfer more than 255 sectors of data in one I/O operation, thus quickly freeing system resources for other purposes. If, for example, we had 200,000 sectors of data being input to the system and the I/O buffer size was 4000 sectors, the entire file could be written to disk in only 50 writes as compared with the 785 writes needed for an extendable contiguous file of the same length. With a larger logical record length, even fewer writes would be necessary.

The long record file type would be advantageous to a user who finds it necessary to transfer vast quantities of data more rapidly than is possible with the other file types. For example, such a user could establish several extremely large buffers in task space, and, in using the long record record file type, each buffer could be written to a disk in a single I/O operation.

#### 4.12.6 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.

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 it does 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 five 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 SVC7. To request transfer of information to the resource given to the task by SVC7, an SVC1 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.

A number of methods for writing application programs that access system services are available. 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 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 Pascal compiler. These programming methods are outlined in Figure 5-1.

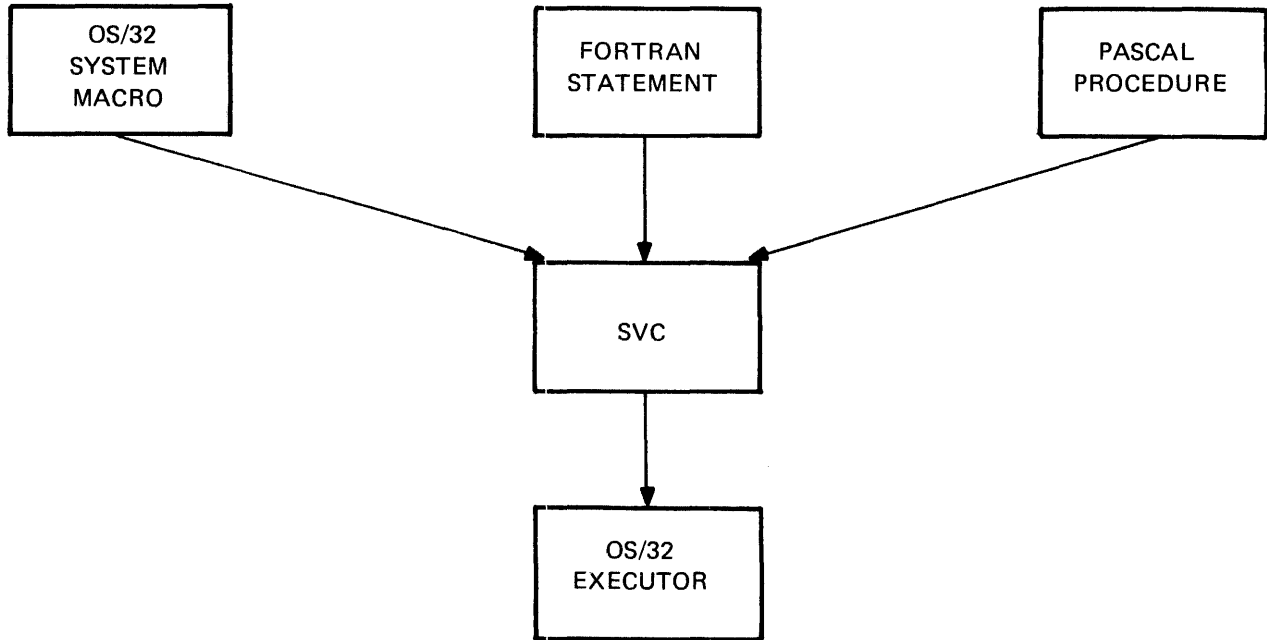


Figure 5-1 Task Interface to OS/32 Executor Routines

This chapter demonstrates how each of these methods is used to write a program that accesses two OS/32 system services; namely, the I/O and file management services. First, the SVC1 and SVC7 parameter block structures that pass data to the I/O and file management executor routines are discussed. See 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 SVC1, SVC5, SVC6, SVC7 and SVC13 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 SVC1 parameter block using the \$SVC1 macro. \$SVC1 defines the parameter block shown in Figure 5-2.

039-20

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) LENGTH OF DATA TRANSFER (SVC1.LXF)			
20(14) EXTENDED OPTIONS (SVC1.XOP)			

Figure 5-2 SVC1 Parameter Block Defined by \$SVC1

TABLE 5-1 SVCL 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 SVC1 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 SVC1 parameter block for an SVC1 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
	DB 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 SVC1 as follows:

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

The following program uses SVC1 to build two parameter blocks, one for an SVC1 that performs a read operation from a file or device assigned to lu1 and one for an SVC1 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 SVC1 parameter block to determine if the I/O operation was successful.

If the mirror disk option is selected and a primary read failure occurred, the SVCL parameter block receives an X'86' in the device-dependent status. Data is read to the secondary disk only. If the read was successful, the SVCL parameter block receives a 0 in the device-independent status. Data is read to both primary and secondary disks.

Example:

```

SVCL      PROG  SIMPLE SVCL EXAMPLE
          MLIBS 8,9,10          DECLARE MACRO LIB TO CAL/MACRO
          NLSTM                 DON'T LIST MACRO EXPANSIONS
          FREZE                 FREEZE LINE NUMBERS
          $SVCL                DEFINE SVCL STRUCTURE
          ALIGN 4              ALIGN PARBLK ON FULLWORD
SVCL.IN   DS      SVCL.        ALLOCATE INPUT PARBLK
ENDPBK    EQU     *
          ORG     SVCL.IN+SVCL.FUN
          DB     SV1.READ!SV1.WAIT  FUNCTION CODE=READ & WAIT
          DB     1              LOGICAL UNIT=1
          ORG     SVCL.IN+SVCL.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
SVCL.OUT  DS      SVCL.        ALLOCATE OUTPUT PARBLK
ENDPBK    EQU     *
          ORG     SVCL.OUT+SVCL.FUN
          DB     SV1.WRIT!SV1.WAIT  FUNC CODE=WRITE & WAIT
          DB     2              LOGICAL UNIT=2
          ORG     SVCL.OUT+SVCL.SAD
          DC     A(BUFF)        SAME BUFFER AS INPUT
          DC     A(BUFFE)
          ORG     ENDPBK
START     EQU     *
          SVC     1,SVCL.IN      ISSUE SVCL TO READ LUL
          LH     0,SVCL.IN+SVCL.STA CHECK STATUS
          BM     ERROR          <0, ERROR
          SVC     1,SVCL.OUT     ISSUE OUTPUT SVC TO LU2
          LH     0,SVCL.OUT+SVCL.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 SVC7. For example, a task requests an lu assignment via the SVC7 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 SVC7 close function is used to cancel the assignment. The SVC7 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 SVC7 parameter block contains the file descriptor (fd) of the file on which the OS/32 executor is to perform the requested operation.

039-21

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) FILENAME (SVC7.FNM)			
16(10)			
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 SVC7 Parameter Block Defined by \$SVC7

SVC2 code 16 can be used to pack an fd into the SVC7 parameter block. This SVC must be used if the fd to be packed into the block specifies an account number. The parameter block for SVC2 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 SVC2 code 16 parameter block.

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

Figure 5-4 SVC2 Code 16 Parameter Block

The following program issues an SVC7 and SVC2 code 16 to assign lu2 to a file named MYFILE.TXT/P. The program uses \$\$SVC7 to define an SVC7 parameter block. A parameter block is also built for SVC2 code 16. This block contains the number of the register that holds the fd to be packed and the address of the SVC7 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 SVC7 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 SVC2,16 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=BUFBE
```

The WRITE macro routine builds an SVC1 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 SVC1 parameter block. The routine then issues SVC1.

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

#### Example:

```
                PROG  SVC1 AND SVC7 MACRO EXAMPLE
                MLIBS 8,9,10
BUFF            DS      80
BUFBE          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=BUFBE
                WRITE LU=2,ADDR=BUFF,RECL=80,ENDADDR=BUFBE
                EOT  RC=0
                END    START
```

In the above example, the ALAS macro routine builds an SVC7 parameter block and then issues an SVC7 to allocate the file TEST2.DTA and assign the file to lu2. The ASSIGN macro routine builds another SVC7 parameter block and issues an SVC7 to assign CARDIN.FMU/S to lu1.

The READ macro routine builds an SVCL parameter block and issues an SVCL to read data from CARDIN.FMU/S into the user buffer at location BUFF. The WRITE macro routine builds an SVCL parameter block and issues an SVCL 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 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 SVCL parameter block using the arguments in the CALL SYSIO statement. After the block is built, SYSIO issues an SVCL 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 SVCL 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' ; SVCL READ WAIT FUNCTION CODE
CALL SYSIO(PBLK, FC, LU, BUFF, NBYTES, RANADD)
CALL IOERR(PBLK, ISTATUS)
IF (ISTATUS.NE.0) GO TO 10
.
.
.
```

**Where:**

PBLK specifies the array in which the parameter block will be built.

BUFF is the buffer array that is to be output.

LU is the logical unit assigned to the output device.

NBYTES specifies the number of bytes that will be output.

Y'28' is the function code telling the OS/32 executor to perform the write operation and suspend task execution until data transfer is completed.

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

To access file management services, the FORTRAN VII compilers provide auxiliary 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 lul by building an SVC7 parameter block and issuing an SVC7 to assign the file. STATUS=OLD tells the OS 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 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:CMD\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 lu. With the OPEN procedure, an SVC7 parameter block is built and an SVC7 is issued to assign the lu. This program checks the status field returned by the OPEN procedure. This status is the SVC7 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		Directory block	4-12
Access methods		Disk	4-14
buffered	4-17	compress	4-29
	4-20	drives	4-2
nonbuffered	4-17	failure	4-9
Access privileges	4-22	file management services	4-1
AFCONT	3-6	formatting	4-3
AFPAUSE	3-6	fragmentation	4-29
APU	3-21	marking on	4-14
	3-34	space management	4-26
task queue entry	3-26	Disk file types. See file types.	4-10
Arithmetic fault		Disk organization	
error messages	3-7	control information	4-6
Assigning files	4-16	user-defined data	4-6
Authorized user utility	1-5	Dynamic system space	4-8
Auxiliary processing unit. See APU.			
		E	
B		End of file	4-21
Bit map	4-6	End of medium	4-21
	4-16	Environment	
		mirror disk	4-9
			4-15
C		real-time	1-1
CAL	5-1		1-3
Central processing unit. See CPU.		time-sharing	3-1
CLOSE	5-12		4-7
Code			1-1
impure	2-2	Executor routines	2-13
pure	2-4	file management	5-2
Common Assembly Language. See CAL.		sample program	5-10
Contiguous files	4-6	task interface to	5-1
	4-7	Extendable contiguous files	4-8
	4-18		4-19
accessing	4-21	accessing	4-21
using	4-26	using	4-28
CPU	1-4		
	3-21	F,G	
	3-34	Faults	
	4-26	arithmetic	3-5
bound processor	4-28		3-7
CTOP	2-8		3-16
		data format/alignment	3-5
D			3-16
Data block	4-5	illegal instruction	3-5
	4-27		3-16
buffers	4-8	memory access	3-5
DEBUG/32	2-4		3-16
Device		power restoration	3-5
driver	4-2		3-16
trap-generating	3-33	fd	4-2

File			H	
directories	4-6			
	4-12	History records		2-2
security	4-23			
support services	4-1		I, J, K	
File allocation				
permanent	4-16	I/O		4-2
temporary	4-16			4-20
File descriptor. See fd.				5-3
File management services				1-3
accessing	5-7			1-3
File manager	4-1			4-16
	4-3			4-10
	4-5			1-3
	4-7			4-16
	4-14			4-10
primary	4-12	Image file		2-4
File organization	4-3	format		2-2
contiguous organization	4-6	Index block		4-5
linked-list indexed	4-5			4-22
File types	1-4			4-27
assigning	4-2	Indexed files		4-28
	4-16			4-5
contiguous	4-6			4-8
	4-7			4-19
	4-20			4-20
	4-21	using		4-27
	4-27	Input/output. See I/O.		
disk	4-25	Intelligent peripheral		
extendable contiguous	4-7	controller. See IPC.		
	4-8	Interrupt servicing		3-1
	4-19	Intertask communication		
	4-22	restrictions on		2-13
	4-27	IPC		4-18
	4-5			4-19
	4-7			
	4-8		L	
	4-19			
	4-20	LIB		2-2
	4-27			2-8
long record	4-7	Link		2-1
	4-18			2-8
	4-22			3-2
	4-29	Linked-list organization		4-8
nonbuffered indexed	4-7	Loader information block.		
	4-8	See LIB.		
	4-19	LOC		3-1
	4-21			3-18
	4-28			
Filemark	4-6	Location counter. See LOC.		
	4-8	Logical unit. See lu.		
	4-27	Long record files		4-9
pseudo	4-21			4-18
Files	4-1	accessing		4-22
assigning	4-2	using		4-29
FORTTRAN	4-16	lu		2-12
	5-1			4-16
	5-10	assignment		5-7
RTL	3-36			
	5-10		M	
sample program	5-11			
trap-handling programs	3-36	MAC		2-5
		MARKON command		4-14
		MAT		2-5
		Memory access controller.		
		See MAC.		





UDL (Continued)  
  building a                   3-31  
  fields                       3-13  
  TSW swap                     3-9  
User-dedicated location.  
  See UDL.  
User task.   See u-task.

V

VAT                             2-5  
Virtual address translator.  
  See VAT.  
Virtual task master.   See  
  VTM.  
Volume descriptor       4-6  
                              4-11  
VTM                             1-4

W, X, Y, Z

WRITE macro                   5-10





## PUBLICATION COMMENT FORM

We try to make our publications easy to understand and free of errors. Our users are an integral source of information for improving future revisions. Please use this postage paid form to send us comments, corrections, suggestions, etc.

1. Publication number \_\_\_\_\_

2. Title of publication \_\_\_\_\_

3. Describe, providing page numbers, any technical errors you found. Attach additional sheet if necessary. \_\_\_\_\_

\_\_\_\_\_

4. Was the publication easy to understand? If no, why not? \_\_\_\_\_

\_\_\_\_\_

5. Were illustrations adequate? \_\_\_\_\_

\_\_\_\_\_

6. What additions or deletions would you suggest? \_\_\_\_\_

\_\_\_\_\_

7. Other comments: \_\_\_\_\_

\_\_\_\_\_

From \_\_\_\_\_ Date \_\_\_\_\_

Position/Title \_\_\_\_\_

Company \_\_\_\_\_

Address \_\_\_\_\_

\_\_\_\_\_

FOLD

FOLD

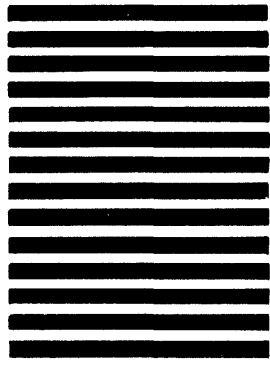


NO POSTAGE  
NECESSARY  
IF MAILED  
IN THE  
UNITED STATES

**BUSINESS REPLY MAIL**  
FIRST CLASS      PERMIT NO. 22      OCEANPORT, N.J.

POSTAGE WILL BE PAID BY ADDRESSEE

**Concurrent Computer Corporation**  
2 Crescent Place  
Oceanport, NJ 07757



**ATTN:  
TECHNICAL SYSTEMS PUBLICATIONS DEPT.**

FOLD

FOLD

STAPLE

STAPLE