PERKIN-ELMER

OS/32
APPLICATION LEVEL PROGRAMMER

Reference Manual

48-039 F00 R00

The Perkin-Elmer Corporation, Computer Systems Division 2 Crescent Place, Oceanport, New Jersey 07757

TABLE OF CONTENTS

CHAPTERS (Continued)

CHAPTERS (Continued)

FIGURES

TABLES

PREFACE

This manual is an introduction to Perkin-Elmer 32-bit software products and includes an overall description of task traps, macros, and file management. The manual is intended for application level programmers designing and programming tasks to run under OS/32.

Chapter 1 is an overall description of Perkin-Elmer 32-bit software products used for program preparation. Chapter 2 describes the task structures. Chapter 3 presents traps and details trap-causing events. Chapter 4 presents system macros and data structures. Chapter 5 contains information on volume organization, file types, classes, and file management methods.

This manual has been extracted from the obsolete OS/32 Programmer Reference Manual, Publication Number S29-613. This manual applies to the OS/32 R06 software release and higher.

The following manuals can be used in conjunction with this manual:

MANUAL NAME	PUBLICATION NUMBER
OS/32 AIDS User's Guide	S29-374
BASIC Level II Reference Manual	S29-488
OS/32 Bit Synchronous Communications Reference Manual	S29-544
COBOL Reference Manual	S29-545
CORAL 66 Reference Manual	S29-587
Source Updater User Guide	S29-630
FORTRAN VII D User Manual	S29-657
FORTRAN VII O User Manual	S29-659
RPG II Reference Manual	S29-661
OS/32 COPY User Guide	S29-676

MANUAL NAME	PUBLICATION NUMBER
OS/32 TEXT User Guide	S29-677
OS/32 Link Reference Manual	48-005
OS/32 Edit User Guide	48-008
32-Bit Systems User Documentation Summary	50-003
Pascal User Guide, Language Reference, and Run Time Support Reference Manual	48-021
OS/32 System Planning and Configuration Guide	48-024
OS/32 Operator Reference Manual	48-030
System Generation (SYSGEN/32) Reference Manual	48-037
OS/32 System Level Programmer Reference Manual	48-040
OS/32 Multi-Terminal Monitor (MTM) Reference Manual	48-043
Common Assembly Language/32 (CAL/32) Programming Reference Manual	48-050
CAL Macro/32 Processor and Macro Library Utility Reference Manual	48-057

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

CHAPTER 1
PERKIN-ELMER OS/32 PROGRAMMING LANGUAGES AND UTILITIES

1.1 INTRODUCTION

This chapter provides an overview of the programming languages and utilities available for program development under the OS/32 and MTM environments. The Reliance environment is also available for transaction processing.

1.2 ENVIRONMENTS

Programs can be developed under the basic OS/32 environment or the multi-terminal monitor (MTM) environment. Programs developed in an OS/32 environment can be entered only at the system console. Programs developed in an MTM environment can be entered at any available terminal. MTM is an OS/32 extension that allows time sliced interactive programming and batch programming. If a system is configured to include the Reliance extension in addition to MTM, programs cannot be developed while running under Reliance. The terminal user, however, can switch from Reliance to the MTM environment by using the environment control monitor (ECM). If programs are then developed in the MTM environment, they cannot be added to the Reliance system unless the Reliance system is completely shut down. For further details concerning the use of the ECM, see the Environment Control Monitor (ECM) Reference Manual.

1.3 PERKIN-ELMER OS/32 PROGRAMMING LANGUAGES

The following nine languages are supported by OS/32:

- Common Assembly Language/32 (CAL/32)
- CAL Macro/32
- FORTRAN VII Development (D) Compiler
- FORTRAN VII Optimizing (O) Compiler
- COBOL
- BASIC Level II

- CORAL 66
- RPG II
- Pascal

1.3.1 Common Assembly Language/32 (CAL/32)

CAL/32 produces the Perkin-Elmer 32-bit object code format from source code input. Features include:

- Program relocation
- Program segmentation
- Complex data definitions
- Expression analysis
- Code optimization
- Conditional assembly instructions

1.3.2 CAL Macro/32

CAL Macro/32 allows programmers to define macros for use in program generation. A macro is a copy of a frequently used assembler code sequence that is inserted into a macro library. A macro call or request is made to the library to insert a macro into the body of the calling task. Once inserted, the macro is expanded by the CAL Macro/32 processor into intermediate source statements that CAL/32 can convert into object code for processing with the rest of the task. Features include:

- Ability to process both user-defined and Perkin-Elmer supplied macros
- Positional, keyword, or mixed mode macro prototype statements
- Nested macro instructions
- Conditional macro expansion independent of conditional assembler instructions
- Ability to incorporate variables within a macro definition
- Macro call instructions for calling frequently used macro definitions into memory at the start of macro processor execution
- Macro trace facility

1.3.3 FORTRAN VII Development (D) Compiler

The Perkin-Elmer FORTRAN VII Development (D) Compiler passes through the source program only once to produce object code in 32-bit format. Features include:

- Over 100 compile time diagnostic routines
- Run time error messages
- Optional run time debug facility for checking subscript values and tracing variables and labeled statements
- Conditional compilation for diagnostic programming

1.3.4 FORTRAN VII Optimizing (O) Compiler

FORTRAN VII O is designed to minimize user program execution time by producing object code in 32-bit loader format or CAL/32 source format. When global optimization is enabled, program flow and language constructs are analyzed at the source program level, reducing the number of computations required at execution. Features include:

- Diagnostic compilation routines (250)
- Run time error messages
- Optional run time debug facility for checking subscript values
- Optional trace facility for variables and labeled statements
- Conditional compilation for diagnostic programming
- Batch compilation facility for compiling several subprograms using a single compiler invocation

1.3.5 COBOL

The COBOL compiler processes COBOL source statements to produce CAL/32 source statements that are assembled by CAL/32 into object programs. Functions include:

- Sequential I/O
- Relative I/O
- Indexed I/O
- Interprogram communication

- Table handling
- Sort
- Debug
- Library functions

1.3.6 BASIC Level II

The BASIC Level II Interpreter allows users to create, execute, and modify programs interactively. Features include:

- Single/double precision floating point
- ASCII and binary I/O
- File and device access
- Program and matrix manipulation
- String operations
- User-defined functions
- Tracing
- Programmed error handling
- Syntax error checking

1.3.7 CORAL 66

CORAL 66 is a high level language primarily designed for implementing online real time systems. Perkin-Elmer CORAL 66, which is based on Algol 60, incorporates features of FORTRAN and Jovial. Major features offered by the compiler include:

- Block structure
- Algol-like procedures
- Independent compilation
- Code inserts
- Built-in macro scheme
- Mixed arithmetic
- Library functions
- Packed data format

1.3.8 Report Program Generator (RPG II)

RPG II is a high level language primarily designed for file updating and report generation. RPG II provides seven preformatted forms used to code programs to input, process, and retrieve data files. Sequential, relative, and indexed files can be accessed randomly or sequentially. Perkin-Elmer RPG II also can be used to process online files maintained by the Perkin-Elmer database system, Data Management System/32 (DMS/32).

1.3.9 Pascal

The Pascal compiler provides a set of control statements for manipulating data structures. Pascal has many standard data types available such as Boolean, character, or real. The programmer can also define data structures that are more appropriate abstractions of the problem data and combine simple data structures into arrays and records.

Pascal automatically expands all integer variable values within an arithmetic expression to the length of the longest operand. In addition, all literal integer constants within an arithmetic expression are compiled as type INTEGER, while all literal real constants are compiled as type REAL.

A variety of executable statements is available to the Pascal programmer. Simple statements, such as the Empty, Assignment, Procedure Call, and the GOTO statements, perform one specific operation. Structured statements are a combination of other statements such as the compound statement that provides a framework for the main body of a program. The Pascal CASE statement is a structured statement that provides the capability of the computed GOTO of FORTRAN but does not require statement labels. Because the alternatives to be executed under each condition are embedded in the CASE statement structure, this statement reduces the need of GOTO statements in a program.

The Perkin-Elmer implementation of Pascal, which is a subset of the standard defined in the Pascal User Manual and Report by Jensen and Wirth, features:

- Syntax graphs
- Header statements
- Run time support
- Command substitution system (CSS) procedures

---

1 Jensen and Wirth, Pascal User Manual and Report, New York, Springer-Verlag, 1975.

1.4 UTILITIES

The following Perkin-Elmer utility programs can be used with OS/32:

- Linkage Editors
- Edit
- Text
- Source Updater
- Automatic Interactive Debugging System (AIDS)
- Copy
- Library Loader
- Macro Library
- Sort/Merge Level II
- Patch

1.4.1 Linkage Editors

Perkin-Elmer linkage editors are used to generate an image load module from one or more object modules. Image load modules can be tasks, sharable segments, or operating systems. External references to task common and to previously established reentrant library segments are also processed. The available linkage editors are:

- Link
- TET

Link replaces TET under OS/32 R06 and higher.

1.4.1.1 Link

Perkin-Elmer Link can build image load modules in sizes up to 16Mb. The Link tree-structured overlay feature allows automatic loading of user-specified routines into an overlay area when the routine is called during task execution. The overlay structure does not have to be defined in the source module.

1.4.1.2 Task Establisher Task (TET)

Each TET overlay must be defined completely before another overlay statement is presented in the command stream. After all overlays are defined, the task overlay area is set to the size of the largest area requested. Only one overlay area is reserved for each task, no matter how many overlay commands are entered.

1.4.2 Edit

Edit is a disk-based editor that can be used to append, alter, or save data on a line-by-line basis. Features include:

- Interactive or batch mode execution
- Global alterations of character strings
- Data deletions, additions, or insertions
- Character string searches
- Permanent file data storage
- User-specified record length, termination characters, and tab settings

1.4.3 Text

Text is used to generate, revise, and print manuals, documents, and letters. The editor is linked with Text to provide all editing capabilities. Text features include:

- Line centering
- Margin definition
- Boldface entries
- Pagination
- Right justification
- Left justification
- Underscoring
- Indention

1.4.4 Source Updater

The Source Updater is used to create and maintain source files on mass storage devices. Source updater commands enable the user to verify, modify, or list source files.

1.4.5 Automatic Interactive Debugging System (AIDS)

AIDS is a user-oriented assembly level debugging program that:

- Displays and modifies memory locations and floating point and general registers
- Prints sections of memory to a list device
- Provides the following program utilities:
  - Snapshot printouts
  - Cell/register protection
  - Trace execution
  - Breakpointing
- Provides single step execution that displays:
  - Current bias
  - Location counter
  - Task status
  - Condition code
- Converts requested data or the current open cell from one format to another

1.4.6 Copy

Copy transfers data from one device to another device and supports:

- A verify operation that guarantees the integrity of copied data
- Blocked or unblocked, labeled or unlabeled input and output tapes

1.4.7 Library Loader

Library Loader is an interactive utility that allows the operator to create object program library files on a mass storage device. Once the object program library files are created, operator commands can be used to search these files. In addition, the automatic link editing feature allows the operator to load all library programs required for any one particular task by using only one command.

1.4.8 Macro Library

The Perkin-Elmer Macro Library Utility Program provides capabilities for establishing and maintaining the system macro library and/or any user-designated macro libraries. This utility:

- Creates a new library
- Maintains an existing library
- Adds new macro definitions to a library
- Deletes macro definitions from a library
- Lists macro definitions from a library to a device file
- Prints the directory macro names of a library to a device or file
- Stores an updated library in a permanent file

1.4.9 Sort/Merge Level II

Sort/Merge Level II allows a user to reorder a file of fixed length records according to user-defined key fields; or produce a single, ordered file from two or more input files of fixed length records that presorted in identical key sequences. The main features of Sort/Merge Level II are:

- Input and output can use disk, magnetic tape, or any sequential, fixed length record device.
- Commands and parameters can be input on an interactive device.
- As many key fields as required can be specified (up to a maximum total key length of 1024 bytes).

- Keys can be of a variety of types:

  - String
  - Signed binary integer (16-, 32-, and 64-bit)
  - Signed floating point binary (32- and 64-bit)
  - Packed decimal
  - Unpacked decimal

- Ascending or descending sequence can be specified separately for each key.

- Up to four files can be merged.

- A single input file can be specified to the merge function, providing a record sequence checking facility.

- A series of input files can be sorted together as an alternative to a sequence of separate sorts followed by merge operations.

- Repeated sort and/or merge operations can be carried out without reloading the program.

1.4.10 Patch

Patch is a program development tool that allows users to add to or change object or image program versions without reassembling the source module. The capabilities provided are:

- A history feature that records all changes made

- The ability to manipulate object libraries and compound overlay files

Patch is a disk-based reentrant program that can run in either interactive or batch mode.

CHAPTER 2
TASK STRUCTURE

2.1 INTRODUCTION

The fundamental work unit of the Perkin-Elmer 32-bit operating system is the task. A task can be a single program or a main program with a number of subroutines and overlays. A total of 252 tasks can reside in the system at one time.

Each task is compiled or assembled into an object code module. From this object module, Link builds an image load module. Once the image load module is built, the task can be loaded using a LOAD command or a load function executed by another task. Tasks are identified by a taskid assigned to the task when it is loaded into the system. A group of special program development commands are available to facilitate compiling, linking, and running of tasks. These commands are described in detail in the OS/32 Multi-terminal Monitor (MTM) Reference Manual.

This chapter discusses task structure as defined by OS/32. Included are explanations of task address space, task segments, task states, and subtasking.

2.2 TASK AND MEMORY PROTECTION

User tasks (u-task) run in a protected mode. They cannot be accessed by tasks outside their boundaries. In addition, u-tasks cannot execute code in common areas or use any of the privileged instructions. The privileged instructions include all I/O instructions; e.g., OC, RH, WH, SSR, and any instruction that changes the state of the processor, such as LPSWR or EPSR.

To execute I/O instructions or change the processor state, u-tasks make requests of the operating system via a supervisor call (SVC) instruction. The relocation/protection hardware provides memory protection for u-tasks. The relocation/protection hardware and the Perkin-Elmer processors associated with it are:

- Memory access controller (MAC)
  - Model 7/32
  - Model 8/32
  - Model 3220

- Memory address translator (MAT)

  - Model 3210
  - Model 3230
  - Model 3240
  - Model 3250

This protection is transparent to u-tasks running under OS/32.

Task memory access errors are handled automatically by the operating system or by the task itself if a trap service routine exists.

2.3 USER TASK (U-TASK) ADDRESS SPACE

When a u-task is loaded into memory, relocation/protection hardware automatically relocates the task relative address to physical memory. The u-task refers to data and instructions relative to the first location in the task as if the task were loaded at location 0 in memory.

U-task address space is divided into segments. A segment is a set of contiguous program addresses starting on a 64kb boundary. A maximum of 16 segments on a MAC machine is available for each u-task. Each segment is divided into 256-byte pages. See Figure 2-1.

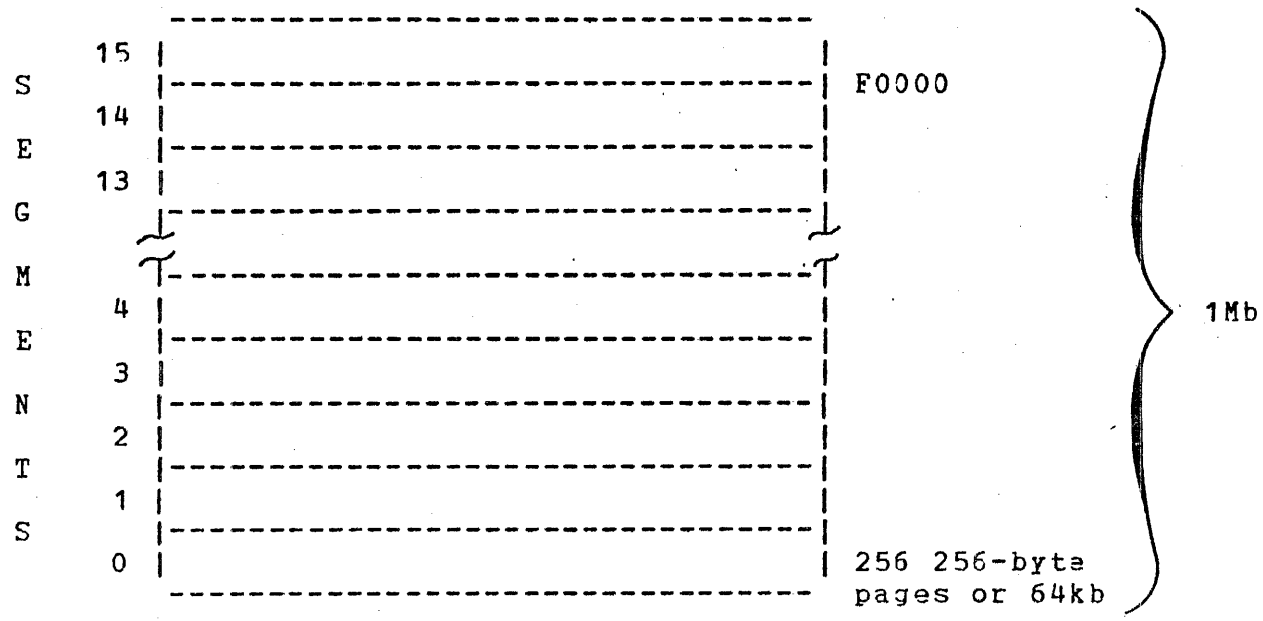


Figure 2-1 Task Address Space on a Memory Access Controller (MAC) Machine

On MAT machines a maximum of 192 segments are available for each u-task, and each of these segments is divided into 2048-byte pages. See Figure 2-2.

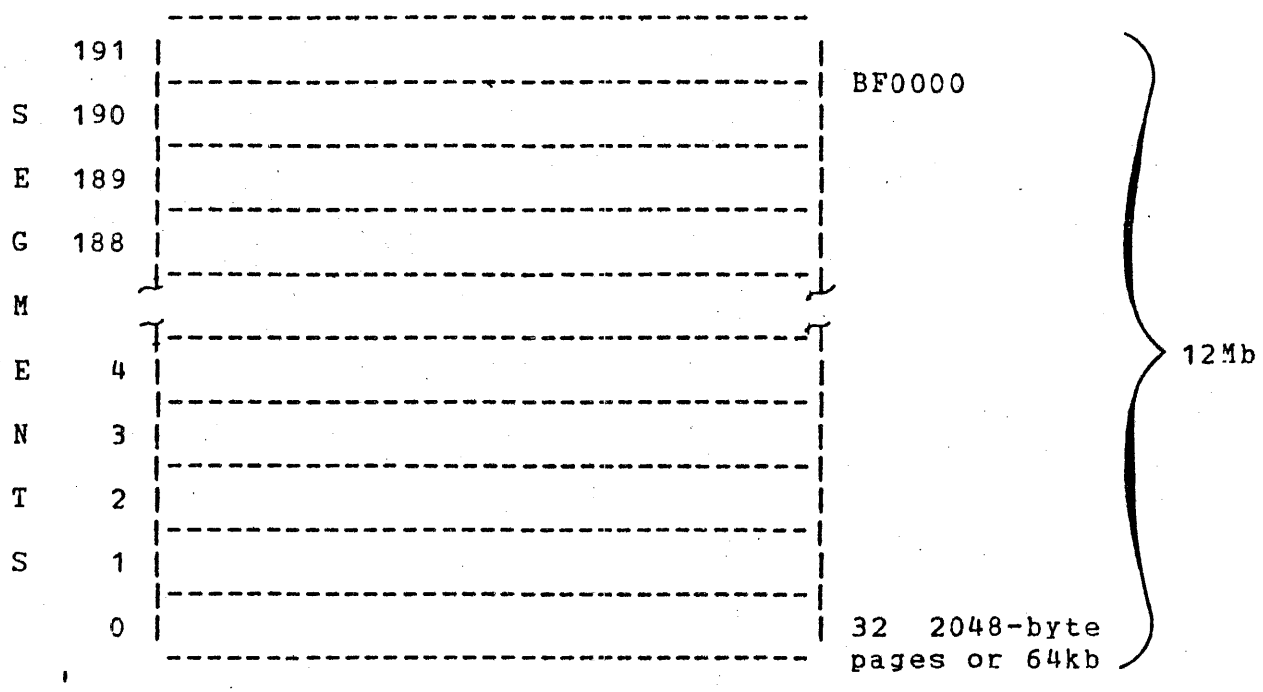


Figure 2-2 Task Address Space on a Memory Address Translator (MAT) Machine

The task address of each segment begins on a segment boundary; e.g., 00000, 10000, 20000, up to F0000 (1Mb) for MAC machines, and up to BF0000 (12Mb) for MAT machines.

2.4 TASK SEGMENT TYPES

Segments within u-task address space are classified as pure or impure. An impure segment can be written to, read from, or executed only by the task in which it resides. A pure segment contains data or instructions that can be read or executed by any task. In addition, a u-task can contain one or more optional reentrant library segments or common data areas.

2.4.1 Impure Segments

Every task must have an impure segment to hold the user program and data. This segment, which cannot be shared with any other u-task, starts with segment number 0. If the u-task occupies more than the 64kb of address space, the task is extended into one or more of the segments contiguous to the first impure segment.

The impure segment is defined by three parameters: UBOT, UTOP, and CTOP. The current values of these three parameters are available to the terminal user through the DISPLAY PARAMETERS command. UBOT always holds the starting address of the impure segment. For u-tasks this address is always Y'0'.

When a task is loaded, UTOP holds the address of the first fullword above the defined portion of the impure segment. The defined portion is the section of the impure segment that is explicitly defined in object code by Link. Some programs use an undefined portion of memory above their defined portion for dynamic storage. An example of such an undefined area is the symbol table area used by Common Assembly Language/32 (CAL/32). While this undefined area lies within the impure segment, it lies above the area to which UTOP points but below the area defined by CTOP, the address of the highest halfword in the impure segment. For a MAC machine, CTOP always contains an address that is one halfword less than a 256-byte boundary. For a MAT machine, CTOP always contains an address that is one halfword less than a 2048-byte (2kb) boundary. See Figure 2-3.

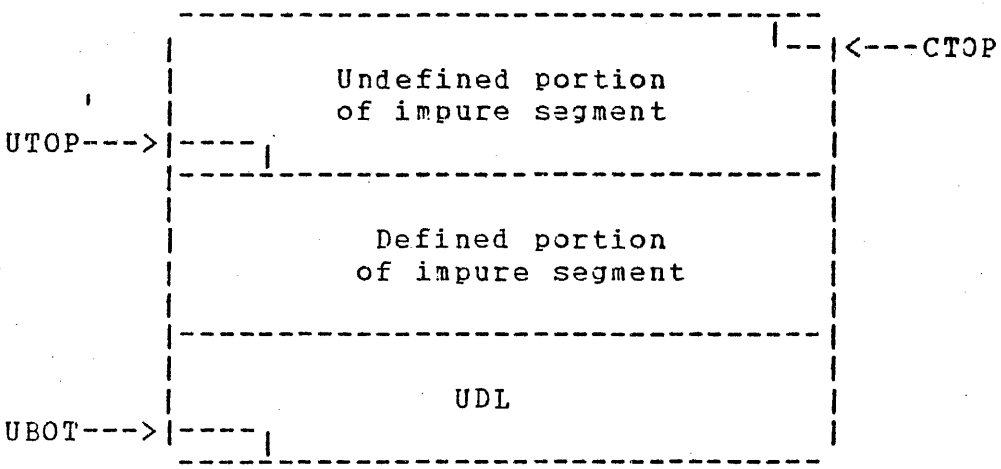


Figure 2-3 Impure Segment

Link defines the initial values for UTOP and CTOP when the u-task is built. Link increases the value of CTOP through the command:

```
OPTION WORK=(min,max)
```

This command gives the task an undefined storage area above its originally defined space. Using the LOAD command can increase the value of CTOP at load time. After a task is loaded, the

value of UTOP can be modified by a GETSTORE and a RELSTORE macro. When a resident task is restarted, the original value of UTOP is restored.

If a program uses overlays, the overlay area becomes a part of the impure segment. UTOP initially is set equal to the address of the first fullword above the overlay area. The overlay area is large enough to contain the largest program overlay. During task execution, the overlay area will have only one resident overlay. Thus, in memory, the impure segment of a task using overlays contains (beginning at the lowest location) a root section, an overlay section, and an expansion section (if used). Any expansion area follows the overlay area. See Figure 2-4.

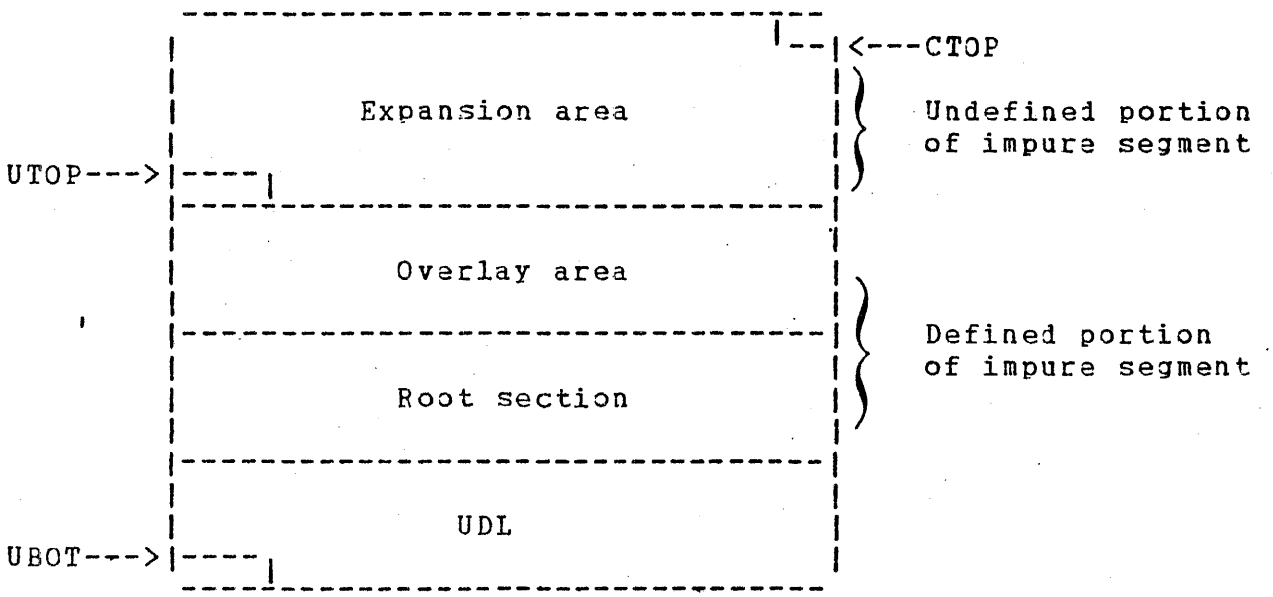


Figure 2-4 Impure Segment with Root and Overlay Area

The task code in the task impure segment is preceded by the user dedicated location (UDL) that occupies memory locations 0 through 255. The UDL contains task status word (TSW) swap areas used for communication between the operating system and the task.

2.4.2 Pure Segments

The user can optionally allocate one pure segment within a task. This segment is the shared portion of the task and uses the lowest available segment after the maximum workspace size reserved by Link. The actual segment number used is computed by Link. The size of the pure segment is limited only by the total number of segments within a task. The pure segment contains reentrant code that can be shared by several tasks concurrently.

The code for this segment is assembled with the CAL/32 option PURE. A task cannot modify any location within its pure segment. The relocation/protection hardware prevents other tasks from writing to the pure segments.

The Link command:

```
OPTION WORK=(min,max)
```

builds the pure segment above the maximum workspace area. See Figure 2-5.

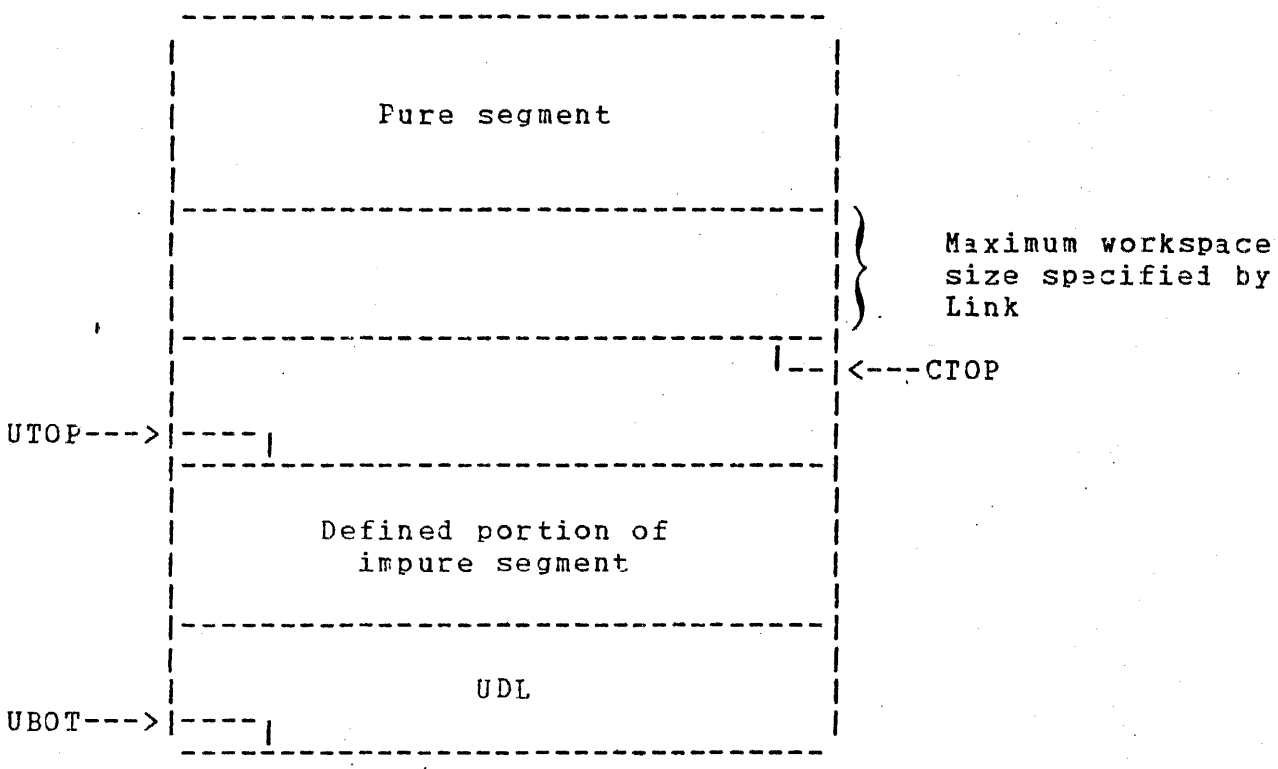


Figure 2-5 Pure Segment

2.5 IMAGE LIBRARIES

Image libraries are segments built by Link outside the u-task address space. An example is the FORTRAN run time library (RTL). When a task that is to use an image library is built, Link sets linkages from the task to the appropriate library segments. A maximum of 15 library segments can be used by a task running on a MAC machine; 191 libraries can be used by a task on a MAP

machine. Once these linkages are set, a u-task can read from or execute the image library segments. Relocation/protection hardware prevents the task from writing to these segments.

Image library segments are allocated to and deleted from memory by the operator with these commands:

- LOAD.LIB
- LOAD.SEG
- REMOVE.SEG

Image library segments also can be loaded automatically at task load time.

2.6 COMMON SEGMENTS

Memory areas can be allocated for storing data that all system tasks can read from or write to. These areas are called common.

Local common segments located in local memory can be allocated, defined, and deleted by the operator with these commands:

- TCOM
- LOAD.TCM
- LOAD.SEG
- REMOVE.SEG

Data areas can also be allocated for storing information that can be read or written to by all tasks under the control of two or more Perkin-Elmer processors. These areas, called global common, located in global memory, are defined and allocated by the system generation (sysgen) TCOM command.

To the u-task, all common segments appear as a task common. A task common is a data area within the task impure segment that can be written to or read from only by the other segments within the task where it resides. An example of a task common area is that area of the impure segment used by the FORTRAN COMMON statement to store variables.

Link initializes common segments. Because common areas can cross segment boundaries, the size of a single common area is limited only by the amount of memory available. Link can designate a common as write-protected, allowing only one task to modify the segment; the remaining tasks can only read that area. Relocation/protection hardware prevents tasks from executing code in common areas.

The user defines the name of a global or local common. This name must correspond to predefined names given to these areas by the operator or at sysgen time. If a user-defined name of a common corresponds to a predefined name, the user-defined common is automatically loaded in the respective local or global common. If the name of a user-defined common does not correspond to a predefined name, the user-defined common becomes part of the task impure segment and is treated as a task common.

Link is given the names of the predefined common areas. When a reference to a predefined common is encountered in the object code of the u-task, Link sets the appropriate linkage from the u-task to the common.

2.7 TASK STATES

Tasks can be resident or nonresident. A nonresident task is removed from memory after execution; a resident task remains in memory after execution. A task can be classified as resident by Link or at run time.

A task in memory can be in any of these states:

- Current
- Ready
- Wait
- Rolled

A task is in the current state while it is executing instructions. Only one task can be in the current state at any given time. All other tasks in memory are in one of the other states.

A task is in the ready state when it is eligible to be dispatched; i.e., there are no obstacles to prevent it from becoming current.

A task is in the wait state when it cannot become ready until some specific event occurs. Table 2-1 lists the wait states and their meanings.

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
Time 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 queue wait	Wait for an I/O queue to be freed when task reaches its QIO limit
Accounting wait	Counters overflowed; task waiting for accounting facility to collect accounting data and remove wait
Intercept wait	Wait for an 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 dormant state and remains there until execution is initiated. When a resident task goes to end of task, it reenters the dormant state

A task is in the rolled state when its task impure segment has been written to a direct access device. A task becomes rollable when it is specified as a rollable, nonresident task by Link. A rollable task is rolled out when a higher priority task requires its memory segment. It is rolled in when it becomes the highest priority rolled task and sufficient memory is available to accommodate it.

2.8 SUBTASKING

A subtasking facility allows one task (monitor task) to start, cancel, delete, and monitor the progress of the task it controls. The monitor task can set starting options and make logical unit (lu) assignments on behalf of the subtasks. The monitor task, using standard SVC 6 calls, can also control the task environment of its subtasks.

The operating system informs the monitor task that the subtask:

- is paused,
- has gone to end of task,
- is suspended,
- is released,
- is rolled out/rolled in,
- has been started, or
- has inherited subtasks from one of its subtasks.

To assign a task as a monitor, use SVC 6 to specify the subtask report option. The number of subtasks that report to a single monitor is unlimited. When all subtasks of a monitor go to end of task, the monitor is no longer referred to as a monitor. The normal SVC 6 functions that provide intertask communication and control are equally applied between monitor/subtask and subtask/monitor.

By adding an entry to the monitor task queue and giving the monitor a task trap, the operating system informs a monitor of a subtask state change. Bit 15 of the current task status word (TSW) enables task queue entries for subtask state changes. The monitor services the subtask report when trap service is enabled.

2.8.1 Tasks Loaded from Multi-Terminal Monitor (MTM) Terminals

All tasks loaded and started from a user terminal execute as subtasks of MTM. Tasks executing under MTM will run at a maximum priority of at least one less than the priority of MTM.

Both interactive and batch processing are supported by MTM. Up to 64 interactive tasks can be executed concurrently, one from each terminal. A terminal initiating an interactive task can be used to submit multiple batch jobs. Batch jobs are queued by MTM. The number of batch jobs that can execute concurrently is specified by the operator during MTM system start up.

CHAPTER 3
TRAPS

3.1 INTRODUCTION

When certain events occur during the execution of a user task (u-task), the task can take traps to handle them. A trap suspends task execution and executes a special routine to handle the event. This routine can be a trap service routine or a task event service routine. When the event has been serviced, the u-task resumes normal execution.

The user dedicated location (UDL) contains the dedicated locations required to use the trap service routine. For each type of trap that is supported, the UDL contains locations for holding the old and new task status words (TSW) affected by the trap. When a trap occurs, the current TSW is saved in the old TSW location and the TSW in the new TSW location is loaded to become the current TSW. This TSW must point to the trap service routine written to service the type of trap caused by the event. After the trap service routine terminates, the contents of the old TSW location of the UDL are loaded into the new TSW location, and the u-task resumes normal execution. No registers are saved as a part of the TSW swap that causes a trap service routine to be initiated. It is the responsibility of the trap service routine to save any registers it requires. Events that can be serviced by the trap service routine are:

- Power restoration
- Arithmetic faults
- Supervisor call 14 (SVC 14) execution
- Memory access faults
- Illegal instruction execution
- Data format/alignment faults
- The addition of items to a task queue

Other events are handled through the task event trap service routine. This routine differs from the trap service routine described above in that the address of the routine is stored in the parameter block of the SVC that caused the trap rather than in the task UDL.

3.2 USER DEDICATED LOCATION (UDL)

The UDL is an area occupying locations 0 through 255 (X'FF') in each task impure memory space, preceding task code. It contains TSW swap areas and other data areas for communication between the operating system and the task. The queue entry and new TSW fields in the UDL are used only if the corresponding bits in the TSW are enabled. In Figure 3-1, the names in parentheses are the symbolic names of the fields as defined in the UDL data structure. A $UDL macro call generates the structures and equates for the UDL data structure. Figure 3-1 depicts the UDL structure. All fields are described following the figure.

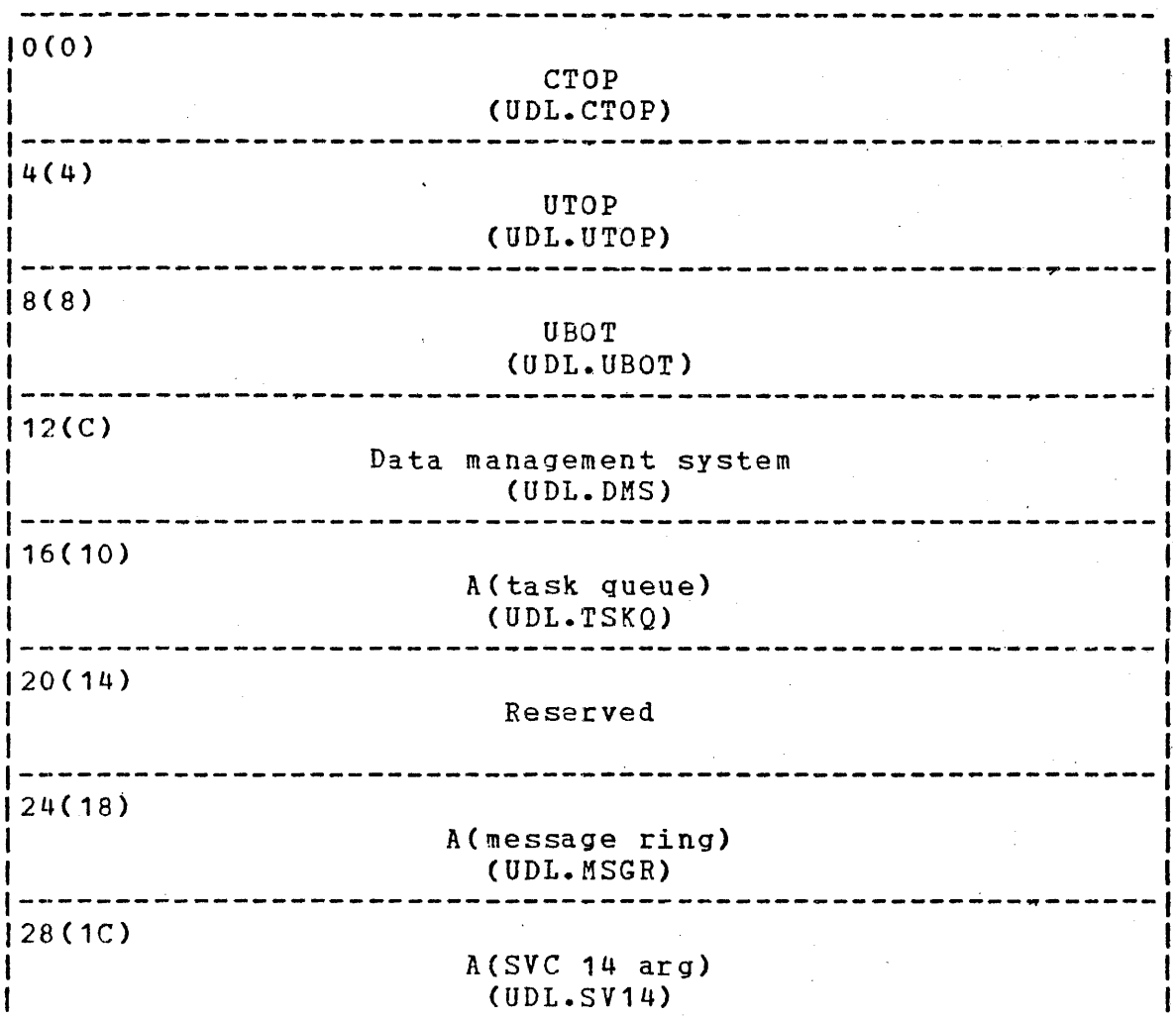


Figure 3-1 User Dedicated Location (UDL) Structure

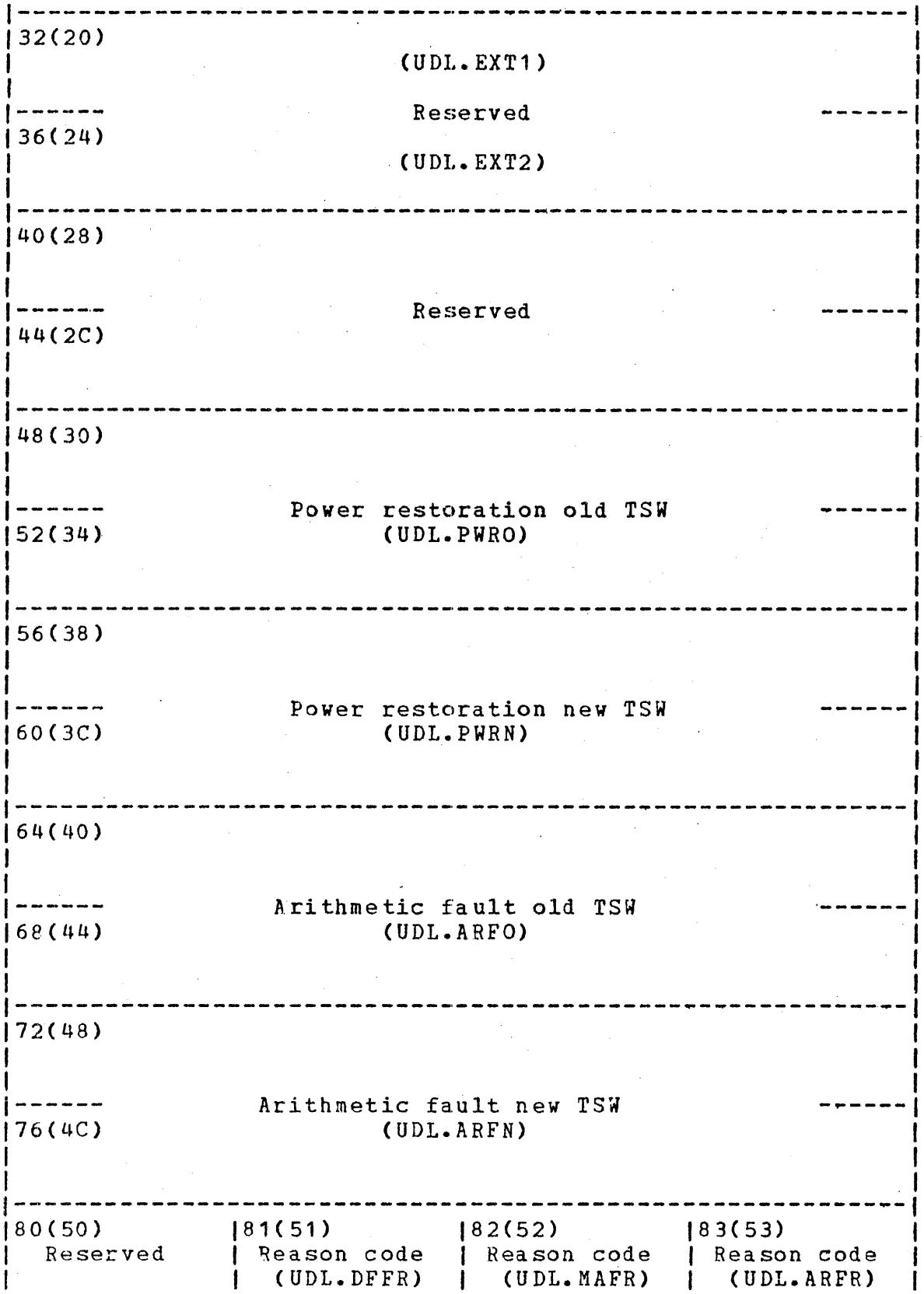


Figure 3-1 User Dedicated Location (UDL) Structure (Continued)

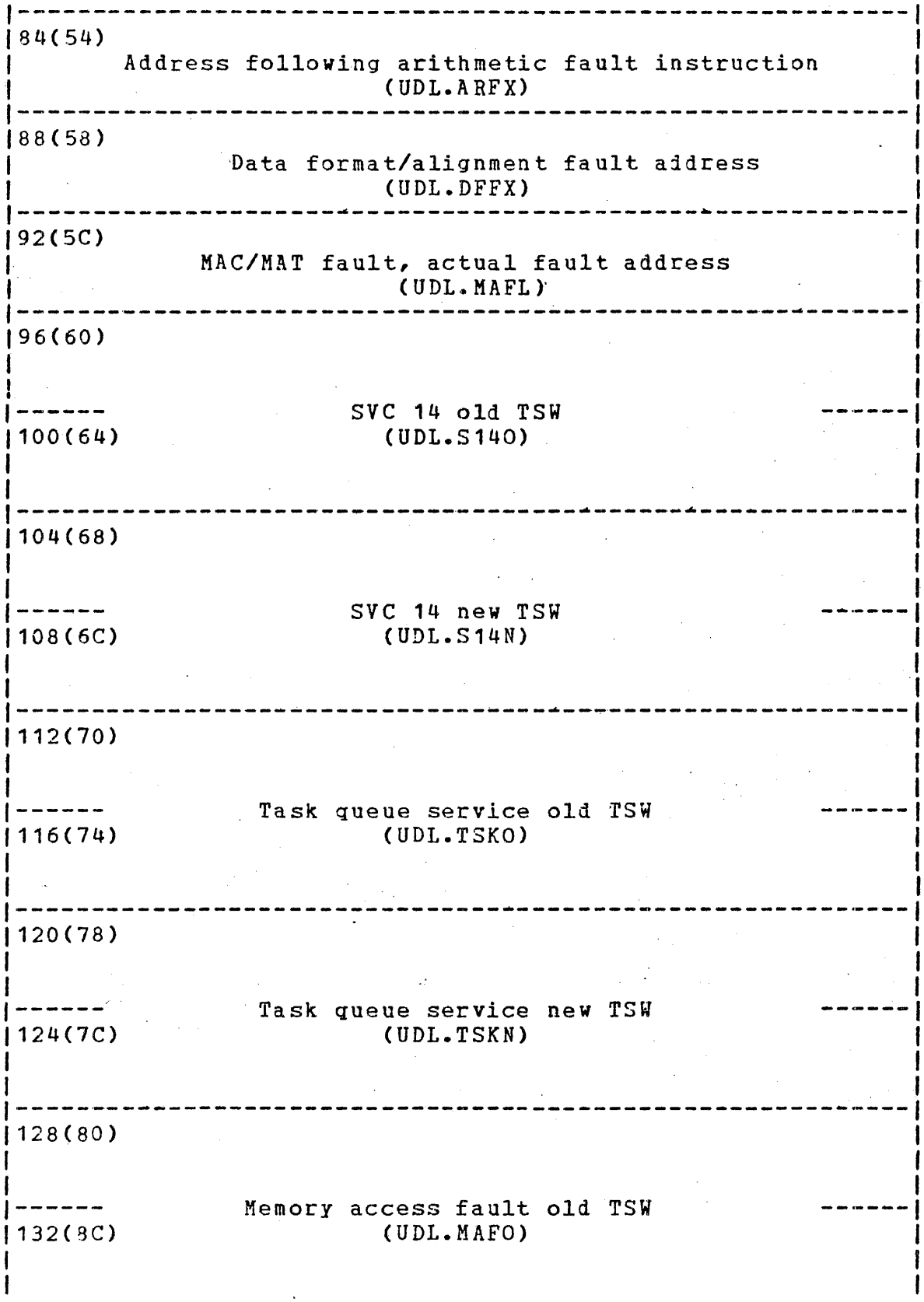


Figure 3-1 User Dedicated Location (UDL) Structure (Continued)

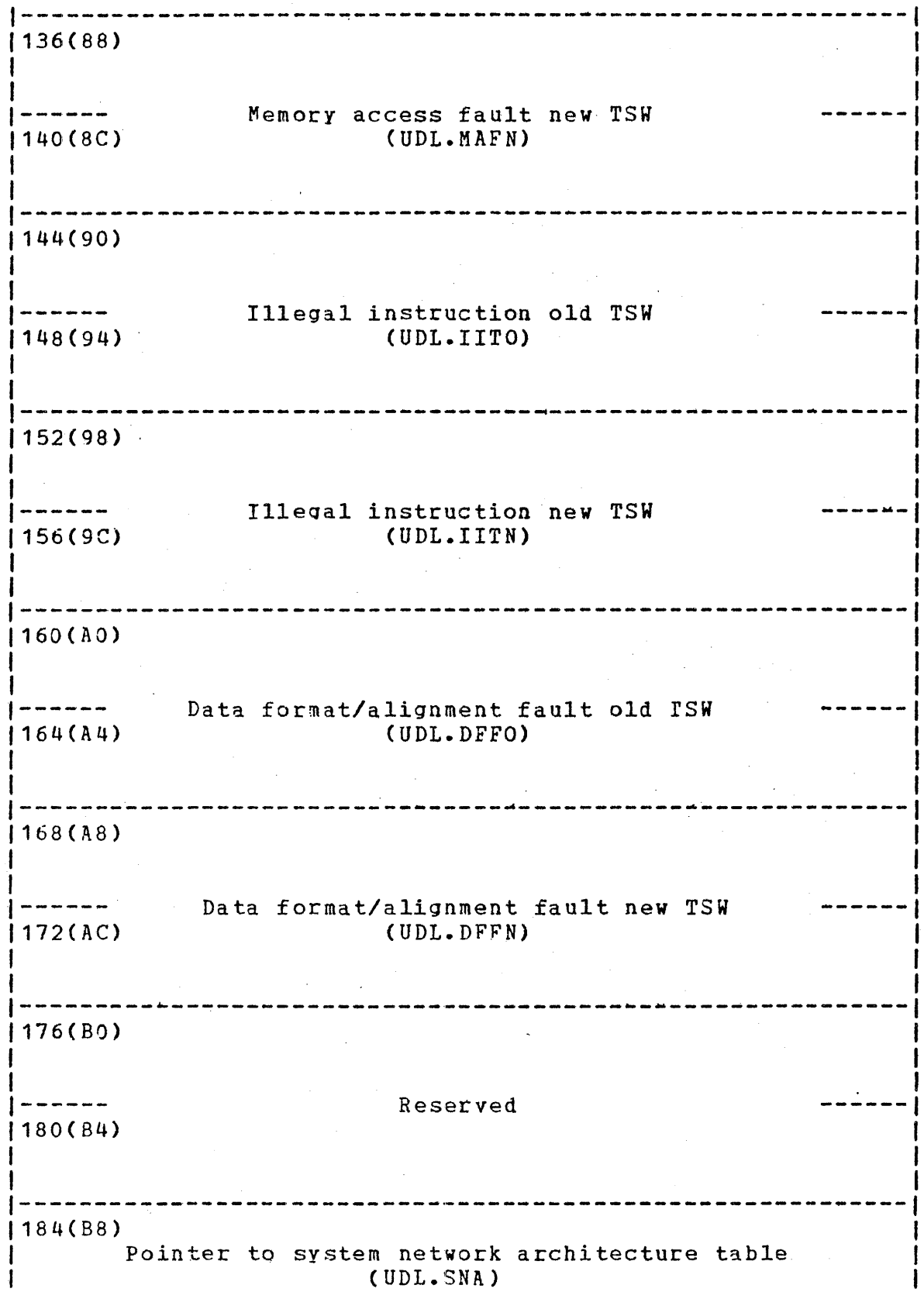


Figure 3-1 User Dedicated Location (UDL) Structure (Continued)

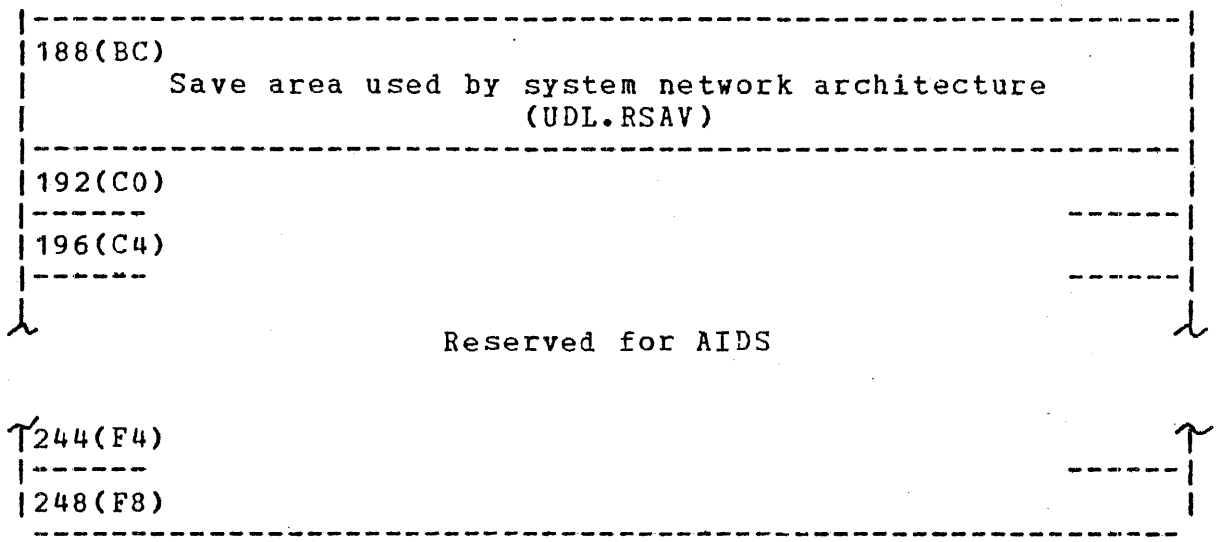


Figure 3-1 User Dedicated Location (UDL) Structure (Continued)

Fields:

CTOP (UDL.CTOP) — is the top of the impure segment. After a fetch pointer (FETPTR) macro call, CTOP contains the program address of the highest halfword in the task impure segment. The value of CTOP which is defined by Link can be overridden at load time.

UTOP (UDL.UTOP) — is the user top. After a fetch pointer (FETPTR) macro call is issued, UTOP contains the program address of the first fullword following the defined portion of the impure segment. The GETSTORE and RELSTORE macro calls manipulate the value of UTOP, which cannot exceed the value of CTOP by more than two bytes.

UBOT (UDL.UBOT) — is the user bottom. After a fetch pointer (FETPTR) macro call, UBOT contains the program address of the bottom of the user program. For user tasks, this value is 0.

Data management system (UDL.DMS) — is the field used by the data management system and must contain zeros.

A(task queue) (UDL.TSKQ) — is the field into which the user stores the address of the task queue. If the content of of this field is zero, no task queue entries can be made, even if the TSW task queue entry bit is set.

Reserved	is a field reserved for future use that must contain zeros.
A(message ring) (UDL.MSGR)	is the field into which the user stores the address of a 76-byte storage area before receiving any messages. This storage area must be aligned on a fullword boundary. If the content of this field is zero, no message can be received, even if the TSW task queue entry on the message bit is set.
A(SVC 14 arg) (UDL.SV14)	is the field where the operating system stores the effective address of the SVC 14 argument when an SVC 14 trap occurs.
Reserved (UDL.EXT1) (UDL.EXT2)	is the field reserved for system use that must contain zeros.
Reserved	is a field reserved for future use that must contain zeros.
Power restoration old TSW (UDL.PWRO)	is the field where the operating system stores the task's current TSW when a power restoration trap occurs.
Power restoration new TSW (UDL.PWRN)	is the field into which the user stores the TSW to be loaded as the current TSW when a power restoration trap occurs. The location counter portion of this TSW should contain the address of a power restoration trap service routine.
Arithmetic fault old TSW (UDL.ARFO)	is the field into which the operating system stores the task's current TSW when an arithmetic fault trap occurs. For the Perkin-Elmer 3200 Series Processors, the location counter portion of this TSW contains the address of the faulting instruction. For 7/32 and 8/32 processors, the location counter portion of this TSW contains the address of the next instruction.
Arithmetic fault new TSW (UDL.ARFN)	is the field into which the user stores the TSW to be loaded as the current TSW when an arithmetic fault trap occurs. The location counter portion of this TSW contains the address of an arithmetic fault trap service routine.
Reserved	is a field reserved for future use that must contain zeros.
Data format/ alignment fault reason code (UDL.DFFR)	is the field into which the operating system stores a reason code indicating the type of data format/alignment fault when a data format/alignment fault trap occurs.

Memory access fault reason code (UDL.MAFR)	is the field into which the operating system stores a reason code indicating the type of memory access fault when a memory access fault trap occurs. This field applies only to the Perkin-Elmer 3200 Series Processors.
Arithmetic fault reason code (UFL.ARFR)	is the field into which the operating system stores a reason code indicating the type of arithmetic fault when an arithmetic fault trap occurs. This field applies only to the Perkin-Elmer 3200 Series Processors.
Address following arithmetic fault instruction (UDL.ARFX)	is the field into which the operating system stores the address of the instruction following the instruction that resulted in an arithmetic fault trap. This field applies only to the Perkin-ELmer 3200 Series Processor.
Data format/ alignment fault address (UDL.DFFX)	is the field into which the operating system stores the address of the location in memory referenced by the instruction which caused the alignment fault or the data format fault trap. This field applies only to the Perkin-Elmer 3200 Series Processors.
MAC/MAT fault, actual fault address (UDL.MAFL)	is the field into which the operating system stores the address of the location that caused a memory access fault trap. The address can be the effective address of data or the instruction address depending on the fault type indicated in the memory access fault reason code field (UDL.MAFR). This field applies only to the Perkin-Elmer 3200 Series Processors.
SVC 14 old TSW (UDL.S14O)	is the field into which the operating system stores the current TSW when an SVC 14 trap occurs. If SVC 14 traps are disabled in the TSW, the execution of an SVC 14 is illegal.
SVC 14 new TSW (UDL.S14N)	is the field into which the user stores the TSW to be loaded as the current TSW when an SVC 14 trap occurs. The location counter portion of this TSW contains the address of an SVC 14 trap service routine.
Task queue service old TSW (UDL.TSKO)	is the field into which the operating system stores the current TSW when a task queue service trap occurs.

Task queue service new TSW (UDL.TSKN)	is the field into which the user stores the TSW to be loaded as the current TSW when a task queue service trap occurs. The location counter portion of this TSW contains the address of the task queue trap service routine.
Memory access fault old TSW (UDL.MAFO)	is the field into which the operating system stores the current TSW when a memory access fault trap occurs.
Memory access fault new TSW (UDL.MAFN)	is the field into which the user stores the TSW to be loaded as the current TSW when a memory access fault trap occurs. The location counter portion of this TSW contains the address of a memory access fault trap service routine.
Illegal instruction old TSW (UDL.IITO)	is the field into which the operating system stores the TSW to be loaded as the current TSW when an illegal instruction trap occurs.
Illegal instruction new TSW (UDL.IITN)	is the field into which the user stores the TSW to be loaded as the current TSW when an illegal instruction trap occurs. The location counter portion of this TSW contains the address of an illegal instruction trap service routine.
Data format/ alignment fault old TSW (UDL.DFFO)	is the field into which the operating system stores the task's current TSW when a data format or alignment fault trap occurs. This field applies only to the Perkin-Elmer 3200 Series Processors.
Data format/ alignment fault new TSW (UDL.DFFN)	is the field into which the user stores the TSW to be loaded as the current TSW when a data format or alignment fault trap occurs. The location counter portion of this TSW contains the address of the data format fault or alignment fault trap service routine. This field applies only to the Perkin-Elmer 3200 Series Processors.
Reserved	is a field reserved for future use that must contain zeros.
Pointer to system network architecture table (UDL.SNA)	is the field into which the operating system stores the address of the system network architecture table.

Save area used by system network architecture (UDL.RSAV)	is a field reserved for internal use by data communications network software.
Reserved for AIDS	is a field used by OS/32 AIDS that must contain zeros.

User-supplied fields in the UDL; e.g, all new TSW fields and the task queue and message ring address fields, can be assembled as constants or loaded during the task initialization phase. Link builds tasks from program object code. Tasks containing a user-assembled UDL are specified by the Link command:

```
OPTION ABSOLUTE=0
```

This command specifies that a UDL exists in the program and that storage should not be reserved for it. If a task does not contain a user-assembled UDL and the ABSOLUTE parameter is omitted in the OPTION command, Link reserves 256 bytes of storage for the UDL at the beginning of the image load module.

3.3 TASK STATUS WORD (TSW)

The TSW describes the task state at any time with respect to user-controlled interaction with the operating system. TSW also acts as a location counter for the task and enables and disables the various task traps and additions to the task queue. A $TSW macro call generates the TSW structures and equates. Figure 3-2 depicts the TSW. All TSW fields are described following the figure.

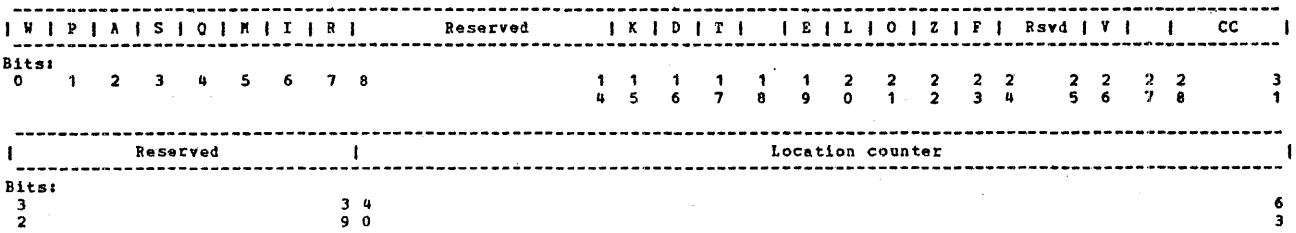


Figure 3-2 Task Status Word (TSW)

Fields:

0(TSW.WTM)	is the trap wait bit. If enabled, the task is suspended until a trap occurs.
1(TSW.PWRM)	is the power restoration trap bit. If enabled, the task receives a trap when power is restored following a power failure. If disabled, the task is paused.
2(TSW.AFM)	is the arithmetic fault trap bit. If enabled, the task receives a trap when an arithmetic fault occurs. If disabled, the task is paused and a message is displayed on the user console.
3(TSW.S14M)	is the SVC 14 trap bit. If enabled, the task can use SVC 14 and receive a trap when an SVC 14 is executed. If disabled, execution of an SVC 14 is illegal. Because SVC 14 is used by AIDS to set breakpoints, it must not be used in a task being debugged with OS/32 AIDS.
4(TSW.TSKM)	is the task queue service trap bit. If enabled, the task receives a trap when an item is added to the task queue or when a TSW with this bit set is loaded and the task queue contains items. If disabled, no trap occurs when item is added to task queue.
5(TSW.MAFM)	is the memory access fault trap bit. If enabled, the task receives a trap when it attempts to access memory outside its task space. If disabled, the task is paused and a message is displayed on the user console.
6(TSW.IITM)	is the illegal instruction trap bit. If enabled, the task receives a trap when it executes an illegal instruction. If disabled, the task is paused and a message is displayed on the user console.
7(TSW.DFFM)	is the data format fault and alignment fault trap bit. If enabled, the task receives a trap when it executes an instruction that causes a data format or alignment fault. If disabled, the task is paused and a message is displayed on the user console.
8-14(Reserved)	is a field reserved for future use that must contain zeros.

15(TSW.SUQM) is the subtask queue entry bit. If enabled, the task receives items on its queue specifying a subtask state change. If disabled, no items are entered on the task queue when a subtask state change occurs and the notification of change is lost.

16(TSW.DIQM) is the task queue entry on device interrupt bit. If enabled, an item is entered on the task queue when a trap is received from an I/O device. If disabled, no items are entered on the task queue when a device trap occurs.

17(TSW.TCM) is the task queue entry on SVC 6 queue parameter call bit. If enabled, an item is entered on the task queue when a task issues an SVC 6 queue parameter option. If disabled, the call to the task is rejected.

18(Reserved) is a field reserved for future use that must contain zeros.

19(TSW.PMM) is the task queue entry on task message bit. If enabled, the task can receive a message from another task or the operator. The task queue receives the address of the message buffer. If disabled, no message is sent.

20(TSW.LODM) is the task queue entry on completion of a load and proceed operation bit. If enabled, the task queue receives an item specifying the parameter block address of the SVC 6 load call when the load is completed. If disabled, no item is entered on the task queue when a load and proceed operation is completed.

21(TSW.IOM) is the task queue entry on I/O completion bit. If enabled, the task queue receives an item specifying the address of the SVC 1 parameter block when an I/O proceed call is completed. If disabled, no item is entered on the task queue when an I/O operation is completed.

22(TSW.TMCM) is the task queue entry on timeout completion bit. If enabled, the task queue receives an item indicating a specified time interval has elapsed. If disabled, the task is not notified that a time interval has elapsed.

23(TSW.ITM)	is the task queue entry on an SVC 15 buffer transfer, command execution, termination, or halt I/O bit. This function is supported only by the data communicaions subsystem. If disabled, no item is entered on the task queue when one of the specified communications operations is completed.
24-25 (Reserved)	is a field reserved for system use that must contain zeros.
26(TSW.TESM)	is the task event trap bit. If enabled, the task can receive task events. If disabled, all task events are queued until a new TSW is loaded with the task event trap enabled.
27(Reserved)	is a field reserved for future use that must contain zeros.
28-31(CC)	is the current condition code contained in the program status word (PSW).
32-39 (Reserved)	is a field reserved for future use that must contain zeros.
40-63(LOC)	is the location counter contained in the PSW.

The task always has a current TSW. The initial TSW or the current TSW when the task is loaded is set by Link. If Link is not requested to set an initial TSW, the ititial TSW defaults to all zeros (no traps or queue entries enabled). The initial location counter defaults to the starting address, if a transfer address is specified at assembly time, or to the start of the impure segment, if a transfer address is unspecified.

A task can change its current TSW at any time by issuing an LTSW macro call. The TSW must be loaded in an 8-byte area aligned on a fullword boundary. The first fullword of the TSW contains the status, and the second fullword contains the location counter. Following an LTSW macro call, the task resumes execution at the location specified in the loaded TSW. If only the status of the current TSW is to be changed, a value of zero should be specified in the location counter portion of the new TSW. In this case, execution resumes with the instruction following the macro call.

3.4 TASK TRAP SERVICE ROUTINE

There are seven types of task traps that can be handled by a task trap service routine:

- Power restoration
- Arithmetic fault

- SVC 14
- Memory access fault
- Illegal instruction
- Data format/alignment fault
- Task queue service

3.4.1 Power Restoration Trap

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 location counter 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 call to load the TSW stored in the UDL.PWRO field as the current TSW.

3.4.2 Arithmetic Fault Trap

An arithmetic fault trap occurs when one of the events listed in Table 3-1 occurs.

TABLE 3-1 ARITHMETIC FAULT TRAP-CAUSING EVENTS

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

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 location counter 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 the PSW. Table 3-2 shows actions taken when combinations of different options are specified. The trap service routine exits by issuing an LTSW macro call to load the TSW stored in the UDL.ARFO field as the current TSW.

TABLE 3-2 ARITHMETIC FAULT TRAP ACTION

ARITHMETIC FAULT BIT SETTING IN TSW	LINK ARITHMETIC FAULT OPTION	ACTION TAKEN
1	AFCONT	Trap occurs
1	AFPAUSE	Message/paused
0	AFCONT	Message/continued
0	AFPAUSE	Message/paused

3.4.3 SVC 14 Trap

SVC 14 is a user-defined SVC, and its function is performed in the SVC 14 trap service routine. An SVC 14 trap occurs when an SVC 14 is executed with the TSW.S14M bit set. The current TSW is stored in the UDL.S140 field, the new TSW stored in the UDL.S14N field is loaded and becomes the current TSW, and the address of the SVC 14 argument is stored in the UDL.SV14 field. The address of the SVC 14 argument can be used by the trap service routine. This trap service routine exits by issuing an LTSW macro call to load the TSW stored in the UDL.S140 field as the current TSW.

AIDS also uses SVC 14; therefore, SVC 14 must not be used by a task that is to be debugged by AIDS.

3.4.4 Memory Access Fault Trap

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

TABLE 3-3 MEMORY ACCESS FAULT TRAP-CAUSING EVENTS

PROCESSOR	EVENT	REASON CODE
3220	SVC address error	X'00'
	Execute protect violation	X'01'
	Write/interrupt protect violation	X'02'
	Reserved	X'03'
	Write protect violation	X'04'
	Reserved	X'05'
	Reserved	X'06'
	Reserved	X'07'
	Segment number not present	X'08'
	Reserved	X'09'
	Program address is greater than segment limit fault (SLF)	X'0A'
3240	Reserved	X'00'
	Execute protect violation	X'01'
	Write protect violation	X'02'
	Read protect violation	X'03'
	Access level fault	X'04'
	Segment limit fault	X'05'
	Nonpresent segment fault	X'06'
	Shared segment table (SST) size exceeded	X'07'
	Private segment table (PST) size exceeded	X'08'

All protection violations are detected by the relocation/protection hardware.

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, the faulting instruction address is stored in the UDL.MAFL field, and a reason code is stored in the UDL.MAFR field. The new TSW location counter should contain the address of the memory access fault trap service routine. This trap service routine exits by issuing an LTSW macro call to load the TSW stored in the UDL.MAFO field as the current TSW.

3.4.5 Illegal Instruction Trap

An illegal instruction trap occurs after a 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

location counter 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.

3.4.6 Data Format/Alignment Fault Trap

A data format or alignment fault trap results when one of the events listed in Table 3-4 occurs.

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

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

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 new TSW 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 location counter contains the address of the data format or alignment fault trap service routine. This trap service routine exits by issuing an LTSW macro call to load the TSW stored in the UDL.DFFO field as the current TSW.

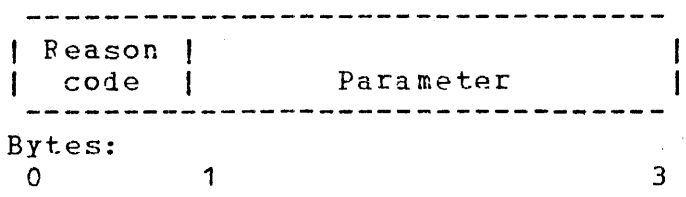
3.4.7 Task Queue Service Traps

A task queue service trap results when one of the events listed in Table 3-5 occurs.

TABLE 3-5 TASK QUEUE SERVICE
TRAP-CAUSING EVENTS

EVENT	REASON CODE
Device interrupt	X'00'
Queue parameter	X'01'
Subtask state changes	X'02'
Reserved	X'03'
Reserved	X'04'
Reserved	X'05'
Message received	X'06'
Load and proceed completion	X'07'
I/O proceed completion	X'08'
Timer termination	X'09'
SVC 15 command execution	X'0A'
SVC 15 buffer transfer	X'0B'
SVC 15 termination	X'0C'
SVC 15 halt I/O	X'0D'
ZDLC buffer input	X'0E'
ZDLC buffer output	X'0F'
ZDLC error condition	X'10'
ZDLC buffer error	X'11'
Reserved	X'12'
Reserved	X'13'
Reserved	X'14'
Reserved	X'15'
Reserved	X'16'
Reserved	X'17'
EMT3270 unsolicited input	X'18'
EMT3270 unrequested disconnect	X'19'
EMT3270 switched line connect timeout	X'1A'

Except for the subtask change event, all items added to the task queue are four bytes long and have the following format:



Fields:

Reason code — is a 1-byte hexadecimal number specifying the reason why the trap occurred. See Table 3-5.

Parameter — is a 3-byte parameter specifying additional information about the particular item added to the task queue. See Table 3-6.

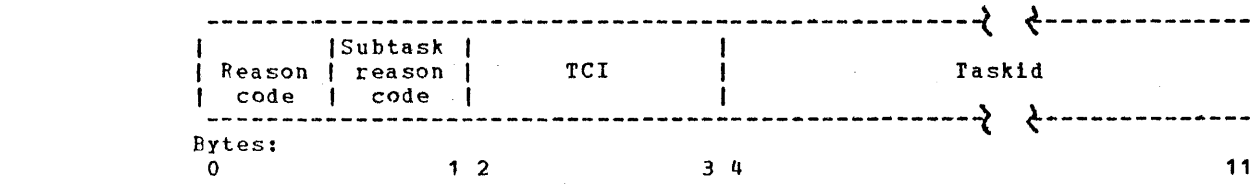
TABLE 3-6 PARAMETERS ENTERED ON THE TASK QUEUE

EVENT	PARAMETER
Device interrupt	Associated with device
Queue	Specified by sending task
Subtask state changes	Type of state change
Message received	A(message ring)
Load and proceed completion	A(SVC 6 parameter block)
I/O proceed completion	A(SVC 1 parameter block)
Timer termination	Specified in call
SVC 15 command	A(SVC 15 parameter block)
SVC 15 buffer	A(SVC 15 parameter block)
SVC 15 termination	A(SVC 15 parameter block)
SVC 15 halt I/O	A(SVC 15 parameter block)
ZDLC read done	A(UDR list)
ZDLC write done	A(UDW list)
ZDLC general error	A(information block)
ZDLC buffer exhaustion error	A(UQR list)
EMT3270 unsolicited input	
EMT3270 unrequested disconnect	
EMT3270 switched line connect timeout	

NOTE

For more information, see the OS/32 Bit Synchronous Communications Reference Manual.

Subtask items in the task queue are three fullwords long. Using three add to the bottom of the list (ABL) instructions, the operating system adds items to the bottom of task queue. The three fullwords form a 12-byte entry as follows:



Fields:

Reason code	is a 1-byte field indicating a subtask state change occurred.
Subtask reason code	is the 1-byte field that defines the particular subtask state change that occurred. See Table 3-7 for possible subtask reason codes.
TCI	is a 2-byte field that provides additional information specific to a subtask state change.
Taskid	is an 8-byte field that indicates the name of the subtask.

TABLE 3-7 SUBTASK REASON CODES (RC) AND CORRESPONDING STATE CHANGES

SUBTASK RC	SUBTASK STATE CHANGE
0	End of task; bytes 2 and 3 are binary end of task codes
1	Paused
2	Continued
3	Suspended
4	Released
5	Rolled out
6	Rolled in
7	Started by a task other than the monitor

When a task queue service trap occurs; i.e., an item is contained in the task queue and the bit is set, the current TSW is stored in the UDL.TSKO field, and the new TSW is loaded in the UDL.TSKN to become the current TSW. The new TSW location counter contains the address of the trap service routine. The trap service routine must issue a remove from top of list (RTL) instruction to

remove an item from the task queue. The item in the queue then can be examined to determine the reason the trap occurred, and appropriate action can be taken. This trap service routine exits by issuing an LTSW macro call to load the TSW stored in the UDL.TSKO as the current TSW. If additional items are on the task queue when the old TSW becomes the current TSW, a trap occurs immediately.

OS/32 also allows a task to receive a trap from external trap-generating devices. The 8-line interrupt module driver can add an item to a task queue in response to a device interrupt. If task queue service is enabled, the addition to the task queue can cause the task to take a trap. Currently, the only Perkin-Elmer driver that supports trap-generating device functions is the 8-line interrupt module driver. Users can write their own trap-generating device drivers. OS/32 provides the following functions for handling trap-generating devices.

- Connect - attach a trap-generating device to a task
- Thaw - enable interrupt on a trap-generating device
- Sint - simulate an interrupt on a trap-generating device (addition to Instrument Society of America (ISA) standards)
- Freeze - disable interrupts on a trap-generating device
- Unconnect - detach a trap-generating device from a task

These functions implement the entire ISA proposed standards for process control.

3.5 TASK EVENT SERVICE ROUTINE

Events that cause task event traps to occur are always associated with previously issued SVCs. The address of the routine that services the task event trap is stored in the parameter block of the SVC that generates the trap.

To take a task event trap, a task must have the TSW.TESM bit in the TSW enabled. If the TSW bit is not set, the task event trap will be queued until a TSW is loaded with this bit set. In addition, a task cannot take a task event trap while executing a service routine for a previous task event trap. Task queue traps and task event traps that occur during execution of a task event trap service routine are queued until the task issues the TEXIT macro to exit from the routine.

During execution, the task event service routine can receive data that is in register 0, 1, or 2 that pertains to the trap. The data contained in these registers before the trap was taken will be lost unless the TEQSAVE option is specified during LINK. Specifying the PARTIAL parameter for TEQSAVE allows TEQSAVE to save the contents of registers 0, 1, and/or 2 that pertain to the trap and to restore those registers after the task exits from the routine. Specifying the ALL parameter allows all registers to be saved and restored; no registers are saved if the NONE parameter is chosen.

CHAPTER 4
SYSTEM MACROS AND STRUCTURES

4.1 INTRODUCTION

The Common Assembly Language/32 (CAL/32) Assembler symbolically references numerical constants and constant displacements within a data structure.

Code written using symbolic references is easier to update than numeric code. In addition, symbolic references are easier to use because it is easy to remember a name. Errors involving references are less likely because an inaccurate numerical constant can still be assembled while an inaccurate symbolic name is recognized as an undefined symbol that cannot be assembled. Assembly language symbolic names can reflect the meaning of the numerical constant represented or the name of the field the symbol points to within a structure.

Numerical constant symbols and displacement symbols are used throughout OS/32 coding. These same symbols and structures are used equally by assembly language programmers writing programs to run under OS/32. The collection of symbols and structures related to OS/32 is contained in the operating system structure macro library supplied with the system source (filename SYSSTRUC.MLB on disk). Individual structures can be included in the user task (u-task) by calling the appropriate macro.

4.2 EXAMPLES USING SYSTEM MACROS

The following example shows task status word (TSW) construction. The status portion of a TSW enabling trap wait, task queue service trap, queue entry on task call (queue parameter from another task), and I/O proceed termination can be written as follows:

Example:

```
    $TSW
    .
    .
    .
    DC    TSW.WTM!TSW.TSKM!TSW.TCM!TSW.IOM
```

This sequence instructs the assembler to perform a logical OR operation on four masks, each mask setting a particular bit, to form a word with all the required bits set. Within the data structure, symbols defining bits are in two forms:

- Symbols ending in M have the value of the bit mask needed to enable a particular bit.
- Symbols ending in B have the value of the bit position.

The following example illustrates loading a TSW into the user dedicated location (UDL). A TSW, enabling queue entries on I/O proceed termination and timeout completion, is loaded into the task queue service new TSW field of the UDL. This allows queue entries for these two events to continue while the task queue is being serviced. The TSW contains a location counter field pointing to the task queue service routine.

Example:

```
            $UDL
            .
            .
            .
            LM     R14,TSKQTSW
            STM    R14,UDL.TSKN
            .
            .
            .
QSERVICE    EQU    *
            .
            .
            .
TSKQTSW     DC     TSW.IOM!TSW.TMCM,QSERVICE
```

The UDL field can be referenced by its displacement within the UDL alone rather than in combination with a pointer to the beginning of the UDL. This is because, in an OS/32 user task, the UDL begins at address 0 within the task program address space. The one line definition of the TSW (TSKQTSW) generates two fullwords: a status portion, enabling certain bits, and a location counter (LOC) portion, pointing to the task queue service routine as QSERVICE.

The following example illustrates loading an SVC 6 function code into an SVC 6 parameter block. A function code specifying load and start immediately for some other task, as opposed to a self-directed SVC 6, is loaded into an SVC 6 parameter block.

Example:

```
            $SVC6
            .
            .
            .
            LI      R1,SFUN.DOM!SFUN.LM!SFUN.SIM
            ST      R1,PARBLK+SVC6.FUN
            .
            .
            .
PARBLK      EQU     *
            DS      SVC6.
```

The DS SVC6. instruction reserves the proper amount of space for the SVC 6 parameter block. SVC6 is the label of the structure defining the SVC 6 parameter block, and is set by CAL/32 to the size of the data structure defined by the $SVC6 macro.

A field within a data structure can be referenced using a structure by directly referencing a field.

Example:

```
            $SVC6
            .
            .
            .
            ST      R1,PARBLK+SVC6.FUN
            .
            .
            .
```

The field also can be referenced using an index register.

Example:

```
            $SVC6
            .
            .
            .
            LA      R3,PARBLK
            .
            .
            .
            ST      R1,SVC6.FUN(R3)
            .
            .
            .
PARBLK      DS      SVC6.
```

The direct method does not use an additional register, while the index register method passes the address of different parameter blocks through a register (perhaps to a subroutine).

The previous examples deal with applications where a program dynamically loads various fields with appropriate values. This approach is correct when the contents of the various fields change with time and must be dynamically initialized and subsequently changed. However, for applications where the contents of fields are static, there is an alternative that permanently assembles the appropriate values into data structures. This alternative saves both assembly code size and execution time.

This example shows how to assemble values into a UDL. Assembling a UDL into a task requires an OPTION WORK=n command when linking.

Example:

```
            $UDL
            .
            .
            .
TUDL        EQU   *
            ORG   TUDL+UDL.TSKQ
            DC    task queue addr
            ORG   TUDL+UDL.PWRN
            DC    status,loc
            ORG   TUDL+UDL.S14N
            DC    status,loc
            ORG   TUDL+UDL.TSKN
            DC    status,loc
            ORG   TUDL+UDL
```

The label TUDL is used to avoid conflict with UDL which is defined in the UDL structure by the $UDL macro. To omit any of the field definitions from the code, delete the ORG corresponding to the relevant field and the constant definition for that field. ORG TUDL+UDL sets the location counter past the end of the UDL.

This example shows how to assemble taskid and function code into an SVC 6 parameter block.

Example:

```
            $SVC6
            .
            .
            .
            ALIGN 4
PARBLK      EQU   *
            ORG   PARBLK+SVC6.ID
            DC    C'TASKA'                  TARGET TASKID
            ORG   PARBLK+SVC6.FUN
            DC    SFUN.DOM!SFUN.SIM         START IMMED. FUNCTION
            ORG   PARBLK+SVC6.
```

ORG PARBLK+SVC6. sets the location counter past the end of the parameter block.

CHAPTER 5
VOLUME, FILE, AND DEVICE INFORMATION

5.1 INTRODUCTION

To provide device independent I/O, programs direct all I/O requests to a logical unit (lu) rather than to a specific device or file. The system maintains an lu table (LTAB) for each task. The lu numbers, which range from 0 to 254, correspond to entries in the task LTAB. Task logical units must be assigned to specific devices or files by an operator, multi-terminal monitor (MTM) command, or a supervisor call 7 (SVC 7) (via macro) prior to their use. Devices can be marked offline, making them unavailable for assignment by user tasks (u-task). All OS/32 supported direct access devices can be accessed through the file manager which provides volume and file management services.

Data on a direct access device is organized into a series of files on a named logical volume. When a direct access device is made available by the operator command MARK ON, the name of the volume mounted on that device is associated with the device and refers to it. A MARK OFF command allows the mounted disk volume to be removed from the device list. A disk mounted without first being marked offline can be marked online only in a write-protected mode. If a disk is not marked offline before dismounting, further writing to the disk might make any information in an indeterminate state unrecoverable.

Before using a direct access volume, the appropriate disk formatter program must format it, and the OS/32 Disk Initializer Utility must initialize it. In addition, the Disk Initializer can send an operating system image to a direct access volume for BOOT loading.

5.2 VOLUME ORGANIZATION

A direct access volume under OS/32 contains several data structures:

- Volume descriptor
- Sector allocation (bit) map
- File directory
- Contiguous file type
- Indexed file type

A u-task cannot access the first three structures. The operating system uses them to control the rest of the storage on the volume.

The bulk of the storage area is used for file storage. All the files on the volume are indexed or contiguous file types. The amount of storage on the volume is the only limit on the number of files on the volume. A task, macro, or an operator or MTM command initiates file allocation, assignment, and deletion. Generally, file storage is permanent. A file remains on the volume until it is deleted.

For applications requiring temporary storage, OS/32 also supports temporary files. The temporary file is like any other file except that when a temporary file is closed, it is automatically deleted. A temporary file can be a contiguous or indexed file type.

5.2.1 Volume Descriptor and Sector Allocation Map

Sector 0, cylinder 0 of a disk volume contains the volume descriptor. The volume descriptor has six fullword fields. See Figure 5-1.

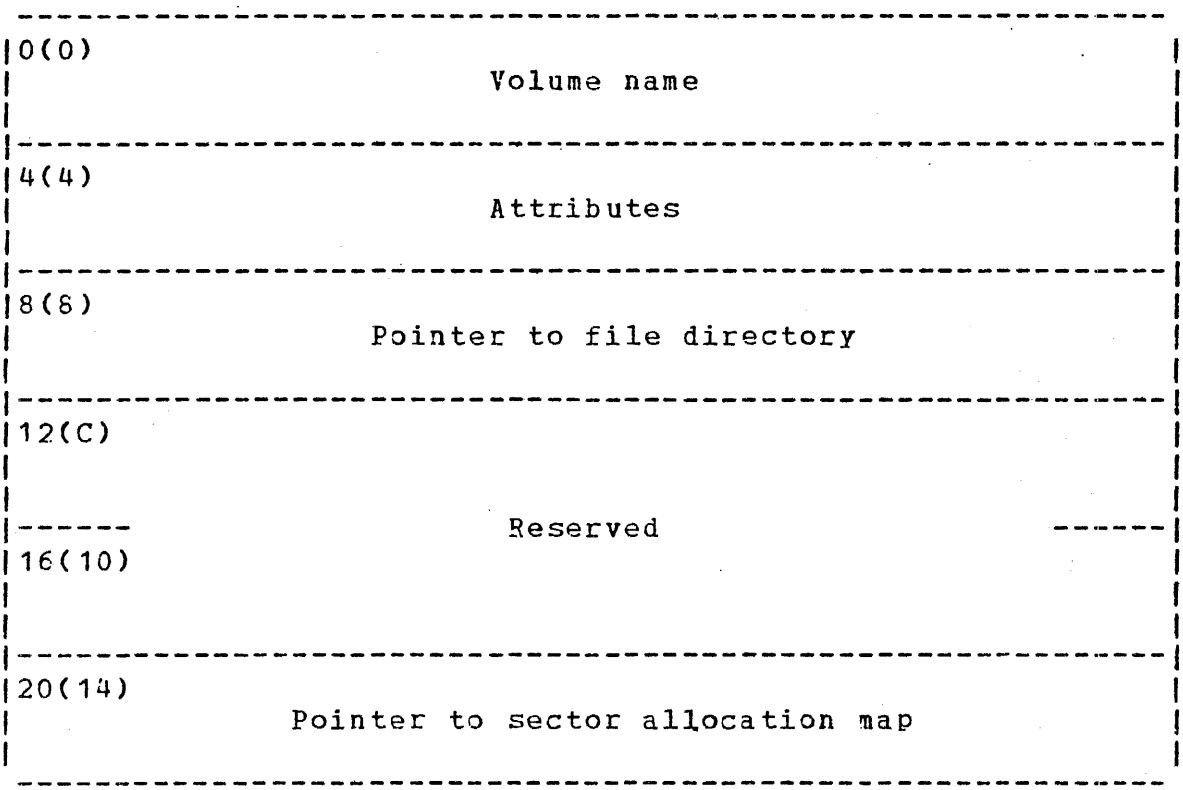


Figure 5-1 Volume Descriptor Structure

Fields:

Volume name	is the volume name field that contains a 4-character ASCII volume identifier, the name by which the volume is known to the system.
Attributes	when the disk is marked off, the attributes field is 0; when the disk is marked on, bit 0 of the attribute field is set to 1.
Pointer to file directory	contains the address of the first sectors of the primary file directory. The file directory contains information the system needs to process files recorded on the volume. An entry in the directory is made for each file.
Reserved	is an 8-byte field reserved for future use.
Pointer to sector allocation map	contains the address of the first sectors of the sector allocation map. The allocation map is a bit map containing one bit for each sector on the volume. This map records allocated, unallocated, and defective sectors. If a sector is allocated or defective, its corresponding bit in the allocation map is set to 1; if unallocated, it is set to 0. The Disk Initializer Utility initially places all data in the volume descriptor. OS/32 does not modify any portion of the volume descriptor, except to indicate the online/offline state of the volume.

5.2.2 Primary and Secondary File Directory

The file directory consists of two sections:

- Primary directory
- Secondary directory

The primary directory is organized as a forward-linked list of 1-sector blocks on a disk. A directory block contains space for five file entries. Each entry contains information about the file: name, type, length, protection keys, data created, and data written. Since only one directory block can be in memory at one time, only five file entries can be memory resident. The other directory blocks remain on disk and are accessed by I/O operations.

To reduce file search time, a secondary directory is available as a system generation (sysgen) option. The secondary directory, portions of which are in memory, is a contiguous file with the reserved name SYSTEM.DIR. This directory contains filenames and primary directory block pointers for all the files on the volume plus available slots (an expansion factor) for a user-defined number of files yet to be allocated (default 100). The secondary directory is organized so that 20 file entries are contained within each sector. The number of secondary directory sectors resident in memory is dependent upon the parameters set by the MARK ON command.

If a secondary directory runs out of free file slots, the operator can choose to continue with a mixture of primary and secondary directories (at a cost in file search time), or mark the disk off and then mark the disk on with an additional expansion area.

5.3 FILE TYPES

OS/32 supports two file types:

- Indexed
- Contiguous

In most cases, the same data manipulations can be performed on both file types. The choice of file type usually depends on how the data is to be accessed and not the data type to be put in the file. Each file type is optimized for one specific form of access.

File descriptors (fd) are entered in a standard format.

Format:

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

Parameters:

voln: is a 1- to 4-character alphanumeric string specifying the name of a disk volume. The first character must be alphabetic and the remaining, alphanumeric. If the volume name is omitted, the default is the system or user volume.

dev: is a 1- to 4-character alphanumeric string specifying a device name. The first character must be alphabetic ani the remaining, alphanumeric.

filename is a 1- to 8-character alphanumeric string specifying the name of a file. The first character must be alphabetic and the remaining, alphanumeric. If a filename is specified when a device name is specified, the filename is ignored.

.ext is a 1- to 3-character alphanumeric string specifying the extension to a filename.

file class is a 1-character alphabetic string specifying the type of file class in a system running under MTM. The file class types are:

- P for private file
- G for group file
- S for system file

actno is a 1-character string specifying the system account number. If MTM is not being used or if the operator is using the system console, the file class is the account number. If this field is omitted, the default value is 0 or the system account. Any other account must be specified by typing the slash (/) and a decimal number within the range of 0 through 255.

Some Perkin-Elmer programs are shown in Table 5-1 with recommended filename extensions.

TABLE 5-1 PERKIN-ELMER I/O FILENAME EXTENSIONS

PROGRAM OR COMMAND NAME	INPUT EXTENSIONS	OUTPUT EXTENSIONS
BAS325, BAS32D	BAS	BAS
CAL32	CAL	OBJ
COBOL	CBL	CAL
CORAL	CRL	OBJ
F7D	FTN	CAL, OBJ
F70	FTN	OBJ
LIBLDR	OBJ	OBJ
Link	OBJ,SEG	TSK,SEG
SBUILD command		CSS
BUILD command		CSS
CALMAC32	MAC,MLB	CAL
Pascal	PAS	OBJ
Patch	OBJ,TSK,SEG	OBJ,TSK,SEG
RPG	RPG	OBJ

5.3.1 Indexed Files

Indexed files are supported on all disk storage devices. The indexed file is an open-ended file composed of a chain of index blocks and a series of data blocks. The index blocks are linked together and contain fullword pointers to one or more data blocks, depending on the number of blocks in the file. The index and data blocks of the indexed file are transparent to the user.

The user allocates data block size, index block size, and logical record size. These parameters are fixed until the file is deleted. Data block size and index block size are specified in sectors (multiples of 256 bytes). Logical records are physically blocked into data blocks.

An indexed file can be sequentially or randomly accessed. These two access methods can be mixed without closing and reassigning the file. Because of the physical structure of the file, random access is readily performed. For example, to read block 1 and then block 60, the indexed file structure requires an overhead read operation for the index block containing the pointers to blocks 1 and 60.

The open-ended structure of the indexed file allows the file to be sequentially extended by writing a logical record numbered one greater than the number of existing records. If five records are currently in a file, a request to write record 6 causes the file to be extended. However, if there are currently five records, a

request to write record 7 or higher causes an end of file (EOF) status. The file can be updated by writing over an existing record.

Indexed files can have shared write access privileges (SRW, SWO, ERSW) that allow more than one task to concurrently append or update an indexed file. Indexed file I/O returns EOF status if a:

- read sequential operation is attempted at the end of file;
- read random operation is attempted and the logical record number specified is greater than the total number of logical records in the file; or
- write random is attempted and the logical record number specified is greater than the total number of logical records in the file, plus one.

End of medium (EOM) status is returned if a write operation is attempted without enough space on the volume containing the file.

If an I/O error occurs during a read operation, I/O is terminated and the I/O error status is returned to the user. If an I/O error occurs during a write operation, data is not written. The system returns the file to its last known state, adjusts the file information in the FCB, and returns an I/O error status to the user. The user should checkpoint the file and issue a fetch attributes macro (FETATR) to obtain the current status of the file.

A forward file or backward file operation positions an indexed file at the end or beginning, respectively.

ASCII, binary, and image operations all are supported on indexed files. Also supported are test and set, write filemark, forward space filemark, and backspace filemark operations.

The block containing the current record pointer or I/O macro that specifies the start of a logical record is read into a system buffer. The contents of the system buffer are then transferred to the user-specified buffer until the user buffer is full or the number of bytes equal to the logical record length of the file has been moved. When the current record pointer is set to the record following the accessed record, the transfer is complete.

Output operates the same way as input, except that the data is moved from the user buffer to the system buffer. If a current record pointer value of one greater than the last record in the file is specified, a record is appended to the file, thus allowing the file to be extended at any time. If the size of the specified buffer is less than the logical record length, the record is padded on the end with spaces (ASCII format) or zeros (binary format).

The test and set macro, TESTIO, provides record locking to synchronize simultaneous updates.

The advantages of using indexed files are that the user does not have to compute the maximum size of the file and unused space on the volume is available for other files. In most cases, the user should choose an indexed file.

5.3.2 Contiguous Files

The contiguous file is a fixed length file. All blocks of a contiguous file are contiguously allocated adjacently on the volume. The file size, in 256-byte sectors, is specified; and all required space is reserved at allocation time. The system considers each sector (block) a record. Random reads and writes can access any record on the file, regardless of which records were previously accessed, making it possible to write a contiguous file in a random fashion. Random and sequential access can be mixed without closing and reassigning the file. Contiguous files are supported on all devices supported by the moving head or floppy disk drivers, or by a mass storage media (MSM) driver.

Contiguous file I/O is nonbuffered, and transfers of variable amounts of data occur directly between the task buffer and the disk. The user can transfer data in logical records greater or smaller than a sector size. The appropriate sector number must be specified to position the file for random access. All transfers begin on a sector boundary (multiple of 256) and end whenever the number of specified bytes is transferred. Following a data transfer, the file's current sector pointer contains the address of the next consecutive sector. The user should always transfer an even number of bytes to a contiguous file.

The contiguous file supports a pseudo filemark capability that gives it some of the characteristics of a magnetic tape device. The pseudo filemark is an X'1313' at the beginning of a record (block). Ensure that data containing an X'1313' is not inadvertently written at the beginning of a record. On a contiguous file the forward file and backward file operations function as they would on a magnetic tape. That is, the file is respectively positioned forward or backward until a filemark (X'1313') is found. For a backward file operation, the current record pointer is left pointing to the record containing the filemark (X'1313'). For a forward file operation, the current record pointer is left pointing to the record following the filemark. The write-filemark results in writing X'1313' at the beginning of the current record. The rest of the record is left in an undefined state.

The shared write access privileges (SRW, SWO, ERSW) are permitted on contiguous files and allow more than one task to append or update a contiguous file concurrently. ASCII binary and image operations are all supported on contiguous files. Also supported are test and set, wait, unconditional and conditional proceed, rewind, backspace record, and forward space record operations.

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 loss of file space on a volume.

5.4 FILE STORAGE

Both indexed and contiguous files can be stored as temporary, permanent, or spool files.

5.4.1 Temporary Files

Temporary files are used for storage of temporary data. The TEMPFILE command allocates and assigns temporary files. See the OS/32 Operator Reference Manual and the OS/32 Multi-Terminal Monitor (MTM) Reference Manual.

Temporary files are allocated on the default temporary volume which is established by the operator VOLUME command. Temporary files are given a special filename consisting of the ampersand character (&) and the date and time of allocation. These files exist only as long as they are assigned and are deleted when the assignment is closed.

5.4.2 Permanent Files

Permanent files are created whenever indexed and contiguous files are created and the TEMPFILE command is not specified. Permanent files are deleted only if explicitly deleted by the operator or user through the DELETE command. Files are allocated on the default system or user volume.

5.4.3 Spool Files

Spool files are created when a task assigns a pseudo printer device. Output is sent to the spool file and queued to a slow speed output device. Spool files are given a special file name consisting of the at sign character (@) followed by eight digits assigned by the Spooler.

5.5 BUFFER MANAGEMENT

OS/32 supports two buffer management methods:

- Buffered logical (BL)
- Unbuffered physical (UP)

5.5.1 Buffered Logical (BL)

Indexed files use the buffered logical (BL) management method. This method divides files into logical records. The logical record length for any given file is fixed when the file is allocated, thus becoming a permanent attribute of the file. It would be impossible to write 20-byte records on a file one time and write 80-byte records on the same file later.

It is possible to read or write less than a logical record. However, this wastes space because the file is physically divided into logical records of the size specified when the file was allocated. Also, it is not possible to write variable length records on a file without wasting space. In this case, the logical record length specified at allocation time must be the size of the longest record the user will ever write on that file. If the user tries to read or write a record that is longer than the file logical record length, data is lost on a write operation or is not returned on a read operation.

The BL method packs logical records into physical blocks as efficiently as possible, allowing logical records to overlay into the next physical block if necessary. The logical record size can exceed the size of a physical block. The only restriction on logical record size is that no logical record can exceed 65,535 bytes.

In the BL method the current record is interpreted as a logical record and not as the physical block number. All logical/physical transformations are handled automatically by the BL method. When a block is read or written, the actual data transfer takes place between the device and a buffer in system space. This buffer is not accessible to the user program. When a task reads or writes a record, data is transferred between the task and the system buffer. Physical reads and writes take place only when required. All actions of the buffer management method are transparent to the user.

5.5.2 Unbuffered Physical (UP)

The unbuffered physical (UP) management method, used by contiguous files, works directly with physical blocks. Data is directly transferred from a buffer in the user program to the device, without being moved into a system buffer. For a write operation, data is moved from the user program directly to the file or device.

In the UP method, the current record pointer points to the current physical block. All data transfers must begin on a physical block boundary. The length of data to be transferred can be less or larger than the length of a physical block, but not larger than the total size of the file. With contiguous files, the current record pointer can be incremented by more than one.

An advantage of the UP method of transfer is that the time required for moving data between a system buffer and user space is eliminated. The primary disadvantage is that space on the disk volume is often wasted.

5.6 FILE ACCESS METHODS

OS/32 supports two methods of file access:

- Random access
- Sequential access

These access methods can be intermixed without closing and reopening the file.

The current record pointer is a number from 0 to the number of logical records currently in the file, indicating the record to be read or written on the next sequential access. Each record is numbered in sequence, starting with 0. The current record pointer is adjusted in one of these ways:

- It is set to 0 by:
  - Rewinding
  - Backspacing to filemark (except on contiguous files where the record pointer is positioned at the record containing the previous pseudo filemark)
  - Assigning (except for write access only)
- It is set to the number of records in the file (the proper position to append new records) by:
  - Assigning for write access only
  - Forwarding to filemark (except on contiguous files where the record pointer is positioned after the record containing the next pseudo filemark)
- It is decremented by one by a backspace record operation, unless the file is already positioned at its beginning.
- It is incremented by one as follows:
  - Forward record (unless already at EOF)
  - Sequential read or write to an indexed file

- A random read or write sets the current record pointer to a value one greater than the record read or written.

- It is incremented by the number of sectors that must be accessed to satisfy a sequential read or write request to a contiguous file.

5.6.1 Random Access

For random access, the user supplies the record number to be accessed. If this record is found, the data transfer is performed, and the current record pointer is set to point to the next sequential record. If the user continues to use random access, the current record pointer is ignored, since it is readjusted on every call. However, the user can read or write a sequence of records, starting with a known record number. In this case, a single random call followed by a number of sequential calls can be used.

Any record allocated for a contiguous file can be read from or written to during random access. When indexed files are randomly accessed, only records currently in the file can be updated. In addition, index files must be sequentially expanded. If the record number specified is more than one record past the end of the file, the call is rejected with EOF status. For example, if the file has only five records, a sixth could be added; but record number 100 could not be added.

With contiguous files, there is no restriction on using random write or read access. Any record within the file's allocation can be read or written.

5.6.2 Sequential Access

When the user accesses the files using the sequential method file, records are read or written in sequence. The current record pointer is automatically adjusted at each access. The rewind, forward record, backward record, forward file, and backward file operations can reposition a file as described.

5.7 CHOOSING FILE TYPES

Follow these guidelines to choose a file type:

If pseudo filemarks are required for magnetic tape emulation, the contiguous file structure is required. This can occur when magnetic tape-oriented programs are used.

If the maximum length of the file is completely unknown, only an indexed file can be used.

Most applications define files so that a logical guess of each data structure's maximum length is generally possible. Assuming the disk is not badly fragmented, the contiguous file can be considered. If the disk is fragmented, the allocation of a large contiguous file might not be possible. Use the OS/32 Disk Backup Utility to compress the disk space and eliminate fragmentation.

If all or most of the file data is to be sequentially accessed, choose the indexed file structure. Long files randomly accessed require the contiguous file structure.

For most applications, choose the indexed file because the indexed file can perform random and sequential operations. However, the index file uses an extra sector as an index block for every 62 data blocks.

Once the indexed file structure is chosen, the physical block size must be selected. Reasons were given for keeping the physical block size small. However, a large block size can be very helpful in some cases. The main time factors involved in a disk access are seek time (for moving-head disks) and rotational latency time. Usually these times overshadow the actual data transfer time. Therefore, transferring two or more sectors generally costs little more time than a transfer of only one sector. For this reason, the number of disk accesses is the critical figure in computing file access time. A large physical block size can reduce the number of accesses. Consider the performance of the overall system. If a given task is not critical or is running in a single task environment, a large physical block size might reduce running time.

If access speed is paramount and the file size is fixed, use the contiguous file structure because the amount of system overhead needed to access contiguous files is less than for any other file type.

It is possible to write programs that use both of these file structures as well as previously existing file structures from other programs. The user can use these programs to test the application to determine which possible file structures are most efficient.

The contiguous file is compatible with the indexed file, provided that the contiguous file does not use the pseudo filemark capability.

5.8 FILE AND DEVICE PROTECTION

Files and devices can be statically and dynamically protected.

5.8.1 Static Protection Using Read/Write Keys

Each file or device has associated with it two protection keys, one for read access and one for write access. Each key is one byte long and has a value from X'00' to X'FF'. If the values of the keys are within the range X'01' to X'FE', the file or device cannot be assigned for read or write access unless the operator or requesting task supplies the matching keys. If a key has a value of X'00', the file or device is unprotected for that access mode. Any key supplied by the operator or requesting task is accepted as valid. If a key has a value of X'FF', the file is unconditionally protected for that access mode. It cannot be assigned for that access mode to any u-task, regardless of the key supplied. An unconditionally protected file can be assigned to an executive task (e-task). Table 5-2 lists the read/write keys used for static protection.

TABLE 5-2 READ/WRITE KEYS

WRITE KEY	READ KEY	MEANING
00	00	Completely unprotected
FF	FF	Unconditionally protected (used by e-tasks)
07	00	Unprotected for read, conditionally. Protected for write (user must supply write key = X'07').
FF	A7	Unconditionally protected for write, conditionally protected for read. User must supply read key of X'A7'.
00	FF	Unprotected for write, unconditionally protected for read
27	32	Conditionally protected for both read and write. User must supply both keys.

The file protection keys are defined when the file is allocated. The system operator or any task having that file assigned for exclusive access can change the protection keys. See Section 5.8.2. The protection keys are changed via the REPROTECT command or a REPROT macro. The device protection keys are defined at sysgen time, and only the system operator can change them.

5.8.2 Dynamic Protection Using Access Privileges

By assigning exclusive access privileges to a file, other tasks are prevented from accessing that file. These privileges remain in effect as long as the file is assigned to them. 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)

A file cannot be assigned with an access privilege incompatible with an existing assignment of that file. For example, a request to open a file for EWO is compatible with an existing assignment of that file for SRO or ERO, but is incompatible with any existing assignment for other access privileges. Table 5-3 shows which access privileges are compatible. If the user attempts to change access privileges by adding a privilege that is incompatible with existing ones, the old access privileges remain.

TABLE 5-3 ACCESS PRIVILEGE COMPATIBILITY

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

LEGEND

* Compatible
\- Incompatible

Exclusive access was discussed in terms of multiple tasks sharing the same file, assuming that a single task does not attempt to assign the same file to multiple logical units. However, occasionally the same file is assigned to multiple logical units as a result of default assignments or system operator assignments. In this case, access privileges to the file must be assigned to the lu as if it were a task. For example, a file cannot be assigned for exclusive read access on one lu and shared read on another. If a file is assigned for exclusive read or write access on any given lu, it cannot be assigned for that access on any other lu.

5.8.3 Write-Protected Volumes

Mark the disk online as a protected device to protect all files on a disk from write operations. When a volume is write-protected, only assignments for SRO and SRW are accepted; SRW is changed to SRO. If the hardware write-protected feature of a disk is enabled, the volume must be marked on as a protected volume. Refer to the OS/32 Operator Reference Manual.

5.8.4 Static and Dynamic Protection Modification

The system operator or any task having that file assigned for exclusive access can change a file's protection keys. If the task file is assigned for exclusive write, the write key can be changed; if the task has the file assigned for ERW, it can change either or both keys.

Under the proper conditions, a task can change its file access privileges without having to close the file. For example, a task having a file assigned for shared read cannot change to exclusive read if the file is also assigned for shared read to another task (or another lu of the same task). The CHANGE ACCESS PRIVILEGES (CHAP) macro changes the access privilege. The user cannot change from read only or write only to read/write, from read only to write only, or from write only to read only. If the user attempts to change access privileges and is unable to get the new privileges, the old access privileges remain.

5.9 FILE MANAGEMENT

This section discusses the ALLOCATE, ASSIGN, close delete (CLDE), and checkpoint (CKPOINT) macros.

5.9.1 File Allocation

When a file is allocated, its directory entry is built; if it is a contiguous file, space is reserved for it on the disk. A file can be allocated from the system console or from a user program via an ALLOCATE macro. Regardless of how a file is allocated, the following information must be specified:

Volume id	specifies the volume on which the file is to be allocated. It must be the name of an online direct access volume, otherwise volume error is returned.
Filename/extension	gives a name to the newly allocated file. There must not be any other file of that name and extension on the specified volume, otherwise an error status message is returned.
Write key/read key	sets up the initial protection keys for the file. If this field is not set, the default is an unprotected file.
Logical record length	is the field used when allocating indexed files. This sets the logical record length for the file. It can be any size up to 65,535 bytes; however once established, it cannot be changed. Specifying zero record length is illegal and will result in an error status.

Size	is the size of the entire file in sectors for a contiguous file. It can be any size up to the maximum contiguous space available on that volume at that time. If the size requested is too large, a message is returned. For an indexed file, this size is the physical block size for the file in sectors. It can be any size up to a maximum set at sysgen time for the system (never greater than 255). If this parameter is too large, it can be difficult to open the file.
File type	Indexed or contiguous.

5.9.2 File Assignment

The ASSIGN (ALAS) macro or the ASSIGN command assigns a file to an lu. At this point, the desired access privileges must be specified; the read key must be given if read access is requested; the write key must be specified if write access is requested.

When a file is assigned, the system allocates within the system space a file control block and buffers. Any buffer space required depends on the chosen buffer management method and the physical block size of the file. If the file's physical block size is too great for the remaining system space, the file is not opened and a buffer error status is returned. When a buffer error occurs, the user program can close another open file assignment, freeing some system space and allowing the first file assignment to be retried. For this reason, do not keep files open longer than necessary. The physical block length of the file should also be kept as short as possible unless there are other overriding considerations.

5.9.3 File Deassignment (Close)

Issuing a CLOSE command closes a file assigned to an lu. Information other than the lu assignment need not be specified. The system waits for any incomplete write data transfers to terminate, writes out to the volume any partially filled buffers, and updates the file directory entry. The lu that the file was assigned to is closed.

5.9.4 File Deletion

A file can be deleted only if it is not currently assigned to a task. When using the DELETE macro or command to delete a file, the user must supply the volume name as the default volume or

supply the volume name along with the filename and extension. When the file is deleted, the directory entry is removed and the deleted file's disk space is made available to other files.

5.9.5 File Checkpointing

The checkpoint macro (CKPOINT) performs the buffer clearing and directory updating functions of a CLOSE macro or command without closing the lu. This operation is a protective operation to guard against system failure for very critical files or for files running for lengthy periods.

If the system fails, data appended to a file after the latest close or checkpoint operation is lost for certain files and buffer management methods because the directory is only updated at close or checkpointing time. If the system failure does not corrupt the volume directory or the physical media, all data appended before the file was closed or checkpointed is guaranteed safe.

INDEX

G H

I J K

L

M N